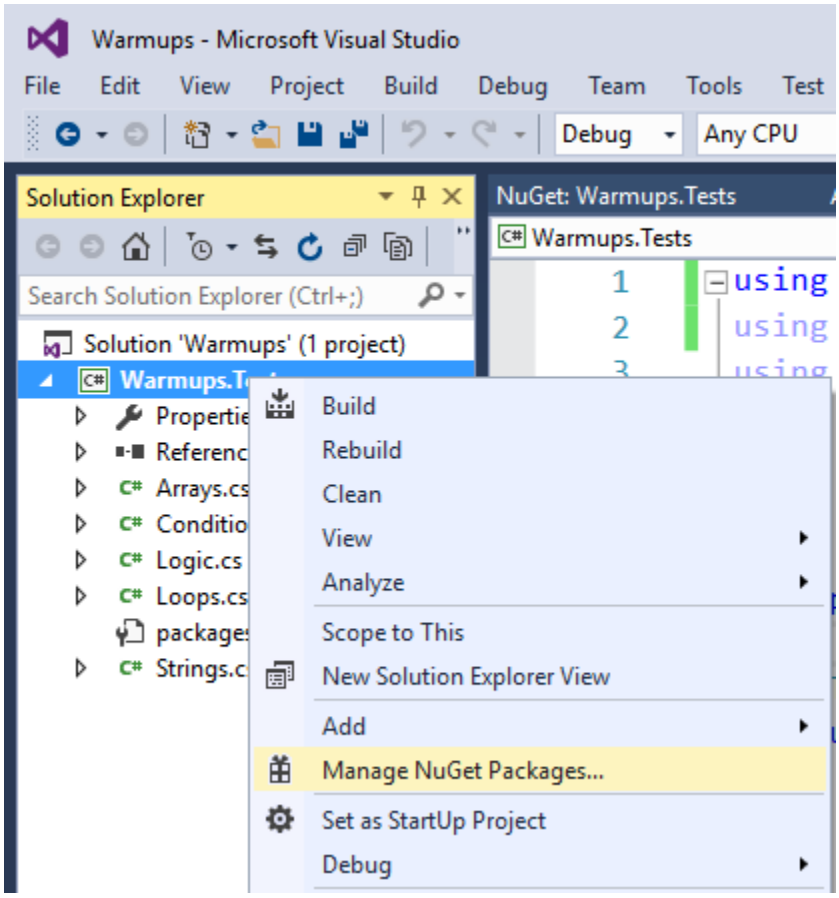
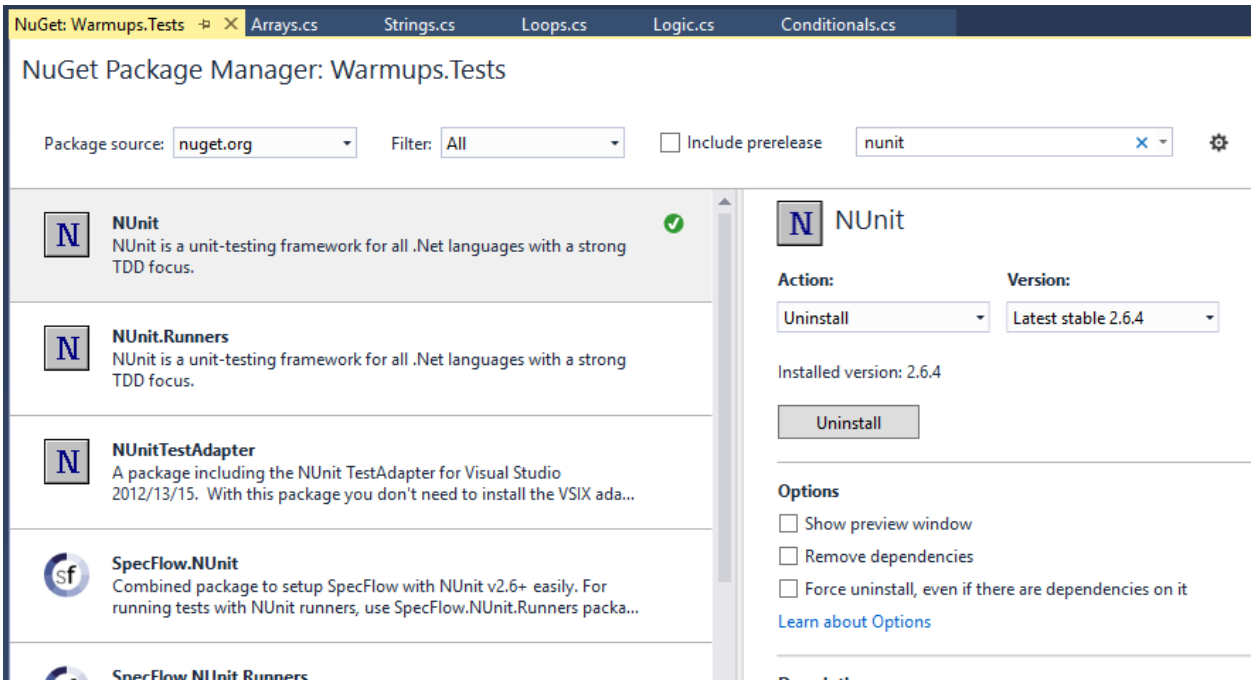
NUnit Warmup Basics

NUnit is a 3rd party framework that allows developers to write Unit Tests for their code. A unit test is a method (or series of methods) that runs **assertions** against the code you write to ensure correctness. The sample project I have given you has the NUnit Framework installed via the NuGet utility in visual studio.

NuGet is a package manager that allows developers to create and share .NET code with the world. To access NuGet, simply right click on a project or solution and choose the "Manage NuGet Packages" menu option:



This will open a search window that allows you to download packages from the internet. Notice here that I have searched for NUnit:



You can click any package on the left and choose install. Note that I have already done this for you, so you should only need to Build the project and the package manager should re-download the packages from the internet. This is so if you want to add NUnit to future projects you can.

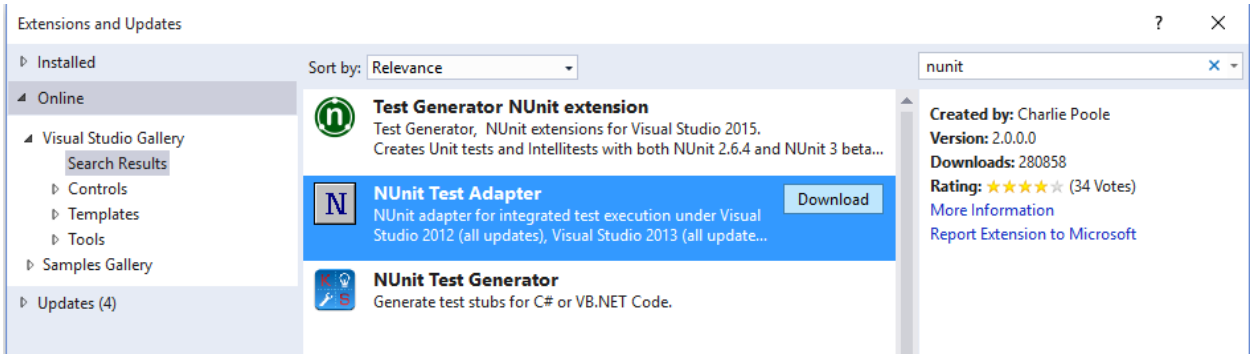
Unit testing follows what I call the triple-a methodology:

- **Arrange-** Set up all the data and instantiate your objects.
- **Act**- Call the methods and run your logic.
- **Assert**- Verify that the results that came back are the correct ones.

I will walk you through the creation of a test that verifies a method that adds two numbers and returns the sum. First though, we need to install an extension to Visual Studio that will allow us to run our NUnit tests from within Visual Studio.

Installing the NUnit Test Adapter

In Visual Studio go to the tools menu, and choose “Extensions and Updates”. In the window that appears click on the left sidebar on the “Online” tab and choose “Visual Studio Gallery”. In the search bar in the upper right hand corner type “NUnit” and hit the enter key. Choose the Nunit Test Adapter by Charlie Poole and click download.



It will prompt you to click install when it finishes downloading and then you will need to restart Visual Studio for the extension to take effect. Go ahead and do that.

Writing Our First Test

Let's write a method in our Example.cs class in the BLL (business logic layer) project that takes in two integers and returns their sum. This code is pretty straightforward:

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

To test this, we will want to come up with a few **test cases** for the method. A good test case provides values for the parameters of the method (x and y in this example) as well as the expected value that should be returned. I have decided to verify that if we pass 1 and 1 in, the expected value is 2 and if we pass 5 and 10 in, the expected value is 15.

Unit tests are normally placed in separate projects from our actual code, so I have created a Warmup.Tests project which I installed NUnit into via NuGet. I also created a class called ExampleTests where I will write the test code for the Example class. For NUnit to run tests, we need to decorate the ExampleTests class with an **attribute** called [TestFixture]. Attributes in C# are placed in brackets and attached to definitions in our code. The TestFixture attribute lets NUnit know that this class will have tests in it. Because this is part of the NUnit framework, be sure to put a "using NUnit.Framework" statement at the top of the file. Because we will also be using code from Warmups.BLL, we will add that using statement as well:

```
using NUnit.Framework;
using Warmups.BLL;

namespace Warmups.Tests
{
    [TestFixture]
    public class ExampleTests
    {
```

Next, we will want to write a method that takes in the two parameters required for our method and one extra parameter to hold the value we expect back. I have named these parameters x, y, and expected.

```
public void AddTest(int x, int y, int expected)
{
```

Inside this method, we will arrange, act, and assert. Here is the body of the method.

```
public void AddTest(int x, int y, int expected)
{
    // arrange, instantiate object
    Example obj = new Example();

    // act, call the method
    int actual = obj.Add(x, y);

    // assert, verify correctness
    Assert.AreEqual(expected, actual);
}
```

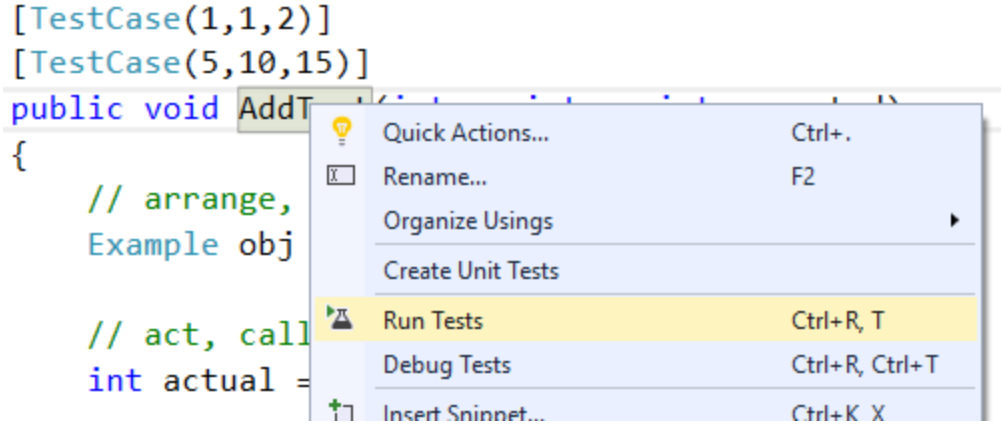
The Assert class is built into the language, and it has many methods on it that help us verify tests. The AreEqual(val1, val2) is probably the most common, but there are many more. Feel free to explore it. The important thing to note though is that it is the Assert statement that determines if a test passes or fails. If you do not put an assertion in your code, the test will not do anything useful!

Finally, we need to get our test data into our code. NUnit provides a [TestCase] attribute which allows us to pass values into the AddTest method. The order you put the values is the order they are passed into the method, so we want to be sure they match up! Here is the final code for the AddTest unit test case:

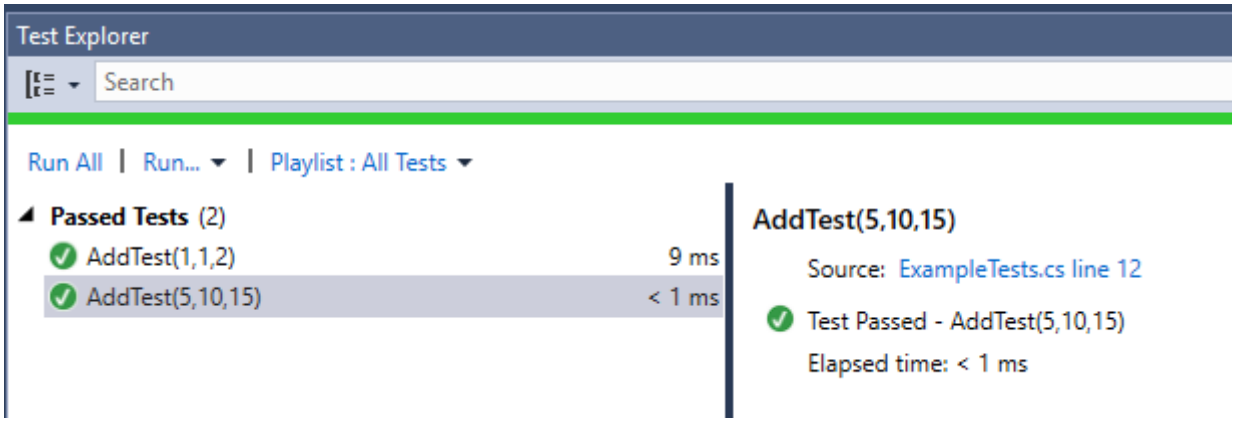
```
[TestFixture]
public class ExampleTests
{
    [TestCase(1,1,2)]
    [TestCase(5,10,15)]
    public void AddTest(int x, int y, int expected)
    {
        // arrange, instantiate object
        Example obj = new Example();

        // act, call the method
        int actual = obj.Add(x, y);

        // assert, verify correctness
        Assert.AreEqual(expected, actual);
    }
}
```

Running Your Tests

At this point, you can right-click the AddTest method and choose the “Run Tests” command.

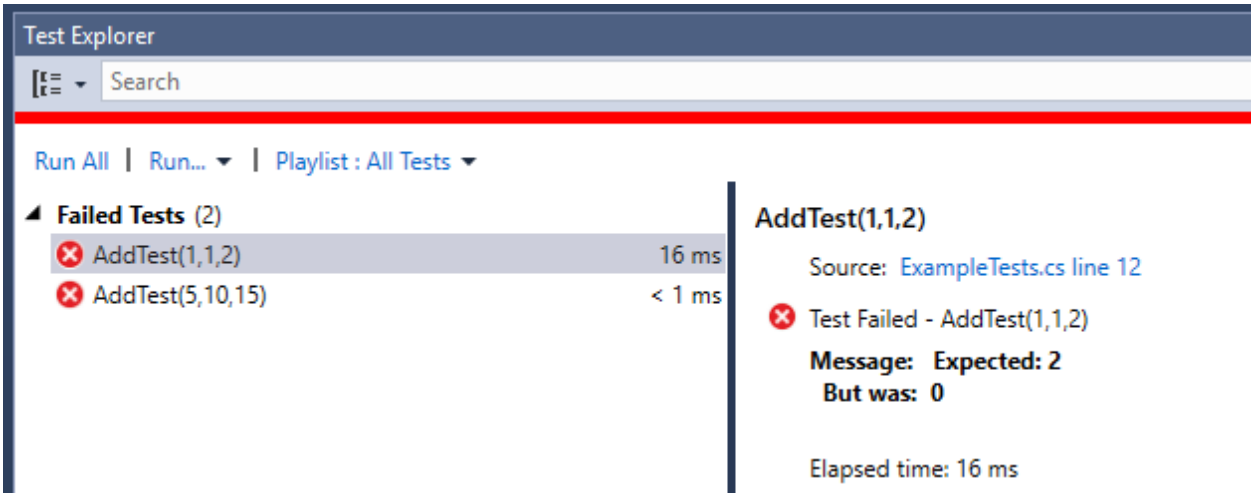


A window called the “Test Explorer” will open, and for each test case it will show a result. A green checkmark next to a test case means the assertion passed while a red mark means that the test failed. In our case, the tests should have come back green. (Note that if you lose the text explorer window, you can get it back by going to the Test Menu->Windows->Test Explorer).



Failing Tests

Let's say in our Add method we accidently subtracted instead of added. In this case our tests would fail and NUnit would report why:



It is actually quite helpful, in the 1+1 test where 2 was expected, it gives us a message that the assertion on line 12 failed. It expected 2 but the value in the actual variable was 0. Now we know we need to go back to our Add method and examine it for issues. Keep in mind one important point though:

If your test data is wrong, your method may be correct, so check both!

I often see students wracking their brains trying to figure out why a test is failing when their method code looks correct, and sometimes the method actually is correct… they just had a typo in their TestCase attribute, like if you had passed 1, 1 and expected 3. So always make sure to double check your test case assertions when the method code looks correct. Just as we can make mistakes in code, we can also make mistakes in writing our tests!

Finishing the Warmups

For your convenience, I have written one of each of the warmup solutions. You can use this as a template for finishing the rest of the warmups. This should not only get you comfortable with unit testing with NUnit, but should also get you comfortable with instantiating class objects and using them in other libraries.

The VisualStudioProject folder has the code files that you can open in Visual Studio 2015 Community Edition. The other folders have the exercises you should work on. Logic and Conditionals are probably the easiest exercises to start with.

Good luck!