Introduction to Unit Testing

Overview

In this lab, you'll learn about unit testing. Unit tests gives you an efficient way to look for logic errors in the methods of your classes. Unit testing has the greatest effect when it's an integral part of your software development workflow. As soon as you write a function or other block of application code, you can create unit tests that verify the behavior of the code in response to standard, boundary, and incorrect cases of input data, and that verify any explicit or implicit assumptions made by the code. In a software development practice known as test-driven development, you create the unit tests before you write the code, so you use the unit tests as both design documentation and functional specifications of the functionality.

Visual Studio has robust support for unit testing, and also supports deep integration with third-party testing tools. In addition, you can leverage the power of Visual Studio Online to manage your projects and run automated tests on your team's behalf.

Objectives

In this hands-on lab, you will learn how to:

- Create unit tests for class libraries
- Create user applications that are highly testable
- Take advantage of advanced Visual Studio features, such as data-driven testing, code coverage analysis, and Microsoft Fakes
- Create a continuous integration environment that automatically runs unit tests upon file check ins

Prerequisites

The following is required to complete this hands-on lab:

- Microsoft Visual Studio 2013
- A Visual Studio Online account (only required for exercise 4)

Exercises

This hands-on lab includes the following exercises:

1. Creating a project and supporting unit tests
2. Unit testing user applications
3. Advanced unit testing features
4. Continuous integration testing with Visual Studio Online

Estimated time to complete this lab: **60** minutes.

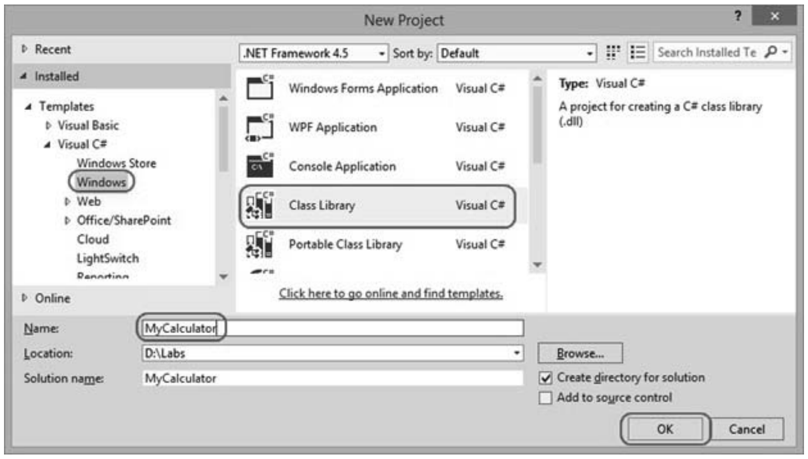
Exercise 1: Creating a project and supporting unit tests

In this exercise, you'll go through the process of creating a new project, as well as some supporting unit tests.

Task 1: Creating a new library

In this task, you'll create a basic calculator library.

1. Open **Visual Studio 2013**.
2. From the main menu, select **File | New | Project**.
3. In the **New Project** dialog, select the **Visual C# | Windows** category and the **Class Library** template. Type **"MyCalculator"** as the **Name** and click **OK**.



New Project

.NET Framework 4.5 | Sort by: Default

Templates > Visual C# > Windows

Class Library — Visual C#

Type: Visual C#

A project for creating a C# class library (.dll)

Name: MyCalculator

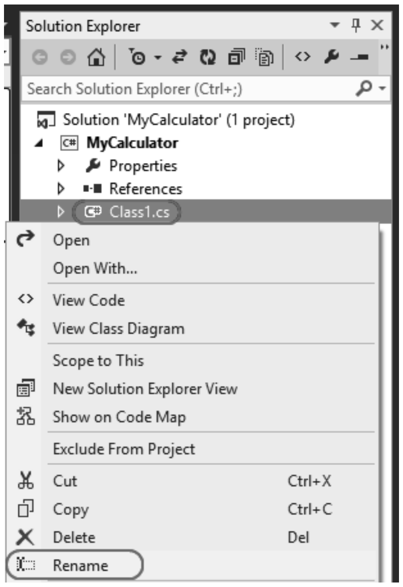
Location: D:\Labs

Solution name: MyCalculator

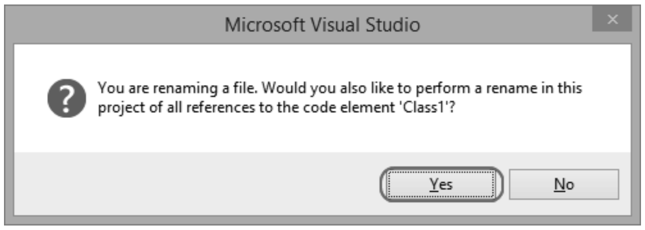
☑ Create directory for solution

☐ Add to source control

4. In **Solution Explorer**, Right-click the **Class1.cs** and select **Rename**. Change the name to **“Calculator.cs”**.



5. When asked to update the name of the class itself, click **Yes**.



Microsoft Visual Studio

You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'Class1'?

6. Add the following **Add** method to **Calculator.cs**.

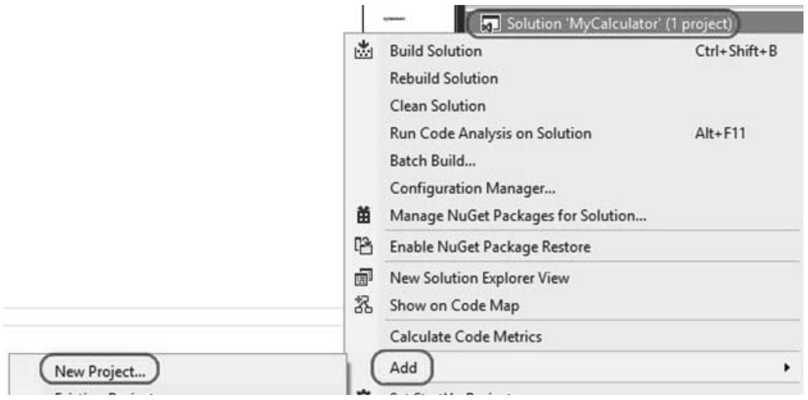
```
public int Add(int first, int second)
{
    return first + second;
}
```

> Note: For the purposes of this lab, all operations will be performed using the **int** value type. In the real world, calculators would be expected to scale to a much greater level of precision. However, the requirements have been relaxed here in order to focus on the unit testing.

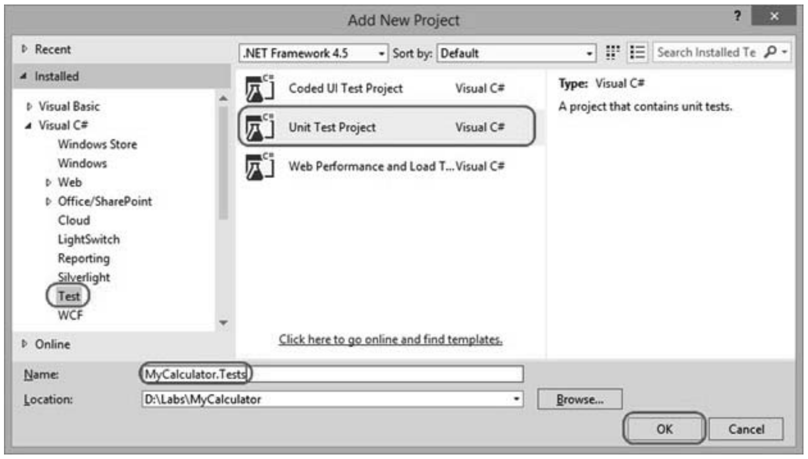
Task 2: Creating a unit test project

In this task, you’ll create a new unit test project for your calculator library. Unit tests are kept in their own class libraries, so you’ll need to add one to the solution.

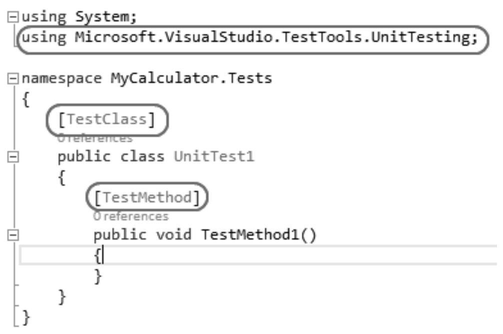
1. Right-click the solution node and select **Add | New Project…**.


Solution 'MyCalculator' (1 project)
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Batch Build...
Configuration Manager...
Manage NuGet Packages for Solution...
Enable NuGet Package Restore
New Solution Explorer View
Show on Code Map
Calculate Code Metrics
Add
New Project...

2. Select the **Visual C# | Test** category and the **Unit Test Project** template. Type **"MyCalculator.Tests"** as the **Name** and click **OK**. It's a common practice to name the unit test assembly by adding a **".Tests"** namespace to the name of the assembly it targets.

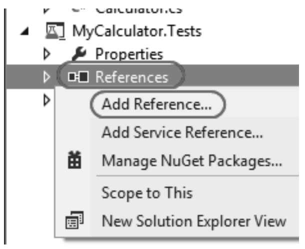


The wizard will end by opening the default unit test file. There are three key aspects of this class to notice. First, it includes a reference to Visual Studio's unit testing framework. This namespace includes key attributes and classes, such as the **Assert** class that performs value testing. Second, the class itself is attributed with **TestClass**, which is required by the framework to detect classes containing tests after build. Finally, the test method is attributed with **TestMethod** to indicate that it should be run as a test. Any method that is not attributed with this will be ignored by the framework, so it's important to pay attention to which methods do and don't have this attribute.

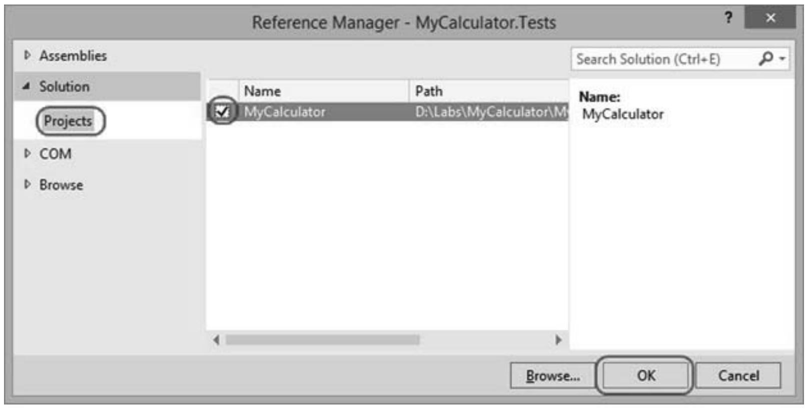


```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MyCalculator.Tests
{
    [TestClass]
    0 references
    public class UnitTest1
    {
        [TestMethod]
        0 references
        public void TestMethod1()
        {
        }
    }
}
```

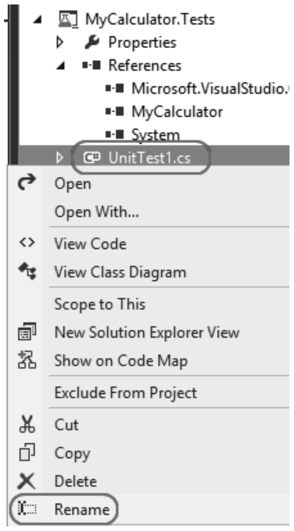
3. The first thing you'll need to do with the unit test project is to add a reference to the project you'll be testing. In **Solution Explorer**, right-click the **References** node of **MyCalculator.Tests** and select **Add Reference…**.



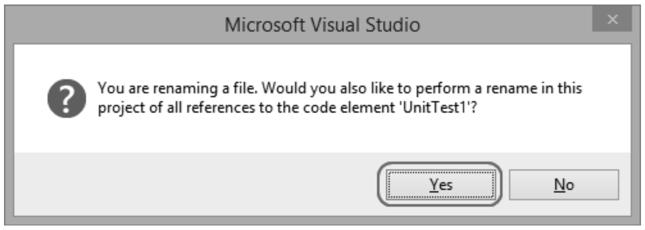
4. Select the **Projects** category in the left pane and check **MyCalculator** in the main pane. Click **OK** to add the reference.



5. To make the project easier to manage, you should rename the default unit test file. In **Solution Explorer**, right-click **UnitTest1.cs** and rename it to **"CalculatorTests.cs"**.



6. When asked to rename the class itself, click **Yes**.



You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'UnitTest1'?

7. At the top of **CalculatorTests.cs**, add the following **using** directive.

```
using MyCalculator;
```

8. Rename **TestMethod1** to **AddSimple**. As a project grows, you'll often find yourself reviewing a long list of tests, so it's a good practice to give the tests descriptive names. It's also helpful to prefix the names of similar tests with the same string so that tests like **AddSimple**, **AddWithException**, **AddNegative**, etc, all show up together in various views.

```
public void AddSimple()
```

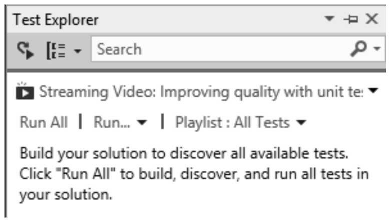
Now you're ready to write the body of your first unit test. The most common pattern for writing unit tests is called **AAA**. This stands for **Arrange, Act, & Assert**. First, initialize the environment. Second, perform the target action. Third, assert that the action resulted the way you intended.

9. Add the following code to the body of the **AddSimple** method.

```
Calculator calculator = new Calculator();
int sum = calculator.Add(1, 2);
Assert.AreEqual(0, sum);
```

This test has only three lines, and each line maps to one of the **AAA** steps. First, the **Calculator** is created. Second, the **Sun** method is called. Third, the **sum** is compared with the expected result. Note that you're designing it to fail at first because the value of **0** is not what the correct result should be. However, it's a common practice to fail a test first to ensure that the test is valid, and then update it to the expected behavior for success. This helps avoid false positives.

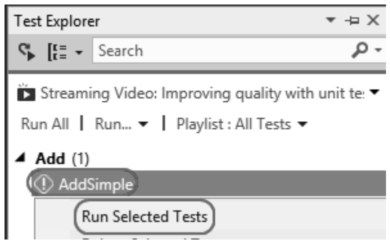
10. From the main menu, select **Test | Windows | Test Explorer**. This will bring up the **Test Explorer**, which acts as a hub for test activity. Note that it will be empty at first because the most recent build does not contain any unit tests.


Test Explorer
Search
Streaming Video: Improving quality with unit te:
Run All | Run... | Playlist : All Tests

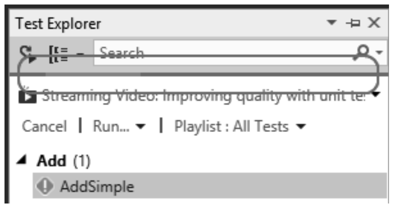
Build your solution to discover all available tests. Click "Run All" to build, discover, and run all tests in your solution.

11. From the main menu, select **Build | Build Solution**.

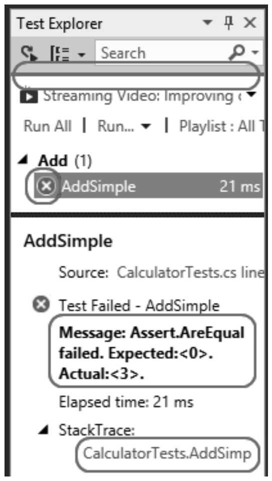
12. The unit test should now appear in **Test Explorer**. Right-click it and select **Run Selected Tests**.


Test Explorer
Search
Streaming Video: Improving quality with unit te:
Run All | Run... | Playlist : All Tests
Add (1)
AddSimple
Run Selected Tests

Note that while the test process is running, the progress bar in **Test Explorer** shows an indeterminate green indicator.


Test Explorer
Search
Streaming Video: Improving quality with unit te:
Cancel | Run... | Playlist : All Tests
Add (1)
AddSimple

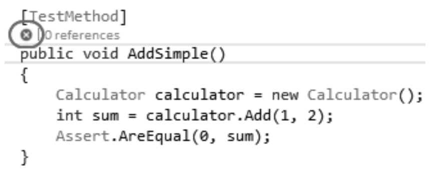
13. However, once the test fails, the bar turns red. This provides a quick and easy way to see the status of your tests. You can also see the status of each individual test based on the icon to its left. Click the test result to see the details about why it failed. If you click the **CalculatorTests.AddSimple** link at the bottom, it will bring you to the method.



Message: Assert.AreEqual failed. Expected:<0>. Actual:<3>.

Elapsed time: 21 ms

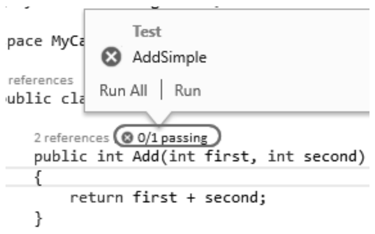
The status indicator also appears in the **CodeLens** directly above the method definition for the test in **CalculatorTests.cs**. Note that **CodeLens** is a feature of Visual Studio Ultimate.



```
[TestMethod]
0 references
public void AddSimple()
{
    Calculator calculator = new Calculator();
    int sum = calculator.Add(1, 2);
    Assert.AreEqual(0, sum);
}
```

And if you switch to **Calculator.cs**, you'll see the unit testing **CodeLens** that indicates the rate of passing (**0/1 passing** here).

14. Click the **CodeLens** to see the results of each test that exercises this method. If you double-click the **AddSimple** test in the **CodeLens** display, it will bring you directly to the test definition.

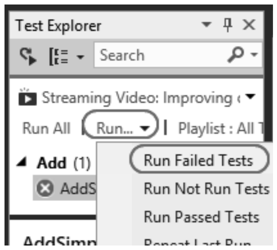


```
pace MyCa
references
ublic cla
    2 references  0/1 passing
    public int Add(int first, int second)
    {
        return first + second;
    }
```

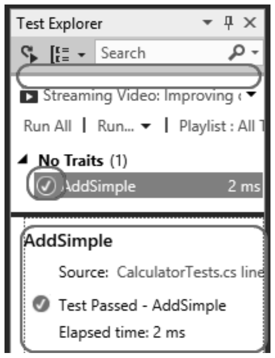
15. In **CalculatorTests.cs**, change the **0** in the **Assert** line to the correct **3**.

```
Assert.AreEqual(3, sum);
```

16. In **Test Explorer**, click **Run | Run Failed Tests**. Note that this option only runs the tests that failed in the last pass, which can save time. However, it doesn't run passed tests that may have been impacted by more recent code changes, so be careful when selecting which tests to run.

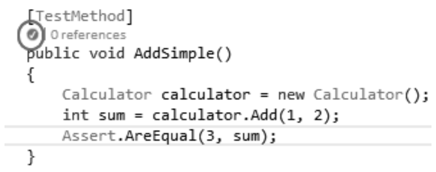


17. Now that the test passes, the progress bar should provide a solid green. Also, the icon next to the test will turn green. If you click the test, it will provide the basic stats for the test's run.



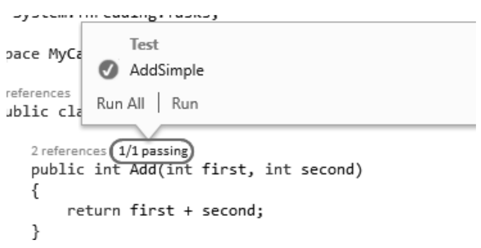
AddSimple
Source: CalculatorTests.cs line
Test Passed - AddSimple
Elapsed time: 2 ms

Also note that the **CodeLens** updates above the test's method definition.



```
[TestMethod]
0 references
public void AddSimple()
{
    Calculator calculator = new Calculator();
    int sum = calculator.Add(1, 2);
    Assert.AreEqual(3, sum);
}
```

As well as in the **CodeLens** above the library method being tested in **Calculator.cs**.



```
System.Threading.Tasks;
pace MyCa
references
ublic cla
    2 references  1/1 passing
    public int Add(int first, int second)
    {
        return first + second;
    }
```

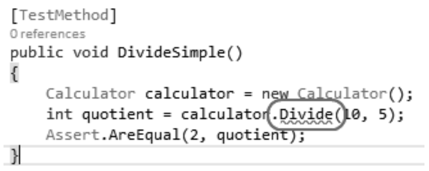
Task 3: Creating a new feature using test-driven development

In this task, you'll create a new feature in the calculator library using the philosophy known as "test-driven development" (TDD). Simply put, this approach encourages that the tests be written before new code is developed, such that the initial tests fail and the new code is not complete until all the tests pass. Some developers find this paradigm to produce great results, while others prefer to write tests during or after a new feature is implemented. It's really up to you and your organization because Visual Studio provides the flexibility for any of these approaches.

1. Add the following method to **CalculatorTests.cs**. It is a simple test that exercises the **Divide** method of the library.

```
[TestMethod]
public void DivideSimple()
{
    Calculator calculator = new Calculator();
    int quotient = calculator.Divide(10, 5);
    Assert.AreEqual(2, quotient);
}
```

2. However, since the **Divide** method has not yet been built, the test cannot be run. However, you can feel confident that this is how it's supposed to work, so now you can focus on implementation.



```
[TestMethod]
0 references
public void DivideSimple()
{
    Calculator calculator = new Calculator();
    int quotient = calculator.Divide(10, 5);
    Assert.AreEqual(2, quotient);
}
```

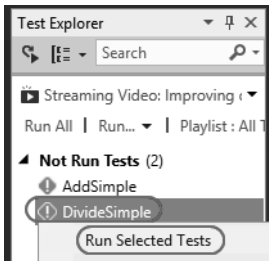
Strictly speaking, the next step should be to implement only the method shell with no functionality. This will allow you to build and run the test, which will fail. However, you can skip ahead here by adding the complete method since it's only one line.

3. Add the **Divide** method to **Calculator.cs**.

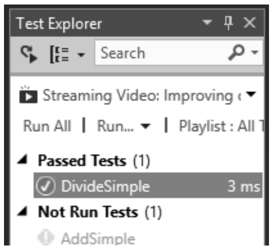
```
public int Divide(int dividend, int divisor)
{
    return dividend / divisor;
}
```

4. From the main menu, select **Build | Build Solution**.

5. In **Test Explorer**, right-click the new **DivideSimple** method and select **Run Selected Tests**.



The test should complete successfully.

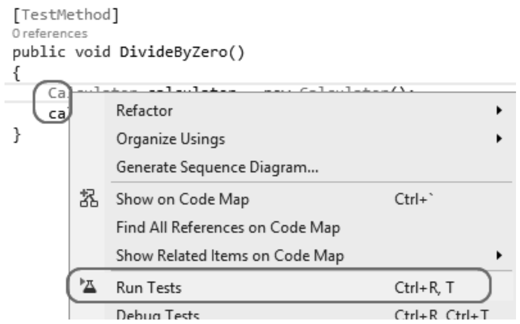


However, there is one edge case to be aware of, which is the case where the library is asked to divide by zero. Ordinarily you would want to test the inputs to the method and throw exceptions as needed, but since the underlying framework will throw the appropriate **DivideByZeroException**, you can take this easy shortcut.

6. Add the following method to **CalculatorTests.cs**. It attempts to divide 10 by 0.

```
[TestMethod]
public void DivideByZero()
{
    Calculator calculator = new Calculator();
    calculator.Divide(10, 0);
}
```

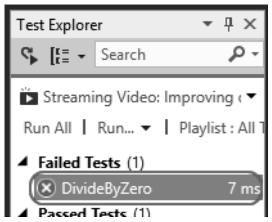
7. Right-click within the body of the test method and select **Run Tests**. This is a convenient way to run just this test.

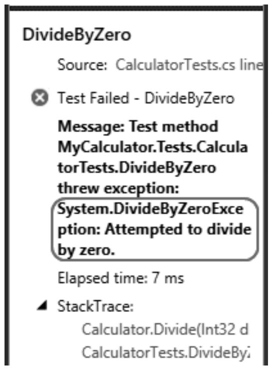


```
[TestMethod]
0 references
public void DivideByZero()
{
    Ca
    ca
}
```

Refactor
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Find All References on Code Map
Show Related Items on Code Map
Run Tests Ctrl+R, T
Debug Tests Ctrl+R, Ctrl+T

8. The test will run and fail, but that's expected. If you select the failed test in **Test Explorer**, you'll see that the **DivideByZeroException** was thrown, which is by design.





DivideByZero

Source: CalculatorTests.cs line

Test Failed - DivideByZero

Message: Test method MyCalculator.Tests.CalculatorTests.DivideByZero threw exception: System.DivideByZeroException: Attempted to divide by zero.

Elapsed time: 7 ms

StackTrace:

Calculator.Divide(Int32 d

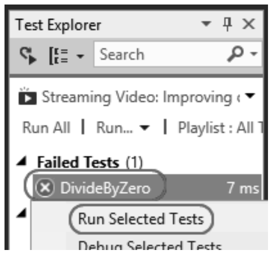
CalculatorTests.DivideBy

However, this is expected behavior, so you can attribute the test method with the **ExpectedException** attribute.

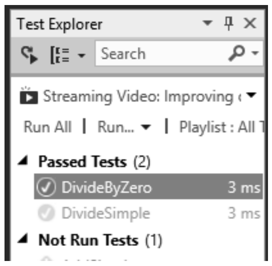
9. Add the following attribute above the definition of **DivideByZero**. Note that it takes the type of exception it's expecting to be thrown during the course of the test.

```
[ExpectedException(typeof(DivideByZeroException))]
```

10. In **Test Explorer**, right-click the failed test and select **Run Selected Tests**.



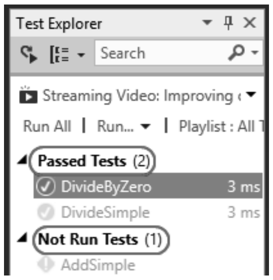
The test should now succeed because it threw the kind of exception it was expecting.



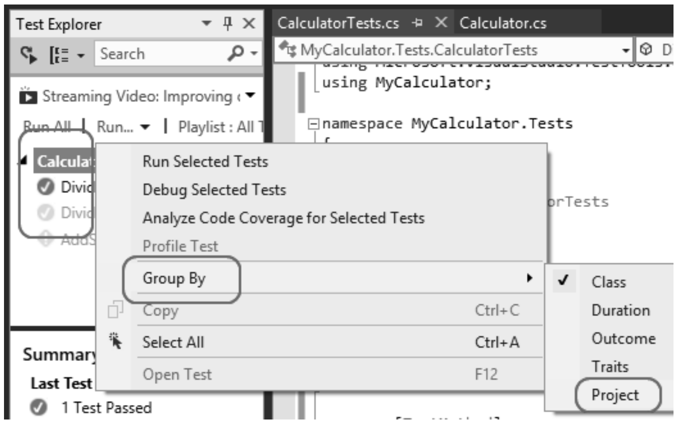
Task 4: Organizing unit tests

In this task, you'll organize the unit tests created so far using different groupings, as well as create custom traits for finer control over organization.

By default, **Test Explorer** organizes tests into three categories: **Passed Tests**, **Failed Tests**, and **Not Run Tests**. This is useful for most scenarios, especially since developers often only care about the tests that are currently failing.

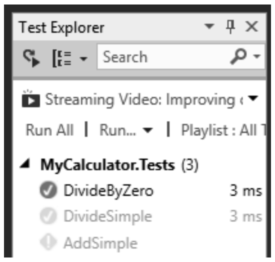


1. However, sometimes it can be useful to group tests by other attributes. Right-click near the tests and select **Group By | Project**.

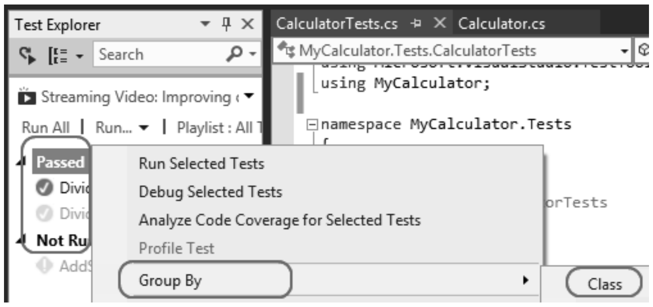


```
using MyCalculator;

namespace MyCalculator.Tests
```

This will organize the tests based on the project they belong to. This can be very useful when navigating huge solutions with many test projects.

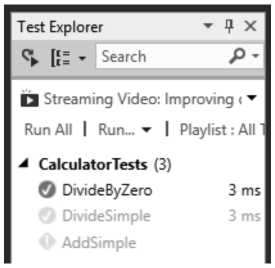


2. Another option is to organize tests by the class they're part of. Right-click near the tests and select **Group By | Class**.

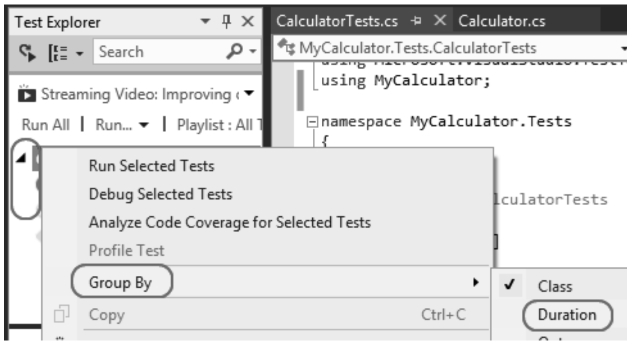


```
using MyCalculator;

namespace MyCalculator.Tests
```

The tests are now grouped by class, which is useful when the classes are cleanly divided across functional boundaries.

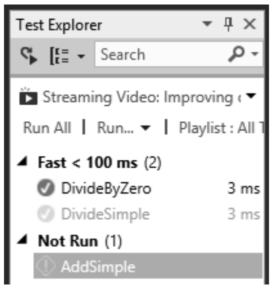


3. Yet another option is to organize tests by how long they take to run. Right-click near the tests and select **Group By | Duration**.



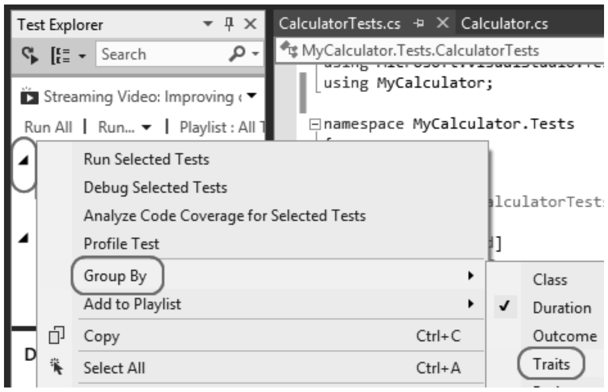
```
using MyCalculator;

namespace MyCalculator.Tests
{
```

Grouping by duration makes it easy to focus on tests that may indicate poorly performing code.



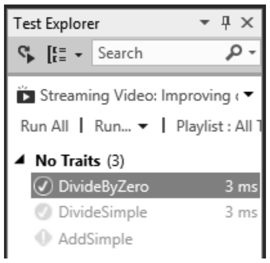
```
Fast < 100 ms (2)
  DivideByZero    3 ms
  DivideSimple    3 ms
Not Run (1)
  AddSimple
```

4. Finally, there is another option is to organize tests by their **traits**. Right-click near the tests and select **Group By | Traits**.



```
using MyCalculator;

namespace MyCalculator.Tests
```

By default, a test method doesn’t have any traits.



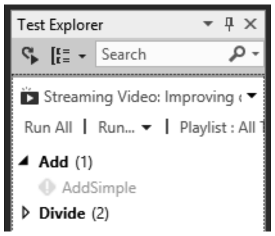
```
No Traits (3)
  DivideByZero    3 ms
  DivideSimple    3 ms
  AddSimple
```

5. However, traits are easy to add using an attribute. Add the following code above the **AddSimple** method. It indicates that this test has the **Add** trait. Note that you may add multiple traits per test, which is useful if you want to tag a test with its functional purpose, development team, performance relevance, etc.

```
[TestCategory("Add")]
```

6. Add **TestCategory** attributes to each of the divide tests, but use the category **Divide**.

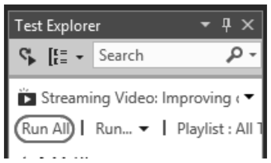
7. From the main menu, select **Build | Build Solution**.



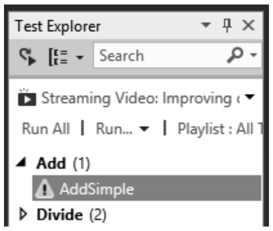
8. Another useful attribute for organizing tests is the **Ignore** attribute. This simply tells the test engine to skip this test was running. Add the following code above the **AddSimple** method.

```
[Ignore]
```

9. In **Test Explorer**, click **Run All** to build and run all tests.



10. Note that **AddSimple** has a yellow exclamation icon now, which indicates that it did not run as part of the last test pass.



There are a few more test attributes that aid in the organization and tracking of unit tests.

- **Description** allows you to specify a description for the test.
- **Owner** specifies the owner of the test.
- **HostType** allows you to specify the type of host the test can run on.
- **Priority** specifies the priority at which this test should run.
- **Timeout** specifies how long the test may run until it times out.
- **WorkItem** allows you to specify the work item IDs the test is associated with.
- **CssProjectStructure** represents the node in the team project hierarchy to which this test corresponds.
- **CssIteration** represents the project iteration to which this test corresponds.