|  |
| --- |
|  |
| Writing Unit Tests |
| Test Frameworks and Test Runners |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

# Overview

After completing this lab, you should understand the following:

* Create a test project in Visual Studio
* Use test framework attributes and APIs
* Write tests against a class in a referenced project.

# Getting Started

1. Open Visual Studio and create a new Console application in the **before** directory of this lab (File -> New Project -> Windows -> Console Application). Name the project **Employees**.
2. **Add a class** to the new project (right-click the project -> Add -> Class) with the name **Employee**.
3. Mark the new class as **public**.
4. Add string properties to the class to hold an employee’s first and last names (**FirstName** and **LastName**).
5. Add a method named **FormatName** that will return the employee’s name in the format LastName, FirstName.
6. At this point, the Employee class should look like the following.

public class Employee

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string FormatName()

{

return LastName + ", " + FirstName;

}

}

1. Right-click the Solution in the Solution Explorer Window and select **Add -> New Project**.
2. Select the **Test Project** from the Test node and give the project the name **Employees.Tests**.
3. **Delete UnitTest1.cs** from the project.
4. **Add a new class** to the project with the name **EmployeeTests**. Ensure the class is **public**.
5. Right-click the References node in the test project and select **Add Reference**.
6. Select the **Employees** project on the **Project** tab and press OK.
7. **Build** the project to ensure there are no compiler errors.

We should now be setup to write some code in a unit test.

# First Test

1. Add a **[TestClass]** attribute to the EmployeeTests class. You’ll need to bring in the Microsoft.VisualStudio.TestTools.UnitTesting namespace with a using statement.
2. Add a **public method** to the class with the name **Formats\_LastName\_FirstName**.
3. Add a **[TestMethod]** attribute to the new method.
4. Inside the method, **instantiate** a new instance of the **Employee** class. Set the name properties to match your name.
5. Next, invoke the **FormatName** method on the new object and assign the **result** to a variable.
6. Assert that the result matches your expectations using the **AreEqual** static method on the **Assert** class (Assert.AreEqual(…)).
7. At this point, your code should look like the following.

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Employees.Tests

{

[TestClass]

public class EmployeeTests

{

[TestMethod]

public void Formats\_LastName\_FirstName()

{

var employee = new Employee();

employee.FirstName = "Scott";

employee.LastName = "Allen";

var result = employee.FormatName();

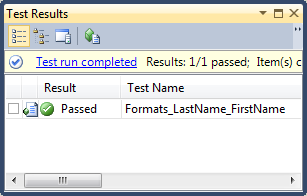
Assert.AreEqual("Allen, Scott", result);

}

}

}

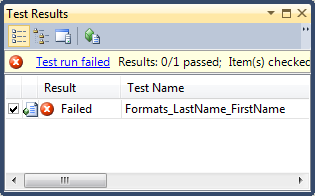
1. Run the tests (Test -> Run -> All Tests in Solution (or use Ctrl+R, A)).
2. Verify the test passes.



# Using Asserts

The AreEqual API we used in the last section is just one of many asserts available in the testing API. AreEqual is overloaded to compare many different types of objects. In this section we’ll explore a couple more.

1. **Change** your Assert.**AreEqual** call to Assert.**AreSame**.
2. Run the tests – the test should **fail**.



AreSame tests to see if two variables refer to the *same object instance*, which in this case is false.

1. Rewrite the tests to work using the **IsTrue** method (it will look like the following).

Assert.IsTrue("Allen, Scott" == result);

1. **Run** the test to ensure it is in a passing state again.

Now we will use AreEqual again, but use it to compare two arrays and see how it behaves.

1. Convert both the **result** and the **expected result** to character arrays (**ToCharArray**), and use **AreEqual** to compare the results.

[TestMethod]

public void Formats\_LastName\_FirstName()

{

var employee = new Employee();

employee.FirstName = "Scott";

employee.LastName = "Allen";

var result = employee.FormatName();

var resultAsArray = result.ToCharArray();

var expectedArray = "Allen, Scott".ToCharArray();

Assert.AreEqual(expectedArray, resultAsArray);

}

1. **Run** the test, and the test should fail.

AreEqual is designed to work with value types or types implementing IComparable (more on that interface later in the course). When faced with two reference types AreEqual compares object references (like AreSame), so it won’t work in this situation.

1. **Rewrite** the test to make the array comparisons pass. Hint: you can use IsTrue and an extension method from the System.Linq namespace to make this job easy. One possible solution is shown below.

Assert.IsTrue(expectedArray.SequenceEqual(resultAsArray));

This isn’t an approach you would typically use in a test when the simpler AreEqual would work against strings. The above code was just a demonstration to make you aware of value types versus reference types in equality checks.

# Giving Raises

1. Add a second test method to your test class named Can\_Give\_Employee\_Raise.

[TestMethod]

public void Can\_Give\_Employee\_Raise()

{

}

1. Before you add anything to the Employee class, write the test you would expect to see that allows you to give an Employee a raise, and assert that their salary changed to the expected amount.

The code you write in the test won’t compile until you change the Employee class, but the goal here is to use the test like a whiteboard and design the API you want to see. Here is one possible solution:

[TestMethod]

public void Can\_Give\_Employee\_Raise()

{

var employee = new Employee();

employee.Salary = 10.0;

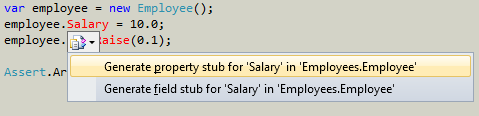
employee.GiveRaise(0.1);

Assert.AreEqual(11.0, employee.Salary);

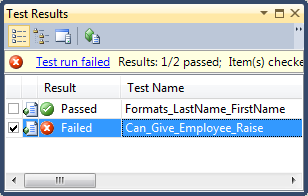
}

1. Once you are happy with the API, use Visual Studio to **generate** the Salary property and GiveRaise method on the Employee class. **Don’t change the generated code just yet!**

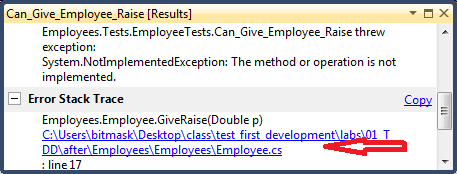
If you put the mouse cursor on a missing member, like the Salary member, you can press Ctrl+. to open a menu and select an option to generate code.



1. **Run** the tests – you should have a successful compile, but the new test should fail.



1. Right-click the failed test and select **View Test Results Details**.
2. Inspect the details and notice the test failed because of an exception in GiveRaise. Click the hyperlink in the test results that will take you directly to **GiveRaise**.



1. **Replace** the line of code in **GiveRaise** with your own working implementation.

public void GiveRaise(double p)

{

Salary = Salary \* (1+p);

}

1. **Run** all the tests to make sure they pass.
2. Go back to **review** the GiveRaise implementation, particularly if it looks like the code we just saw above. Is there anything you would change? Is the variable name “p” really a good idea? Change whatever you want about the method now, because we have a test to verify the behavior. I’d change the implementation to look like the following.

public void GiveRaise(double percentage)

{

Salary = Salary \* (1+percentage);

}

1. Change the test to give an employee a **33.33%** raise and assert the expected result.

[TestMethod]

public void Can\_Give\_Employee\_Raise()

{

var employee = new Employee();

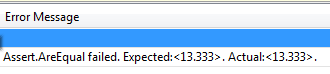
employee.Salary = 10.0;

**employee.GiveRaise(0.3333);**

**Assert.AreEqual(13.333, employee.Salary);**

}

1. **Run** the tests – and the test should fail. Do you know why?



When comparing floating point numbers, an extra parameter to AreEqual can specify the tolerance to use for imprecise floating point comparisons.

1. Use a **tolerance** of 0.0001 in your test of the employee salary.

Assert.AreEqual(13.333, employee.Salary, 0.0001);

1. **Run** the tests again to ensure everything is green.

# More Tests + Debugging

In this section we will continue to write tests to determine how an Employee object should behave. The type of questions to ask include:

* What should the properties have as initial values after construction?
* How should FormatName behave if FirstName or LastName (or both) are empty.

We won’t be looking at validation and error conditions yet – we’ll examine those after the next lecture.

1. **Write tests** to answer the above two questions. The tests might look like the following.

[TestMethod]

public void Initial\_Name\_Values\_Are\_Empty\_Strings()

{

var employee = new Employee();

Assert.AreEqual(employee.FirstName, string.Empty);

Assert.AreEqual(employee.LastName, string.Empty);

}

[TestMethod]

public void Format\_Empty\_Name\_Returns\_Empty\_String()

{

var employee = new Employee();

var result = employee.FormatName();

Assert.AreEqual(result, string.Empty);

}

1. Make the Initial\_Name\_Values\_Are\_Empty test pass by adding a **constructor** to the Employee class.

public Employee()

{

FirstName = "";

LastName = "";

}

1. **Run** the tests to ensure Initial\_Name\_Values\_Are\_Empty passes.
2. Change the logic in **FormatName** to the following (yes, there are two bugs).

public string FormatName()

{

if(string.IsNullOrEmpty(LastName) || string.IsNullOrEmpty(FirstName))

{

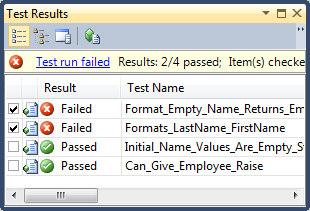
return LastName + "," + FirstName;

}

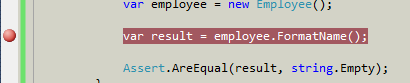
return string.Empty;

}

1. **Run** the tests, and you should now have two failing tests.



1. Put a **breakpoint** in **one** of the two failing tests on the call to FormatName (you can put the cursor on the line and press F9).



1. **Run** the tests with a **debugger** attached.

You can run with the debugger using the Test -> Debug -> All Tests In Solution.

1. When you hit the breakpoint, step (**F11**) through the code and make sure you see the bug.
2. **Stop debugging** (Shift+F5). Notice the test runner will show the tests as aborted. **Fix** the logic errors and continue running tests until all the tests pass again.

# The StringCalculator Kata

In this section we’ll start a new test-driven development effort known as a Kata (to read more about Code Katas, see <http://codekata.pragprog.com/>). This section is open ended in the sense your reward will be commensurate with the effort you apply! There are fewer step by step instructions. Feel free to ask the instructors about any questions you have.

1. Add a new class to the **Employees** project named **StringCalculator**.
2. Give the StringCalculator an Add method that takes a string parameter and returns an integer.
3. Add a new class to the **Employees.Tests** project named **StringCalculatorTests**.

The goal is for the StringCalculator to take 0, 1, or 2 numbers in the string parameter. The class should return the sum of the numbers (and return 0 for an empty string or null value).

Start with the simplest possible test case – pass an empty string and assert the calculator returns a 0. Build only the simplest possible implementation to satisfy the test. Add complexity in small amounts. The second test should “sum” a single number (like “3”), then eventually move to summing two numbers (like “1,4”). Finally, see if your class can correctly sum a large number of arguments (like “1,2,5,6,10”).

# Conclusion

Congratulations! You’ve started a journey towards test-driven development.