Course Outline

- Writing your First NUnit Test
- Understanding NUnit Tests
- Asserting on Different Types of Results
- Controlling Test Execution
- Creating Data Driven Test and Reducing Test Code Duplicaton

NUnit Packages

- NUnit: give access to the NUnit attributes and asserts that is need to write the tests.
- NUnit3TestAdapter
- Microsoft.NET.Test.SDK

NUnit Test Features/Attributes

- `[TestFixture]`: used at the class level to mark the class that contains tests
- `[Test]`: applied to a method to mark the method as a test.
- `[Category]`: Organize tests into categories
- `[TestCase]`: Data driven test cases
- `[Values]`: Data driven test parameters
- `[Sequential]`: How to combine test data
- `[SetUp]`: Run code before each test
- `[OneTimeSetup]`: Run code before first test in class

Understanding NUnit Tests Module 2 Overview

- Why write automated tests?
- Regression example
- Understanding the NUnit test framework
    - NUnit Library
    - Test runner
- Recognizing different testing scenarios
- The logical phases of a test
- Add a second test
- Qualities of good test

Why write automated tests?

- Help to find defects and regressions
- Happier:
    - End-Users experience
    - Business/management
    - Developers
- Reduce cost of software development
- Increase long-term speed of development

- Quick to execute
- Free to run as often as required

Automated tests give greater confidence that the software is working as it should.

Understanding the NUnit Test Framework

NUnit Library	Test Runner
Attributes	Recognizes attributes
Assertions	Execute test methods
Extensibility/customization	Report test results
	Test Explorer
	dotnet test command line

NUnit Assertions Module 3 Overview

```
// Constraint Model of assertions (newer)
Assert.That(sut.Years, Is.EqualTo(1));
Assert.That(test result, constraint instance);

// Classic Model of assertions (older)
Assert.AreEqual(1, sut.Years);
Assert.NotNull(sut.Years);
Assert.Xyz(...);
```

Recognizing different Testing scenarios

- Business Logic: follow the money or the value that the application creates. It's essential that this core business logic works correctly for the application to add value to the business.
- Exercise all code branches: exercise or execute all code branches in the method or class that is being tested. This is often described as code coverage.
- Bad-data/Input: check for bad data or bad input into the system. In addition to checking how the system behaves with expected values, such as 0, also check what happens when there is an unexpected input, such as negative numbers, null values, or empty strings.

The Logical Arrange, Act, Assert Test Phases

- `Arrange`: set up test object(s), initialize test data, e.t.c.
- `Act`: call method, set property, e.t.c.
- `Assert`: compare returned value/end state with expected.

Qualities of Good Tests

- Fast
- Repeatable

- Isolated
- Trustworthy
- Valuable

Asserting on Different Types of Results Module 4 Overview

- What are Asserts?
- The NUnit Constraint Model of assertions
- How many asserts per test?
- Asserting on equality
- Asserting on reference equality
- Adding custom failure messages
- Asserting on floating point values
- Asserting on collection contents
- Asserting that exceptions are thrown
- Other assertion examples

Asserts

Evaluate and verify the outcome of a test based on a returned result, final object state, or the occurrence of events observed during execution. An assert should either `pass` or `fail`.

- If all asserts pass, the test passes.
- If any assert fails, the test fails.

A single test usually focuses on testing a single "behaviour'.
Multiple asserts are usually OK if all the asserts are related to testing this single behaviour.

Assert Methods

```
// Assertion Methods
Assert.That(actual, Is.EqualTo(excpected));
Assert.That(actual, Is.Not.EqualTo(excpected));
Assert.That(actual, Is.SameAs(excpected));
Assert.That(actual, Is.Not.SameAs(excpected));

//Floating point values
Assert.That(actual, Is.EqualTo(excpected).Within(tolerance)); // tolerance can be
a decimal e.g 0.004
Assert.That(actual, Is.EqualTo(excpected).Within(percentValue).Percent); //
percentValue is a percentage e.g 10percent e.t.c.

//List
Assert.That(actual, Has.Excatly(excpectedNoOfItems).Items);
Assert.That(actual, Is.Unique);
Assert.That(actual, Does.Contain(excpected));

//String
Assert.That(actual, Is.Empty);
Assert.That(actual, Is.Not.Empty);
```

```
Assert.That(actual, Is.EqualTo(excpected).IgnoreCase);
Assert.That(actual, Does.StartWith(excpected));
Assert.That(actual, Does.EndWith(excpected));
Assert.That(actual, Does.Contain(excpected));
Assert.That(actual, Does.Not.Contain(excpected));
Assert.That(actual, Does.StartWith(excpected).And.EndsWith(expected));
Assert.That(actual, Does.StartWith(excpected).Or.EndsWith(expected));

// Bool
Assert.That(actual, Is.True);
Assert.That(actual, Is.False);
Assert.That(actual == false);
Assert.That(actual == true);
Assert.That(actual, Is.Not.True);
Assert.That(actual, Is.Not.Fasle);

// Within Ranges
Assert.That(actual, Is.GreaterThan(excpected));
Assert.That(actual, Is.GreaterThanOrEqualTo(excpected));
Assert.That(actual, Is.LessThan(excpected));
Assert.That(actual, Is.LessThanOrEqualTo(excpected));
Assert.That(actual, Is.Range(excpectedRange));

// DateTime
Assert.That(actual, Is.EqualTo(excpected));
Assert.That(actual, Is.EqualTo(excpected).Within(NoOfDays).Days);

//Link http://bit.ly/nunit3asserts
```

Check more on NUnit 3 Asserts

Controlling Test Execution Module 5 Overview

- Preventing tests from running
- Organizing tests into categories
    - Running categories in Test Explorer
    - Running categories at the command line
- An overview of the test execution lifecycle
- Run setup code before each test executes
- Run clean up code after each test executes
- Run setup code before the first test in a test class executes
- Run clean up code afte the last test in a test class executes.

Preventing tests from running

```
// To prevent a test from running
// add the below line of code add the top of the method or class
[Ignore("Message")]
```

Organizing tests into categories

```
// To prevent a test from running
// add the below line of code add the top of the method or class
[Category("Message")]
```

An overview of the test execution lifecycle

Test Class	One instance per test run
`ctor`	
Test 1	
Test 2	
`Dispose`	If IDisposable is implemented

Test Class	One instance per test run
`ctor`	
One-time Setup	Initialize data/environment/files, e.t.c
`Setup`	Reset data, create *sut*, e.t.c.
Test 1	
`Tear down`	Clean up data, dispose of sut, e.t.c.
`Setup`	
Test 2	
`Tear down`	
One-time tear down	Clean data/environment/files e.t.c.
`Dispose`	If IDisposable is implemented

Run setup code before each test executes

```
// Ren setup code before all tests
// add the below line of code add the top of the method or class
[Setup]
```

```
// Ren setup code before each test
// add the below line of code add the top of the method or class
[OneTimeSetup]
```

```
[Setup]

[TearDown]
```

Creating Data Driven Test and Reducing Test Code Duplicaton Module 6 Overview

- **Creating data driven tests**
  - Providing method level test data
  - sharing test data across multiple tests
  - Reading external test data
  - Generating combinatorial tests
  - Generating test data ranges
- **Customizing NUnit**
  - Creating custom attributes
  - Creating custom constraints