10

AUTOMATED TESTING

OVERVIEW

In this chapter, you will bring together core parts of C# to see how code can be verified for correctness in a way that is simple to run and easy to grasp and maintain. You will learn about the different forms of testing before taking an in-depth look at unit testing using **`NUnit`**, the most popular third-party testing library for C#. By the end of this chapter, you will be able to create effective unit tests in C#.

INTRODUCTION

Getting software to work correctly is difficult enough but ensuring that the software continues to work for days, months, or even years after it was first written can be particularly tough. There may be dozens of potential execution paths that need to be considered.

For example, you could have a UI app where different users have different levels of access; there may be a **power user** who has access to all features and some occasional users with access to far fewer features. Typically, for such an app, there are several lines of code with various interactions between multiple classes. Now, consider that you need to reconfigure this code and restrict access levels for the users who haven't used the app in more than a year. How can you be confident that making changes to such a vast and complex code will not introduce bugs, no matter how subtle the change is? This is where automated testing can be useful.

The point of automated testing is to provide a certain level of confidence that the code is correct and will remain correct even after being altered. Taking a step back and creating unit tests at the early stages of coding often results in an improved design, and a code that is easier to grasp and maintain.

In this chapter, you will see how unit testing can be done in C# to provide the correct code. You'll start by looking at the types of testing and briefly cover the commonly used unit test frameworks, before building actual tests.

TYPES OF AUTOMATED TESTING

The two main types of automated testing often followed in software development are unit testing and integration testing. Let's take a look at each of these briefly.

UNIT TESTING

A **unit test** is a code that invokes another piece of code and **verifies** a result. The result could be the value of a single property or multiple properties of a class instance. The following example updates similar address properties for a customer instance, but it contains one tiny bug:

Program.cs

```
public class Address
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string Line3 { get; set; }
    public string Line4 { get; set; }
    public string Line5 { get; set; }
}

public class CustomerOrder
{
    private readonly Address _deliveryAddress = new Address();

    public void UpdateDeliveryAddress(string line1, string line2, string
line3,
        string line4, string line5)
```

You can find the complete code here: https://packt.link/okhuq.

The bug should be quite easy to spot as there are only a few lines of code:

```
    {
        _deliveryAddress.Line1 = line1;
        _deliveryAddress.Line2 = line2;
        _deliveryAddress.Line3 = line3;
        _deliveryAddress.Line3 = line4;
        _deliveryAddress.Line5 = line5;
    }
}
```

When **`UpdateDeliveryAddress`** is called, it should update **`Line1`**, **`Line2`**, **`Line3`**, **`Line4`**, and **`Line5`** for the **`CustomerOrder`** class, **`_deliveryAddress`** property, but a mistake has been added to the second from last line:

```
_deliveryAddress.Line3 = line4.
```

This line incorrectly updates **`_deliveryAddress.Line3`** to be **`line4`**. Instead of this, it should have set:

```
_deliveryAddress.Line4 = line4
```

Imagine if there were hundreds of lines of code around this, it would have been harder to spot. A carefully crafted unit test could be used to detect such an error. Unit tests can also be used to verify how many times a particular interface method was **expected** to be invoked by a caller and how times it was **actually** invoked.

In the next example, you have an **`Employee`** class with an **`AnnualSalary`** property:

```
public class Employee
{
    public double AnnualSalary { get; set; }
}
```

Once per year, an instance of **`PayrollService.cs`** is created and is used to increase an employee's **`AnnualSalary`** using a class that implements an **`ISalaryCalculator`** interface. Here, the **`SalaryCalculator`** class simply adds the specified amount to the employee's **`AnnualSalary`**:

```
public interface ISalaryCalculator
{
    public void CalculateSalary(Employee employee, double amount);
}

public class SalaryCalculator : ISalaryCalculator
{
    public void CalculateSalary(Employee employee, double amount)
    {
        employee.AnnualSalary += amount;
    }
}
```

Depending on the various rules, there may be different types of **`ISalaryCalculator`** implementations. Consider the following snippet:

`Program.cs`

```
public class PayrollService
{
    private readonly Employee _employee = new() {AnnualSalary = 1000};

    public void CalculateSalary(ISalaryCalculator calculator)
    {
       calculator.CalculateSalary(_employee, 200);
       calculator.CalculateSalary(_employee, 200);
    }

    public Employee Employee => _employee;
}

public class Program
{
```

`You can find the complete code here:` https://packt.link/PbfJA.

Running the program produces the following output:

```
Initial Salary: 1000
New Salary: 1400
```

In this case, when **`PayrollService.CalculateSalary`** runs, it actually calls the **`calculator.CalculateSalary`** method twice as **`calculator.CalculateSalary(_employee, 200)`**.

It's probably safe to assume that **`CalculateSalary`** should only be called once by **`PayrollService`**. However, it is called twice. How can this be verified? With a unit test, the interaction between **`CalculateSalary`** and a class that implements the **`ISalaryCalculator`** interface can be verified, without even knowing how the **`CalculateSalary`** method is implemented.

> **NOTE**
>
> You can find the code used for this example at https://packt.link/cWLT0.

The verb **assert** is commonly used to describe such verifications and, as such, many popular unit testing frameworks contain classes named **`Assert`**, as you will discover shortly. The word unit was often used to refer to a single method that was being tested but has since been revised to refer to a **unit of work**. This unit of work can still be a single method and cover interactions with multiple classes or interfaces.

Unit testing is often the first line of defense against bugs. These tests should be self-checking (they should not need any human interaction to confirm success or failure), easy to run, and should produce the same results if they are given the same set of inputs. No matter what your preferred IDE is, they should be easy to run with little extra work required. A well-designed unit test should not take more time to write than the code it is testing.

In the following example snippet, **`GetMonthNameTest`** is a basic unit test method. A **`MonthFormatter`** class (code omitted here) contains a method that converts a numeric month into its English name. For this test, you will pass the number **`4`** and expect the resulting **`month`** to be the string, **`April`**.

Using the **`Assert`** and **`Is`** classes, the test verifies that the value returned is **`April`** (you will learn more about these classes next). When this test is run, it will throw an exception if the value of the **`month`** is not as expected:

```
public void GetMonthNameTest()
{
  var month = MonthFormatter.GetMonthName(4);

  Assert.That(month, Is.EqualTo("April");
}
```

Unit tests can be thought of as foundation blocks that verify the correctness of a single class or a small number of cooperating classes to achieve a result. The term **correctness** is often used in testing and simply refers to how you verify an expected outcome with the actual outcome observed. There is also a second type of testing called integration testing. Proceed to know more about this type of testing.

INTEGRATION TESTING

For efficiency, unit tests should be memory-based that is while running they should only allocate memory for variables on a host machine and not depend on any real resources, such as databases or files, to run. In contrast, **integration testing** is a way to test integrated units of work that require real dependencies to run.

For example, you may have a service that needs to create a customer record in a database table. An integration test can verify this behavior by defining the customer's details, invoking the service that creates the record in a database, and finally connecting to the destination database and extracting the record from a table. There are potentially many costly steps to cover in integration tests. In the customer record creation scenario, connecting to an external database could be costly in terms of time.

Integration testing would not be covered in detail in this chapter, as database usage is an entire topic of its own. Instead, you will focus on using C# to write effective unit tests. The following example shows a unit test method, **`AssertNumberExamples`**:

```
using System;
using NUnit.Framework;
[TestFixture]
public class ExamplesTests
{
[Test]
      public void AssertNumberExamples()
```

```
        {
                Assert.That(2, Is.LessThan(3));
              Assert.That(5, Is.InRange(4, 6));
        }
}
```

The syntax of the preceding code will be covered in detail next.

GETTING STARTED WITH UNIT TESTS

Typically, you will work with production code that performs units of work and will maintain a separate suite of tests in a different project; there is no logical reason for shipping production code that is mixed with a set of unit tests.

There are numerous unit testing frameworks available for .NET that make it easy to create and run unit tests. You will briefly cover the two most popular frameworks, **`NUnit`** and **`MSTest`**, to see how tests can be defined for each one, before exploring **`NUnit`** in depth.

> **NOTE**
>
> Unit tests are often created and maintained at the same time as changes are made to the production code. This practice is referred to as **Test-Driven Development** (**TDD**). In this chapter, you will not follow a strict TDD approach as the focus is on testing in C# itself, rather than a detailed study of practices.

Before introducing the frameworks though, it is worth taking a quick look at attributes, as they are used extensively by each framework. An **attribute** is a standard C# class that can be used to provide extra information (metadata) about your code. This can be queried at runtime using classes in the **`System.Reflection`** namespace. This information can be in the form of strings, enums, integers, or in fact any system or user constant that you want to declare.

You can create your own custom attributes by creating a class derived from the **`Attribute`** type. In the following snippet, you are creating an attribute named **`MyValidationAttribute`**:

```
public class MyValidationAttribute : Attribute
```

Note that attributes can also be assigned to assemblies, classes, methods, and many other parts of C# code using the syntax:

```
[AttributeName]
```

When you use an attribute, you wrap square brackets around the attribute name. In the case of unit testing, attributes are used to mark classes and methods as unit-test related, being read at runtime by a **test runner**, using the attribute-based syntax.

The **`TestFixture`** attribute of **`NUnit`** is one such example. In the next snippet, it has been assigned to a class called **`NunitExamples`**. This tells the test runner that this class, **`NunitExamples`**, has test methods that it can search for and run (you will look at this in more detail later):

```
[TestFixture]
public class NunitExamples
{}
```

Each framework provides its own set of attributes, which you will look at in more detail.

NUNIT

`NUnit` has been the de facto standard for C# unit testing since 2006. It has undergone numerous improvements as the C# language itself has evolved. **`NUnit`** quickly became the preferred standard due to the initial limitations of the **`MSTest`** implementation. These limitations made it difficult to pass various parameters to test methods. Also, **`MSTest`** tended to run slower than **`NUnit`** and it was not initially open source, so it was difficult to extend.

NUnit v3 runs on the following platforms:

- .NET Framework 3.5+
- .NET Standard 1.4+
- .NET Core

Here is a brief overview of an **`NUnit`**-based test class. Its **`TestFixture`** attribute is used to decorate a class called **`NUnitExamples`**. At runtime, a test runner will look for all classes that have this attribute assigned to them in this way:

```
using System;
using NUnit.Framework;

namespace Chapter10.Examples
```

```
{
    [TestFixture]
    public class NUnitExamples
    {
```

The test runner looks for any public methods that are decorated with the **`Test`** attribute and automatically runs them for you. Typically, you create instances of your own production classes and invoke methods, which you then verify for correctness, by calling various methods found in **`NUnit`**'s static **`Assert`** class.

In the next snippet, **`Assert.That`** takes two arguments, an expected value, and an actual value. You are simply asking **`Assert.That`** to confirm that the expected value of **`1`** is equal to the actual value of **`1`**. Clearly, this assertion will pass, but as you will see later these assertions can be built up to verify any of the properties of a class instance after a method has run:

```
        [Test]
        public void One_EqualTo_One()
        {
            Assert.That(1, Is.EqualTo(1));
        }
```

In addition to specifying the **`Test`** attribute for a test method, you can also use the **`TestCase`** attribute to define any number of parameters that you want passed into a test method by the test runner:

```
        [TestCase(1, 1)]
        [TestCase(2, 2)]
        public void EqualTo_TestCases(int expected, int actual)
        {
            Assert.That(expected, Is.EqualTo(actual));
        }
    }
}
```

The test runner will run this test method twice, first passing in arguments **`1, 2`** and then **`2, 2`**. This test asserts that the **`expected`** and **`actual`** values are equal using the **`Assert`** and **`Is`** classes. This test will always pass, as the two sets of arguments being passed are valid. With this code, you are simply showing how to call the **`Assert.That`** methods. In your production tests, you would create an instance of a class, interact with it, and then verify its final state using various **`Assert`** methods.

You will cover the **`Assert`** and **`Is`** classes in more detail shortly. Let's take a brief look at Microsoft's **`MSTest`**; you don't really need to know about it, but it is handy to see how similar **`MSTest`** is.

MSTEST

Microsoft's **`MSTest`** has been available since 2015 and has been a regular choice for many, although it is not as popular as **`NUnit`**.

`MSTest` v2 runs on the following platforms:

- .NET Framework 3.5+
- .NET Standard 1.4+
- .NET Core

`MSTest`-based test classes are structurally similar to those for **`NUnit`**, though they use different attribute names. Here, you will see the main differences between **`NUnit`** and **`MSTest`**. You may find this helpful if you work on a project that already has **`MSTest`**-based unit tests.

`MSTest` uses the **`TestClass`** attribute to declare a test class (**`NUnit`** uses **`TestFixture`**). The test runner will look for any classes decorated with this attribute, like in the following snippet:

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

namespace Chapter10.Examples
{
    [TestClass]
    public class MSTestExamples
    {
```

The **`TestMethod`** attribute is used to mark a test method (**`Test`** for **`NUnit`**) and **`Assert.AreEqual`** is used to compare actual and expected values. Again, the test in the next snippet will pass as you are showing how to call **`Assert.AreEqual`** with two values that are equal. As with **`NUnit`**, you would create instances of your production code and assert that the values are correct:

```
        [TestMethod]
        public void One_EqualTo_One()
        {
            Assert.AreEqual(1, 1);
        }
```

MSTest uses the **DataRow** attribute to mark a test method that allows arguments to be passed to a test method. The same values are used in this example as you saw for **NUnit**: comparing **1** with **1** and then **2** with **2**:

```
        [DataRow(1, 1)]
        [DataRow(2, 2)]
        public void EqualTo_TestCases(int expected, int actual)
        {
            Assert.AreEqual(expected, actual);
        }
    }
}
```

As you can see, **MSTest** and **NUnit** tests are similar in style, with the attributes being named differently in each testing framework. The actual test assertions use a similar **Assert.AreEqual** call to check for equality.

You have seen a brief overview of the two main unit-testing frameworks. The following section will take a more focused look at **NUnit**. As stated earlier, **NUnit** remains the most popular framework for testing C# applications largely due to its earlier adoption in C# development.

UNIT TESTING WITH NUNIT

Previously, you saw a small example of an **NUnit**-based test class. Using various **NUnit** attributes, you can control which methods, if any, are called by the **NUnit** test runner before or after any test methods are called.

The following example shows an overview of these attributes, assigned to descriptively named methods:

```
Using System;
using NUnit.Framework;

namespace Chapter10.Examples
{
    [TestFixture]
    public class NunitExamples
    {
        [OneTimeSetUp]
        public void RunsOnceBeforeAnyTest()
        {}
```

```
        [OneTimeTearDown]
        public void RunsOnceAfterAllTest()
        {}

        [SetUp]
        public void RunsBeforeEveryTestMethod()
        {}

        [Test]
        public void One_EqualTo_One()
        {
            Assert.That(1, Is.EqualTo(1));
        }

        [TestCase(1, 1)]
        [TestCase(2, 2)]
        public void EqualTo_TestCases(int expected, int actual)
        {
            Assert.That(expected, Is.EqualTo(actual));
        }
    }
}
```

The **`NUnit.Framework`** namespace contains the following extensively used attributes:

- The **`TestFixture`** attribute is assigned to a class definition. The **`NUnit`** test runner needs this assigned to find the classes in a project that contains unit tests.
- The **`OneTimeSetUp`** attribute can optionally be assigned to a method and gets called **once** before any tests are run in the class. This can be used to clear any objects or states needed by the tests. For example, you may have various tests that need to delete a selection of files from the system prior to running a test (maybe the production code creates files in a certain system folder.) Using the **`OneTimeSetUp`** attribute, you can declare a method to be called before any tests are run.
- The **`OneTimeTearDown`** attribute is the opposite to **`OneTimeSetUp`** in that it is also optional but gets called **after** the last test has run. It can be useful for clearing up any post-test resources.

- The **`SetUp`** attribute is used to optionally define a method that is called prior to each test method in the class. This can be used to clear any shared state or to initialize an instance to be tested. For example, you may have a suite of test methods that test a calculator. To create your **`Calculator`** class, you could define a **`SetUp`** method that simply creates a new class-level **`Calculator`** instance each time the method is called by the test runner. This saves you from having to add a **`Calculator`** constructor method to the beginning of every test method.
- The **`Test`** attribute is used to mark a test method. The test runner will run this method; this is where you use the **`Assert`** class to verify test values.
- The **`TestCase`** attribute can be used to pass any number of arguments to a test method. The arguments must be compatible with the method signature they precede.

CREATING UNIT TEST METHODS

A unit test method is usually written with three distinct sections, commonly referred to as the **Arrange-Act-Assert** (**AAA**) pattern:

- Arrange: This is where objects to be tested are created and configured as needed.
- Act: Perform an action on the object to be tested (by calling a method).
- Assert: Verify that something is in an expected state, for example, that a property is of an expected value. This is done using methods found in the static **`Assert`** class.

The reason you should clearly define three separate sections is to make the test as readable and maintainable as possible. By separating each section, you can show any dependencies required to call your code. You can also decide how your code is *meant* to be called and what you are expecting. If done well, unit tests can serve as documentation for your code.

Let's take a further look at **`NUnit`**'s static **`Assert`** class. This class contains numerous methods that can be used to verify the expected state after an action has run. Typically, these assertions will tend to include value equality checks, such as checking that the number or string properties match. **`NUnit`** will simply throw an **exception** if your actual and expected values do not match. These exceptions are caught by the project's unit test adapter and are typically displayed as a summary message after all tests have run.

Most methods found in the **`Assert`** class take two arguments, an actual parameter, and a second constraint parameter. It is up to you to pass the actual value and then pick the appropriate constraint from **`NUnit`**'s various helper classes, depending on what you are trying to assert for correctness. Each of these constraints contains all the code required to verify the actual and expected outcomes for you.

`Assert.That()` is one of the most often used methods and one that you'll be using frequently in the upcoming examples and exercises. **`Is.EqualTo()`** is a commonly used constraint found in the **`Is`** helper class. This returns a constraint that tests two items for equality, which makes it an ideal constraint to pass to **`Assert.That`**.

The next example shows a test method that contains each section of the AAA pattern, prefixed with a comment for clarity. For readability, the code to define the **`Order`** class has been included here, but normally you would declare that in a different project; it is rare to mix production and test code together in this manner:

```
using NUnit.Framework;
class Order
{
  public double Total { get; set; }
  public void ApplyDiscount(double amount)
  {}
}
```

First, define the **`OrderTests`** test class with a test method, **`ApplyDiscount_FiftyPercent_UpdatesTotal`**. Then create an instance of the **`Order`** class with an initial **`Total`** value of **`20`**:

```
[TestFixture]
public class OrderTests
{
    [Test]
    public void ApplyDiscount_FiftyPercent_UpdatesTotal()
    {
        // Arrange
        var order = new Order {Total = 20};
```

Next, the order's **`ApplyDiscount`** method is called, passing in **`0.5`** (as you want to apply a 50% discount). The order's **`TotalValue`** should then be **`10`**:

```
        // Act
        order.ApplyDiscount(0.5);
```

```
        // Assert
        Assert.That(order.Total, Is.EqualTo(10));
    }
}
```

If you were to run this unit test, it would fail as **`ApplyDiscount`** contains no code at the moment. Here fail means that an exception would be thrown because you are expecting the **`Total`** property to be **`10`**, but it is still at the initial value of **`20`**. You will look at how to run test methods using the **`dotnet test`** command shortly.

> **NOTE**
>
> You can find the code used for this example at https://packt.link/lTV49.

This section covered **`Assert.That()`**, which is one of the most used methods and the commonly used constraint found in the **`Is`** helper class. With this knowledge, you can now learn about naming the unit test methods, as demonstrated/ discussed in the following section.

NAMING UNIT TEST METHODS

You have briefly looked at the three main sections found in a unit test method, but it is also important to consider the name that you use for your test methods. Rather than a cryptic method name, such as **`TestMath`**, you should follow this three-part convention:

- Method or feature under test
- Scenario under test
- Expected behavior

This naming convention clearly states what the test method is trying to confirm; for example, notice how you named the unit test method in the preceding discount application example. Instead of simply writing **`ApplyDiscount`**, it was named as **`ApplyDiscount_FiftyPercent_UpdatesTotal`**. Other typical examples could look like this:

- **`Add_DuplicateUniqueId_ThrowsException()`**
- **`Remove_NewOrder_UpdatesBasketTotal()`**
- **`Calculate_NegativeDistance_ReturnsNegativeHeight`**

Each of the three parts is separated by an underscore for readability. Even without any example code, you can infer what each test might be trying to verify.

> **NOTE**
>
> It is better to place unit tests in their own project so that you do not need to worry about packaging production code with potentially large binaries generated from thousands of lines of unit test code. You may also not want to include dependencies, such as **`NUnit`**, which are unnecessary for the production code to run.

This concludes the theoretical portion of this topic. In the following section, you will put this into practice with an exercise.

EXERCISE 10.01: CREATING A UNIT TEST WITH SIMPLE VALUE ASSERTIONS

In this exercise, you will create a new unit test project using the CLI and add a basic test that follows some of your conventions. This will be a simple value-based assertion using a test that will pass if **`10`** times **`10`** is equal to **`100`**.

Such a test may look overly simplistic and even unnecessary; however, the purpose of this exercise is simply to show how assertions can initially be defined. You might use a similar approach in practice when you start testing your own production code. Perform the following steps to do so:

1. Create a folder called **`Chapter10`** and change to that folder.
2. Run the following CLI command to create a new **`nunit`** test project called **`Chapter10.Tests`**:

```
dotnet new nunit -o Chapter10.Tests
```

   Running the command produces an output similar to this:

```
The template "NUnit 3 Test Project" was created successfully.
Processing post-creation actions...
Running 'dotnet restore' on Chapter10.Tests\Chapter10.Tests.csproj...
  Determining projects to restore...
  Restored source\Chapter10\Chapter10.Tests\Chapter10.Tests.csproj
```

In VS Code, you will see a new **`Chapter10.Test`** folder created in the **`Chapter10`** folder. This test folder contains files named **`Chapter10.Tests.csproj`** and **`UnitTest1.cs`**.

3. Open **`Chapter10.Tests.csproj`** file. Ensure the **`TargetFramework`** and **`LangVersion`** values are as follows (you are targeting C#10 language features):

```
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <LangVersion>10.0</LangVersion>
  </PropertyGroup>
```

You will also see the three package references in **`Chapter10.Tests.csproj`**:

```
  <ItemGroup>
    <PackageReference Include="NUnit" Version="3.13.0" />
    <PackageReference Include="NUnit3TestAdapter" Version="4.1.0" />
    <PackageReference Include="Microsoft.NET.Test.Sdk"
Version="17.2.0"/>
  </ItemGroup>
```

4. Rename **`UnitTest1.cs`** to **`Exercise01Tests.cs`**.

5. Open **`Exercise01Tests.cs`** and remove all the default code, as you will be starting from scratch.

6. As these are **`NUnit`**-based tests, you need to include the **`NUnit.Framework`** namespace with a **`using`** statement. Your test project is called **`Chapter10.Tests`**, so that will be your default namespace:

```
using NUnit.Framework;
namespace Chapter10.Tests;
```

7. As mentioned earlier, **`NUnit`** requires that each class containing unit tests has the **`TestFixture`** attribute declared at the class level using the standard **`[attributename]`** syntax. The **`NUnit`** test runner looks for classes that are marked with this attribute and without this, the tests will not be run:

```
[TestFixture]
public class Exercise01Tests
{
```

8. Your test class can now be given a test method. Recall that each test method must be marked with the **`Test`** or **`TestCase`** attributes since this is how the **`NUnit`** runner determines which methods are the actual test methods to be called. Add a new test method called **`AssertThat_TwoNumbers_ReturnProduct`**. Note that this follows the three-part naming convention discussed earlier:

```
        [Test]
        public void AssertThat_TwoNumbers_ReturnProduct()
```

9. You now need to assert that the value of **`10 * 10`** is equal to **`100`** using the **`Assert.That`** method, as follows:

```
        {
            Assert.That(10 * 10, Is.EqualTo(100));
        }
    }
```

10. Run this unit test to confirm that it does in fact pass. Change to the **`Chapter10.Tests`** folder and run the following command:

```
dotnet test
```

    This command runs your unit test and produces the following output:

```
source\Chapter10\Chapter10.Tests>dotnet test
Test run for source\Chapter10\Chapter10.Tests\bin\Debug\net6.0\
Chapter10.Tests.dll (.NETCoreApp,Version=v6.0)
Microsoft (R) Test Execution Command Line Tool Version 17.2.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:    56, Skipped:     0, Total:
56, Duration:  - Chapter10.Tests.dll (net6.0)
```

You will see that one test was found (see the highlighted code line) and that test passed (**Passed**). The output also provides additional information, such as how long the tests took to run.

> **NOTE**
>
> You can find the code used for this exercise at https://packt.link/10mEo.

11. Alter the test to produce a failure by changing the **Is.EqualTo** constraint from **100** to **300**, as follows:

```
Assert.That(10 * 10, Is.EqualTo(300))
```

12. Running **dotnet test** again produces the following output:

```
Test run for source\Chapter10\Chapter10.Tests\bin\Debug\net6.0\
Chapter10.Tests.dll (.NETCoreApp,Version=v6.0)
Microsoft (R) Test Execution Command Line Tool Version 17.2.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
  Failed AssertThat_TwoNumbers_ReturnProduct [40 ms]
  Error Message:
     Expected: 300
  But was:  100

  Stack Trace:
     at Chapter10.Tests.Exercise01Tests.AssertThat_TwoNumbers_
ReturnProduct() in  \source\Chapter10\Chapter10.Tests\
Exercise01Tests.cs:line 12

Failed!  - Failed:     1, Passed:    55, Skipped:     0, Total:
56, Duration:  - Chapter10.Tests.dll (net6.0)
```

As expected, there is a failure message. **AssertThat_TwoNumbers_ ReturnProduct** is reported as a failure along with a reason: the **Expected** value was **300** but the **actual** result was **100**.

13. If you run the unit test directly from VS Code, then you have various options:

    - Click **`Run Test`** in the Code Window:

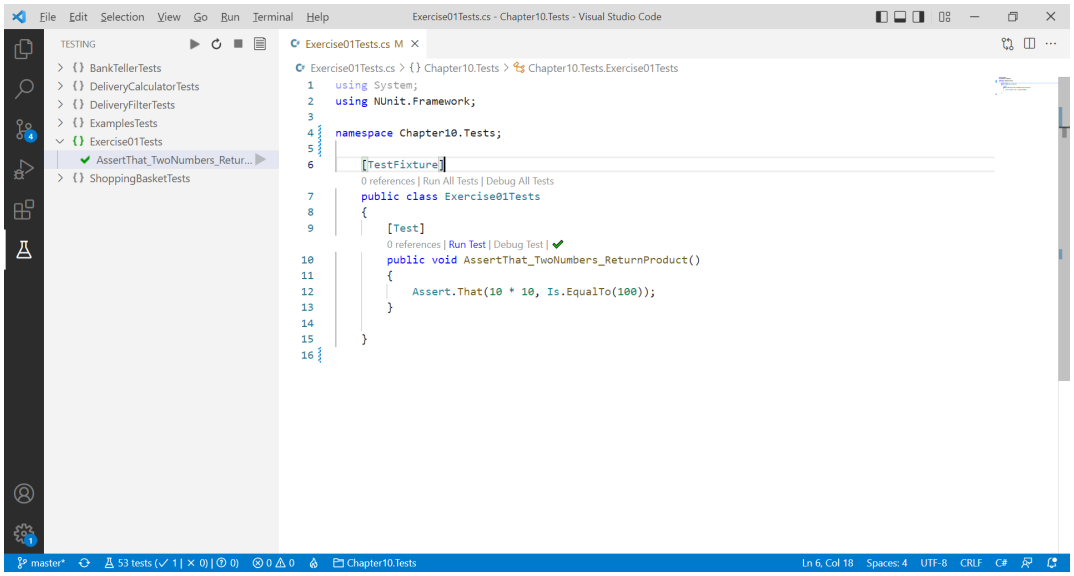

Figure 10.1: VS Code's Run Test option

    - Right-click anywhere on the **`Exercise01Tests.cs`** window and choose the **`Run Tests in Context`** menu option:

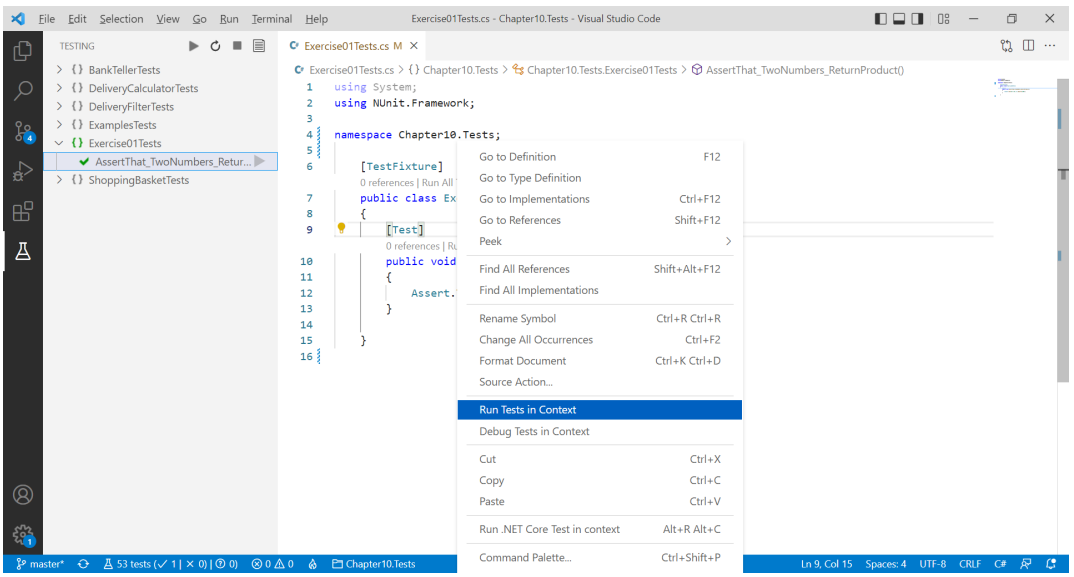

Figure 10.2: The Run Tests in Context option in VS Code

- A third option is to pick **`.NET: Run Tests in Context`** from the Command Palette option under the View menu:

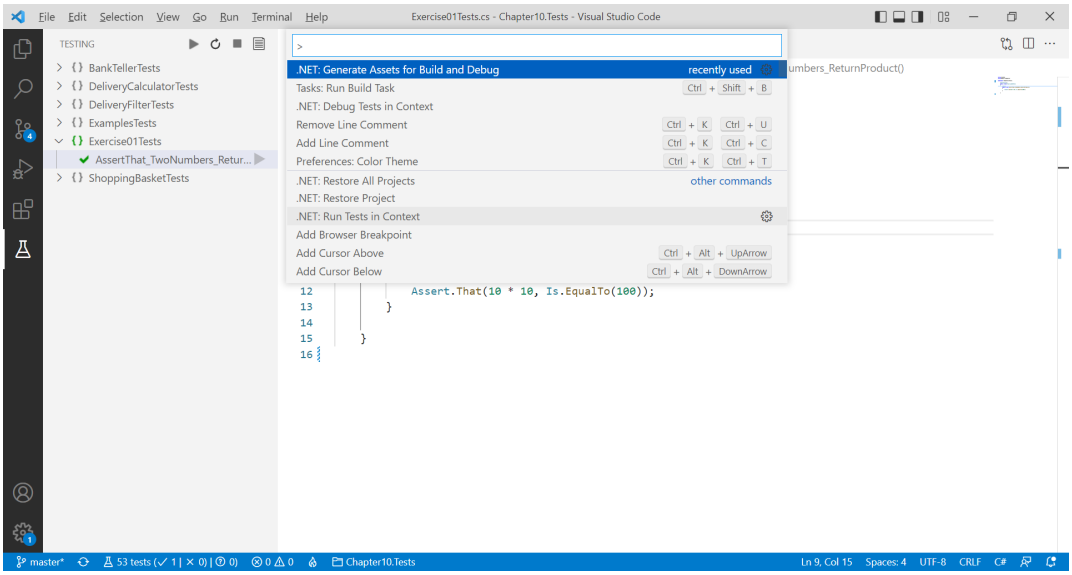

Figure 10.3: Unit test run directly from VS Code with ,NET Run Tests in Context option

The output window will show results similar to this:

```
 ----- Running test(s) in context "Exercise01Tests.cs(12,33)" -----

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.79

NUnit Adapter 4.1.0.0: Test discovery starting
NUnit Adapter 4.1.0.0: Test discovery complete
NUnit Adapter 4.1.0.0: Test execution started
Running selected tests in source\Chapter10\Chapter10.Tests\bin\Debug\
net6.0\Chapter10.Tests.dll
   NUnit3TestExecutor discovered 1 of 1 NUnit test cases using
Current Discovery mode, Non-Explicit run
NUnit Adapter 4.1.0.0: Test execution complete
----- Test Execution Summary -----

Chapter10.Tests.Exercise01Tests.AssertThat_TwoNumbers_ReturnProduct:
```

```
      Outcome: Failed
      Error Message:
        Expected: 300
  But was:  100

      Stack Trace:
         at Chapter10.Tests.Exercise01Tests.AssertThat_TwoNumbers_
ReturnProduct() in source\Chapter10\Chapter10.Tests\Exercise01Tests.
cs:line 12

Total tests: 1. Passed: 0. Failed: 1. Skipped: 0
```

Again, **`300`** is the constraint value that produces a failure. In this exercise, you saw how the **`TestFixture`** attribute is used to define a basic test class with the **`Test`** attribute, then assigned to a single test method. Your test method used the static **`Assert.That`** method to compare an actual value with an expected value, which was initially passed. You then altered the expected value (**`100`**) to one that was incorrect (**`300`**), which resulted in a test failure when run.

With this exercise, you created a new unit test project using the CLI and added a basic test that follows some of your conventions. Now proceed to learn how to carry out debugging in VS Code in three ways.

DEBUGGING UNITS TESTS

In addition to running unit tests to completion, you can also debug tests and pause at any breakpoints, as you would with any C# code. It is not often that you need to debug unit tests. The main reason for this is that you create them using small blocks of code. They initially run successfully and then continue running for as long as the project is maintained.

If a test fails, then the compiler will tell you which line caused the failure. From here, it is beneficial to debug in order to know why a test is failing. Maybe a parameter passed to a production method is incorrect. Or perhaps someone incorrectly altered the production implementation and forgot to consider a case that your tests had expected.

To carry out debugging from VS Code, you can follow either of these steps:

- Click the **`Debug Test`** option in the code window.
- Right-click on the code window and choose the **`Debug Tests in Context`** option.
- From the Command Palette option from the View menu, choose **`.NET: Debug Tests in Context`**.

Either of these options will launch the test runner, which will run each test method and allow you to pause and view variables interactively:

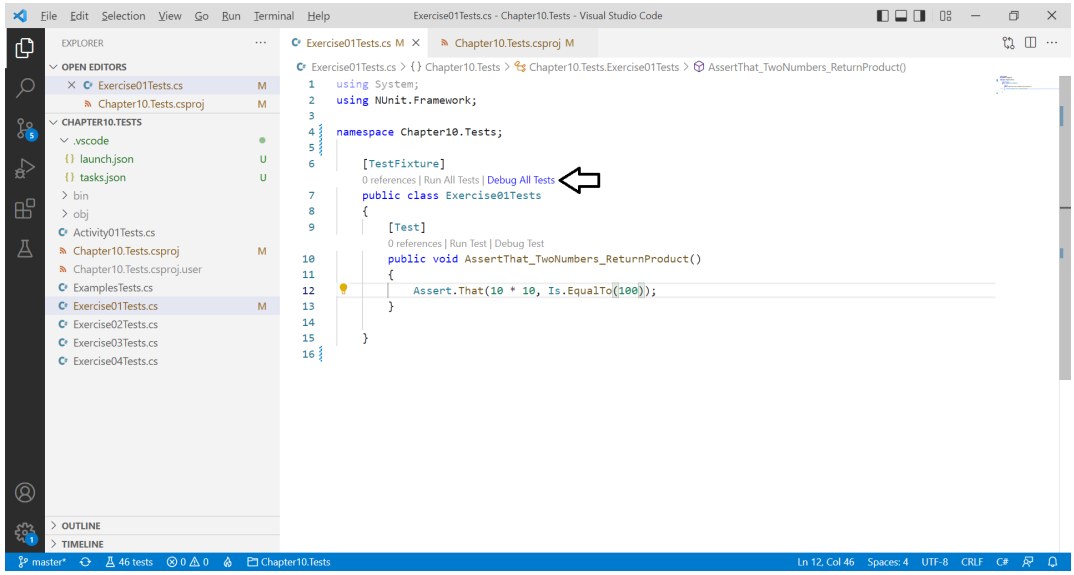

Figure 10.4: Debugging directly from VS Code with Debug Test option

This test method contains only one statement. However, if it were longer, it can be useful to set breakpoints and step through individual method calls.

You have looked at running unit tests using VSCode and **`dotnet`** command line. There are also alternative graphical UI-based tools to run unit tests, which you will read about next.

TEST EXPLORERS

If you prefer to use more interactive UIs to run or debug your unit tests, there are various unit test explorer extensions available for VS Code. The following are the two most popular extensions:

- .NET Core Test Explorer
- Test Explorer UI

These extensions show a list of unit tests within a project and typically a green tick or red cross to indicate a status of success or failure, respectively. To install the .NET Core Test Explorer extension, open the Extensions pan (or press **`Ctrl+Shift+X`**) and search for **`.NET Core Test Explorer`**:

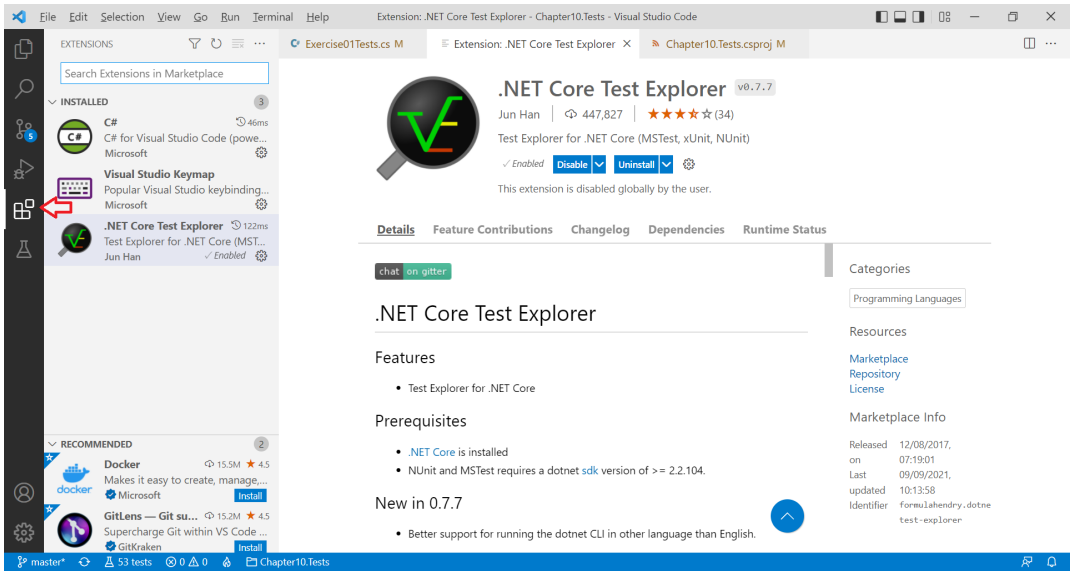

Figure 10.5: Adding the.NET Core Test Explorer Extension to VSCode

Once installed, a new Testing icon appears in the left pane. *Figure 10.6* shows the .NET Core Test Explorer extension:

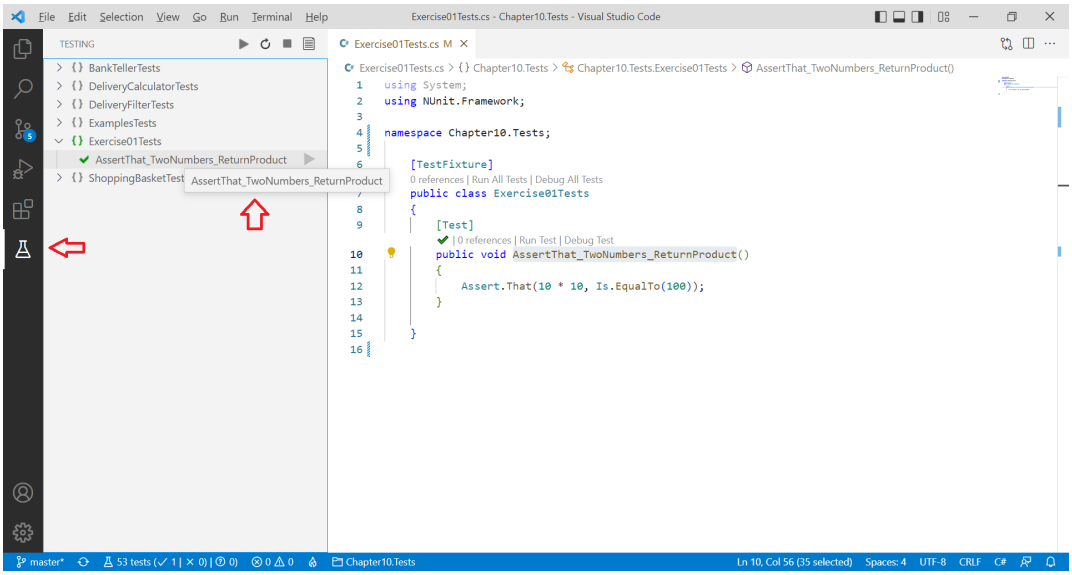

Figure 10.6: Running unit tests with .NET Core Test Explorer

In the left pane, click on the lower **`Tests`** icon to display a list of unit tests. In this example, a single unit test method called **`AssertThat_TwoNumbers_ReturnProduct`** has been run; a red cross beside it indicates that it failed to pass the test, whereas a green tick show success.

The .NET Core Test Explorer will need the following entry added to your **`.vscode\settings.json`** file, otherwise it will not be able to find **`Chapter10.Tests`** unit tests, as they are not located in the **`root`** project folder:

```
{
    "dotnet-test-explorer.testProjectPath":
      "**/*.Tests.csproj"
}
```

Now that you have reviewed the various ways to easily run or debug unit tests, take another look at the static **`Assert.That`** method and the various constraints that can be passed as the second argument in your assert-based test methods.

USING THE ASSERT CLASS TO VERIFY CODE

NUnit constraint methods can be chained together into a **fluent** style. You saw examples of this in *Chapter 4*, *Data Structures and LINQ*, where you joined LINQ extension methods together to provide more readable code.

For example, **Is.EqualTo(100)** can be chained to the **Not** constraint, as in **Is.Not.EqualTo(300)**, to invert the assertion. This style often provides a much clearer way to express the intent of your tests and is a popular style to follow.

There are almost 60 constraints available in **NUnit**. Each of these constraints can be accessed via the static **Is**, **Does**, or **Has** helper classes which can be found in the **Nunit.Framework** namespace.

> **NOTE**
>
> The official **NUnit** documentation contains further examples and can be found at https://docs.nunit.org/articles/nunit/writing-tests/constraints/Constraints.html.

In this section, you will look at the most commonly used constraints, learning how single value constraints can be verified using string and numeric variables. You will then cover collection and dictionary constraints. You may notice a certain amount of overlapping functionality between **Contains**, **Does**, and **Has**, particularly with collections. There are no definitive rules as to which class to use; it is often down to personal preference or whether you plan to compound constraints together into a fluent style.

Let's learn this concept through an example. Here you will need to start by including the **NUnit.Framework** namespace to access the **Test** attribute and **Assert** constraints. In the following **AssertThat_NumberExamples** test method, the **Is.LessThan** constraint is used to confirm that an expected value is less than an actual value. Since **2** is less than **3**, this assertion will pass when the test method is run:

```
using System;
using System.Collections.Generic;
using NUnit.Framework;

[TestFixture]
public class ExamplesTests
```

```
{
  [Test]
  public void AssertThat_NumberExamples()
  {
            Assert.That(2, Is.LessThan(3));
```

Next, for numeric ranges, **`Is.InRange`** allows a lower and upper range to be specified:

```
            Assert.That(5, Is.InRange(4, 6));
```

This example confirms that the number **`5`** is between **`4`** and **`6`**.

To invert an assertion, maybe for readability, the **`Not`** constraint offers the fluent style shown in the following snippet:

```
            Assert.That(10 + 2, Is.Not.EqualTo(20));
```

Here, you assert that the value of **`10 + 2`** is not equal to **`20`**.

The fluent style continues with the **`Within`** and **`Percent`** constraints in the next snippet, which are linked together to confirm that **`20.1`** is within **`1`** percent of **`20.2`**, as follows:

```
            Assert.That(20.1, Is.EqualTo(20.2).Within(1).Percent);
        }
}
```

Clicking the **`Run Test`** option in the VS Code will produce this result:

```
Chapter10.Tests.ExamplesTests.AssertThat_NumberExamples:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

In the next snippet, the **`IgnoreCase`** constraint is used for a case-insensitive string assertion. This test will pass as the mixed case string **`C# Workshop`** is equal to **`C# WORKSHOP`** if you apply the **`IgnoreCase`** constraint:

```
        [Test]
        public void AssertThat_StringExample()
        {
            Assert.That("C# Workshop", Is.EqualTo("C# WORKSHOP").
IgnoreCase);
        }
```

Running the preceding test method produces the following output:

```
Chapter10.Tests.ExamplesTests.AssertThat_StringExample:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

The **`Is`** class has method overloads that accept most built-in C# types. Here, the **`DateTime`** assertion checks that the last day in **`2020`** is before the first day in **`2021`**, using the **`LessThan`** constraint method:

```
        public void AssertThat_DateExamples()
        {
            Assert.That(new DateTime(2020, 12, 31),
                        Is.LessThan(new DateTime(2021, 1, 1)));
        }
```

Running the **`AssertThat_DateExamples`** test method produces this output:

```
Chapter10.Tests.ExamplesTests.AssertThat_DateExamples:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

Now, in the test class, the **`AssertThat_IsCommonlyUsed`** method is added as follows. This will be expanded further by creating further common uses of the **`Is`** constraint. At the end of the section, you'll run the test method in its entirety to confirm that all assertions are true:

```
[Test]
public void AssertThat_IsCommonlyUsed()
{
```

In the following snippet, the area of a rectangle is calculated. The result is expected to be positive, hence **`Is.Positive`** is used:

```
    const int Height = 10;
    const int Width = 20;
    var area = Height * Width;
    Assert.That(area, Is.Positive);
```

Next, for a list of integers (**`{-1, -3, -7, -9}`**), it can be verified that the values are in descending order and that all values are negative:

```
    var numbers = new List<int>{-1, -3, -7, -9};
    Assert.That(numbers, Is.Ordered.Descending);
    Assert.That(numbers, Is.All.Negative);
```

In the following snippet, the **DateTime** list contains no items. So **Is.Empty** can be used to confirm that **dates** is an empty list:

```
    var dates = new List<DateTime>();
    Assert.That(dates, Is.Empty);
```

For the next line of code, **Is.AtLeast** will pass if the current year is **2021** or later:

```
    Assert.That(DateTime.Now.Year, Is.AtLeast(2021));
```

Next, **Is.Zero** will pass if you subtract the amount **100** from an **OpeningBalance** of **100**:

```
    const int OpeningBalance = 100;
    const int AmountWithdrawn = 100;
    const int FinalBalance = OpeningBalance - AmountWithdrawn;
    Assert.That(FinalBalance, Is.Zero);
```

In .NET, **ApplicationException** is a subclass of the **Exception** type. You can confirm a value is a subclass of a type using the **Is.Instance** method and passing in the expected type, as in the following snippet:

```
    Assert.That(new ApplicationException(), Is.InstanceOf<Exception>());
```

To check that the two variables point to the same reference in memory, you can use **Is.SameAs**. In C#, calling **AddDays** on a **DateTime** instance creates a new **DateTime** instance. Here, you can verify this. By compounding **.Not**, you confirm that they do not point to the same instance:

```
    var today = DateTime.Today;
    var tomorrow = today.AddDays(1);
    Assert.That(today, Is.Not.SameAs(tomorrow));
```

Is.AnyOf is joined in a fluent style to the **.Not** constraint to verify that the day of the week list, **days**, does not contain a weekend:

```
    var days = new[] {DayOfWeek.Monday, DayOfWeek.Tuesday,
                      DayOfWeek.Wednesday};
    Assert.That(days, Is.Not.AnyOf(DayOfWeek.Saturday, DayOfWeek.
Sunday));
  }
}
```

Running the test using the **Run Test** command produces this output:

```
Chapter10.Tests.ExamplesTests.AssertThat_IsCommonlyUsed:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

> **NOTE**
>
> You can find the code used for this example at https://packt.link/5dKCc.

This section covered the most commonly used constraints explaining how single value constraints can be verified using string and numeric variables. With this knowledge, you will get to know the **Does** constraint class, as demonstrated in the following section.

USING THE DOES CLASS

The **Does** class contains the following methods primarily aimed at a string, collection, dictionary, filesystem, and regular expression-based constraints:

- **Does.StartWith** is used for string constraints to confirm that a string starts with a specific value (this may also be a string).
- **Does.EndWith** confirms that a string ends with a specific value.
- **Does.Contain** confirms that a collection contains a sub-item, or in the case of strings, that one string can be found inside another string.

In the following test method, you will confirm that the string **.Net Core**, starts and ends with certain strings:

1. Add a new test method called **AssertThat_DoesStringExamples** that shows the **Does** constraints in action:

```
[Test]
public void AssertThat_DoesStringExamples()
{
    Assert.That(".Net Core", Does.StartWith(".Net"));
    Assert.That(".Net Core", Does.EndWith("Core"));
```

   Here, the **Does.StartWith** constraint confirms that the string **".Net Core"** starts with **.Net** and **Does.EndWith** verifies that the string ends with **Core**.

2. Use the fluent style to append the **IgnoreCase** constraint:

```
    Assert.That("this is vs code", Does.Contain("VS").IgnoreCase);
```

The string **"this is vs code"** does contain **VS**. When the string's case is ignored, the next test is passed.

3. Next, use **Or** in a fluent style to confirm that the string **"red blue green"** either starts with **red** or ends with **green**:

```
    Assert.That("red blue green", Does.StartWith("red").
Or.EndWith("green"));
}
```

4. Running **AssertThat_DoesStringExamples** produces this output:

```
Chapter10.Tests.ExamplesTests.AssertThat_DoesStringExamples:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

5. In this example, the **AssertThat_StringListExamples** test method has a string array variable containing some names:

```
[Test]
public void AssertThat_StringListExamples()
{
    var names = new[] { "paul", "craig", "richard", "geoff", "dan" };
```

6. Next, **Contains.Item** confirms that **names** contains the string **"paul"**. An alternative option is to use **Does.Contain** to search for **geoff**:

```
    Assert.That(names, Contains.Item("paul"));
    Assert.That(names, Does.Contain("geoff"));
```

The **Contains.Item** and **Does.Contain** provide the same functionality.

7. Use the **Has.No.Member** fluent way to check whether an element exists in a collection:

```
    Assert.That(names, Has.No.Member("jason"));
}
```

8. Running the **AssertThat_StringListExamples** test gives the following output:

```
Chapter10.Tests.ExamplesTests.AssertThat_StringListExamples:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

9. Now, use an integer array along with the **`Has.Some.GreaterThan`** constraint to check whether at least one item in the array exists with a value greater than **`90`**:

```
[Test]
public void AssertThat_HasSomeGreaterThan()
{
    var angles = new [] {45,90, 135, 180};
    Assert.That(angles, Has.Some.GreaterThan(90));
}
```

10. Clicking **`Run Test`** in the code window runs the test and the following output can be seen:

```
Chapter10.Tests.ExamplesTests.AssertThat_HasSomeGreaterThan:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

> **NOTE**
>
> You can find the code used for this example at https://packt.link/5dKCc.

In this section, you learned about the **`Does`** class that contains the methods primarily aimed at a string, collection, dictionary, filesystem, and regular expression-based constraints. With this knowledge, you can now proceed to dictionary constraints, as discussed in the following section.

DICTIONARY CONSTRAINTS

You saw earlier that a dictionary is used to store collections of items, using a **key** that must be unique to each item. The **`Does.ContainKey`** and **`Does.ContainValue`** are used to verify that a dictionary contains the specified keys or values, respectively.

For example, in the following snippet, **`Does.ContainKey("UK")`** will pass as there is a value with the key **`UK`**. The second assertion will also pass, as there is a value of **`3000`** (for **`France`**) in the **`new Dictionary`**:

```
[Test]
public void AssertThat_DoesDictionary()
{
    var valuesByCountry = new Dictionary<string, int>
    {
```

```
        {"UK", 1000},
        {"Spain", 2000},
        {"France", 3000}
    };
    Assert.That(valuesByCountry, Does.ContainKey("UK"));
    Assert.That(valuesByCountry, Does.ContainValue(3000));
}
```

Running the test method produces the following output:

```
Chapter10.Tests.ExamplesTests.AssertThat_DoesDictionary:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

The primary reason for creating unit tests is to verify whether the application code is correct in a way that is consistent, reproducible, and easy to run. The set of assertions confirms that your application code does exactly what you expect it to do and continues to do even after the code has been changed. In large or complex applications, a minor, or seemingly unrelated, code change may have unintended consequences. However, by running unit tests regularly, you can build a level of confidence so that regression bugs do not creep into an application's code.

With that in mind, you can test your own class for correctness in the following exercise.

EXERCISE 10.02: USING RECORDS TO VERIFY PATTERN-MATCHING CORRECTNESS

C#'s pattern matching feature offers a flexible way to process data, particularly if that data is unrelated in a class hierarchy. In this exercise, you will create a calculator that works out the charge for a courier to deliver certain items, as per the following rules:

- Envelopes are priced by the distance covered, at $1.00 per mile.
- Packages less than 10 kg (22 pounds approximately) are priced at $2; anything more is $3.00 per pound/kg.
- Cold meals are $4 per item.
- Hot meals are $6 per item.

With the rules defined, there will likely be multiple execution paths through the code found in such a delivery cost calculator. As the number of routes through code increases, it becomes increasingly difficult to test code manually using a combination of breakpoints and stepping through code, so you will need automated unit tests to help verify correctness.

The first part of the exercise creates the calculator library, while the second creates a test for the calculator. Perform the following steps to do this exercise:

1. In the **`Chapter10`** source folder, add a new class library project called **`Chapter10.Lib`** using the following command:

```
dotnet new classlib -o Chapter10.Lib
```

2. Rename the **`Class1.cs`** file to **`Exercise02.cs`** and remove the template code.
3. Open **`Chapter10.Lib\Chapter10.Lib.csproj`** and replace the contents with the following:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <LangVersion>10.0</LangVersion>
  </PropertyGroup>
</Project>
```

4. Start by creating a C# **`record`** that represents an envelope.
5. Use **`Chapter10.Lib`** namespace to keep this separate from any other classes. The **`Envelope`** constructor takes a single **`distance`** argument, which you assign to the **`Distance`** property:

```
using System;
namespace Chapter10.Lib
{
    public record Envelope
    {
        public Envelope(int distance)
            => (Distance) = (distance);

        public int Distance { get; }
    }
```

6. Do the same for a class named **`Package`**, with the constructor being passed the **`distance`** and **`weight`** arguments:

```
    public record Package
    {
        public Package(int distance, int weight)
            => (Distance, Weight) = (distance, weight);
```

```
    public int Distance { get; }
    public int Weight { get; }
}
```

7. Pass a Boolean **`isHot`** value to the **`Meal`** record's constructor:

```
public record Meal
{
    public Meal(bool isHot) => (IsHot) = (isHot);

    public bool IsHot { get; }
}
```

This determines whether the record represents a hot or cold meal, which, according to the requirements, will be charged at different prices.

8. Declare the **`DeliveryCalculator`** class as **`static`** so that callers can use it without needing to create an instance, as there is no state needed between calls:

```
public static class DeliveryCalculator
{
```

9. In the next snippet, make the **`GetCost`** function accept an object type and return a **`double`** value:

```
    public static double GetCost(object item)
```

You could have created a hierarchy for the **`Envelope`**, **`Package`**, and **`Meal`** classes, but, in this example, you want to use C# pattern matching that allows you to pass objects that are unrelated by type.

10. Use a **`switch`** statement with the **`item`** as the source:

```
        => item switch
        {
```

11. Now identify the types, starting with an envelope, which is **`$1`** per mile, and add the following code:

```
            Envelope e
                => e.Distance * 1,
```

12. For **`Package`** types, use a **`when`** statement to return a different price for items for instances where the weight is less than **`10`**:

```
                    Package p when p.Weight < 10
                        => p.Distance * 2,

                    Package p when p.Weight >= 10
                        => p.Distance * 3,
```

13. For **`Meal`** types, use a **`when`** statement to return a fixed price for hot or cold meals:

```
                    Meal m when !m.IsHot
                        => 4,

                    Meal m when m.IsHot
                        => 6,
```

14. Add the following code to throw an exception for unknown types, as something may go wrong if the caller passes in an object that isn't recognizable:

```
                    { }
                        => throw new ArgumentException(),
```

15. It's also important to handle a null reference being passed in. So, ensure the caller should not pass in a null object to deliver by adding the following code:

```
                    null
                        => throw new ArgumentNullException()
                };
        }
    }
```

Now that the types and rules are defined, you can look at adding some unit tests to verify this code.

The **`GetCost`** method has seven possible paths through the code, which you could manually step through to confirm the correctness, but that would soon get quite frustrating. So, you need to create a unit test that verifies the flow through each code path, rather than creating tests that check every possible value for distance and weight combinations.

16. From the **source\Chapter10** folder, run the **dotnet add** command to add a **Chapter10.Lib** reference to **Chapter10.Tests** project:

```
dotnet add Chapter10.Tests\Chapter10.Tests.csproj reference
Chapter10.Lib\Chapter10.Lib.csproj

Reference `..\Chapter10.Lib\Chapter10.Lib.csproj` added to the
project.
```

17. Add a new file called **Exercise02Tests.cs** to **Chapter10.Tests** folder by using statements for **NUnit.Framework** and **Chapter10.Lib** as you will be using objects from both:

```
using NUnit.Framework;
using Chapter10.Lib;

namespace Chapter10.Tests
{
```

18. Add the **TestFixture** attribute to the **DeliveryCalculatorTests** class:

```
    [TestFixture]
    public class DeliveryCalculatorTests
    {
```

19. To test various types of objects, start by creating tests that target **Envelope** objects first. Rather than creating a set of individual tests with various values, you can use the **TestCase** attribute to pass parameters into your test code; the same test method will run using each **TestCase** combination. The test method needs to have the same number of arguments as specified in the **TestCase** attribute:

- For a distance of **0**, you expect the cost to be **0**.
- For a distance of **5** the cost should be **5** (as the charge is **$1** per mile).

Now start off with two test cases:

```
        [TestCase(0)]
        [TestCase(5)]
        public void GetCost_Envelope_DistanceEqualsCost(int distance)
        {
```

The test method is called twice, passing **0** and then **5** as the distance for each test case parameter.

20. Use **`Is`** to create an **`Envelope`** object using **`distance`**.
21. Pass this to the **`DeliveryCalculator.GetCost`** method with the resulting **`actualCost`** verified against **`Assert.That`** and **`Is.EqualTo`** for equality:

```
            var envelope = new Envelope(distance);
            var actualCost = DeliveryCalculator.GetCost(envelope);
            Assert.That(actualCost, Is.EqualTo(distance));
        }
    }
}
```

22. From the **`source\Chapter10\Chapter10.Tests`** folder, run the **`dotnet test`** command to run the tests. You should see the following output:

```
source\Chapter10\Chapter10.Tests>dotnet test
A total of 1 test files matched the specified pattern.
Passed!  - Failed:     0, Passed:     5, Skipped:     0, Total:
5,
```

You can add a test for packages by using the **`Random`** attribute to pick a random distance between **`0`** and **`100`**.

23. Keeping the package condition in mind (packages less than 22.04 pounds/ 10 kg are priced at $2), use the **`Random`** attribute to pick five random weights between **`0`** and **`9`**:

```
        [Test]
        public void GetCost_LightPackage_ProductOfDistance(
            [Random(0, 100, 5)]int distance,
            [Random(0, 9, 5)]int weight)
        {
            var package = new Package(distance, weight);
            var actualCost = DeliveryCalculator.GetCost(package);
            var expectedCost = distance * 2;
            Assert.That(actualCost, Is.EqualTo(expectedCost));
        }
```

> **NOTE**
>
> The third parameter to the **`Random`** attribute represents the number of times to pick a random value.

24. To test the heavy-weight packages, use the **`Values`** attribute to specify **three** distinct distances and **two** distinct weights:

```
[Test]
public void GetCost_HeavyPackage_ProductOfDistance(
    [Values(100, 200, 300)]int distance,
    [Values(10, 20)]int weight)
{
    var package = new Package(distance, weight);
    var actualCost = DeliveryCalculator.GetCost(package);
    var expectedCost = distance * 3;
    Assert.That(actualCost, Is.EqualTo(expectedCost));
}
```

This will result in five test cases being run. Notice that you expect the cost to be **`$3`** multiplied by the **`distance`** for these heavy packages.

25. To create a test for meals, use a **`TestCase`** attribute to drive two test cases by passing in a **`bool`** to indicate whether the food **`isHot`** or not at an **`expectedCost`**:

```
[TestCase(false, 4.0)]
[TestCase(true, 6.0)]
public void GetCost_Meal_FixedCost(
    bool isHot,
    double expectedCost)
{
    var meal = new Meal(isHot);
    var actualCost = DeliveryCalculator.GetCost(meal);
    Assert.That(actualCost, Is.EqualTo(expectedCost));
}
```

If a developer inadvertently changes any of the rules within **`DeliveryCalculator.GetCost`**, then the suite of unit tests will fail when run.

Your tests will verify the delivery cost for known types of objects. But have you wondered what may happen if you pass an unknown object or even a null reference? The last two statements in **`GetCost`** will throw an **`ArgumentException`** or **`ArgumentNullException`** in these circumstances.

Using the **`Assert.Throws`** extension method, it is possible to create tests that will **fail** unless a certain type of exception is thrown. **`Assert.Throws`** is passed a generic parameter that represents the type of exception that is expected, such as **`ArgumentException`**, along with a lambda statement to execute. The test method will pass only if the lambda statement is executed by **`Assert.Throws`** does throw the expected type of exception.

26. Add a **`new`** test method to **`Chapter10.Tests\Exercise02Tests.cs`** that verify the unknown type behavior. Also, use the standard **`[Test]`** attribute declared with the test **`new`** method:

```
[Test]
public void GetCost_UnknownType_ThrowsException()
{
    Assert.Throws<System.ArgumentException>( () =>
      {
        DeliveryCalculator.GetCost(new object());
      });
}
```

Notice how you make a call to **`DeliveryCalculator.GetCost`** passing **`new object()`** using a **lambda** statement.

27. Run the tests from the **`source\Chapter10\Chapter10.Tests`** folder by running the **`dotnet test`** command:

```
source\Chapter10\Chapter10.Tests>dotnet test
Passed!  - Failed:     0, Passed:    39, Skipped:     0, Total:    39
```

You will see that **`39`** tests were run from your combination of **`Test`**, **`TestCase`**, **`Values`**, and **`Random`** attributes.

In this exercise, you created a delivery calculator class to calculate the price for certain types of items. Although the calculator code is not particularly complex, it does have various code paths that can be executed at runtime.

You guaranteed that the calculator was correct by generating various tests, some with randomized data, to capture all of the rules that you were aware of. You can now be confident that the calculator will remain correct. If someone inadvertently changes these rules, these unit tests will then fail.

> **NOTE**
>
> The library code for this exercise can be found at https://packt.link/OyKye. You can find the unit test code for this exercise at https://packt.link/0PeHr.

You have seen how single values can be easily verified. Let's now look at how collections of objects can be verified.

VERIFYING COLLECTIONS USING COLLECTION CONSTRAINTS

You have seen how single value properties can be verified for correctness; a similar set of attributes and **`Assert`** calls can also be used for collections of objects. In this section, you will look at collections and verify whether the items in those collections are as expected and that they appear in a specific order.

You used the **`Assert.That`** and **`Is.EqualTo`** methods to verify numeric values were equal. In addition to single values, types that implement the standard **`IEnumerable`** interface, such as **`array`** and **`List`**, can also be asserted. As long as both the expected and actual parameters contain the same items, in the same order, the assertion will pass. You do not need to walk through the results comparing individual references, as **`Is.EqualTo`** will do that for you.

In this example, an array and a list of string colors are compared. As each contains the same three strings in the same order, the assertion will pass:

```
[Test]
public void AssertThat_IsEqualToExamples()
{
  var newColors = new[] { "red", "orange", "blue" };
  var oldColors = new List<string> { "red", "orange", "blue" };

  Assert.That(newColors, Is.EqualTo(oldColors));
}
```

When the **AssertThat_IsEqualToExamples** test method is run, it will pass because the **newColors** array and the **oldColors** string **List** do contain the same string elements ("red", "orange" and "blue"). The output is displayed as follows:

```
ExamplesTests.AssertThat_IsEqualToExamples:
    Outcome: Passed
Total tests: 1. Passed: 1. Failed: 0. Skipped: 0
```

You have seen how to string collections of objects can be verified. Let's take this further by creating an exercise that verifies a collection that contains instances of your objects.

EXERCISE 10.03: VERIFYING COLLECTIONS

As part of a delivery scheduling system, you are tasked with creating a method that returns a filtered and sorted list of **Package** items. The following rules must be applied to the filter:

- Each package must be middleweight that is, weighing between **10** and **20** units.
- The list must be sorted, with those that have the farthest distance first in the results.
- A delivery driver can only carry a certain number of items, and this can vary. So, you must allow the caller to specify the maximum number of items to be returned by this filter.

Perform the following steps to complete this exercise:

1. Add a new class file called **Exercise03.cs**.You will reuse the **Package** record that was defined in **Exercise02.cs** earlier
2. Include the **System.Linq** namespace for access to **Linq** methods.
3. Include the **System.Collections.Generic** namespace to use the **List<T>** generic collection class:

```
using System.Linq;
using System.Collections.Generic;

namespace Chapter10.Lib
{
```

4. Name the class **DeliveryFilters**, rather than keeping the name **Exercise03**. This is because **DeliveryFilters** contains various filters:

```
    public static class DeliveryFilters
    {
```

By using the **static** keyword, you declare that the entire class is **static** that is, members can be invoked without needing to create an instance.

5. Add a new **static** method called **GetSortedMiddleWeightPackages**. This method is passed a source list of packages to be filtered, along with the number of **items** to be returned by the filter.

6. Use the three **Linq** methods—**Where**, which calls a local **IsMiddleWeightPackage** method to find the middle-weight packages, second **OrderByDescending** to sort by distance, and finally **Take** to return the number of items requested:

```
        public static Ienumerable<Package>
GetSortedMiddleWeightPackages(
             Ienumerable<Package> items, int top)
        {
            return items
                .Where(IsMiddleWeightPackage)
                .OrderByDescending(p => p.Distance)
                .Take(top);
```

7. Now add a local **IsMiddleWeightPackage** method, as follows, to complete the class:

```
            static bool IsMiddleWeightPackage(Package package)
                => package.Weight is (>= 10 and <= 20);
        }
    }
}
```

Although it is a small utility method, there are several sections that could be incorrect or inadvertently changed, such as the weight ranges within **`IsMiddleWeightPackage`** or the sorting used by **`OrderByDescending`**.

> **NOTE**
>
> You can put the body of this call into the **`Where`** call in the preceding code, but it is more readable this way and shows how you identify a middle-weight package.

To test that this correctly filters a list of packages, create a unit test that compares the **actual** and **expected** outcomes. To do so, do as follows:

1. In **`Chapter10.Tests`** project, add a new class file called **`Exercise03Tests.cs`**.
2. Include the **`Nunit.Framework`** namespace for the **`Assert`** and **`Test`** attributes.
3. Include **`Chapter10.Lib`** since you want to test the **`DeliveryFilters`** class.
4. Include the **`System.Collections.Generic`** namespace for generic lists:

```
using Nunit.Framework;
using Chapter10.Lib;
using System.Collections.Generic;

namespace Chapter10.Tests
{
```

5. Name the class **`DeliveryFilterTests`** and add the **`TestFixture`** attribute, as follows:

```
    [TestFixture]
    public class DeliveryFilterTests
    {
```

6. Now create a test method for testing the **`GetSortedMiddleWeightPackages`** method with multiple items and expect an exact order. These input and expectation requirements are used to create a descriptive method name **`GetSortedMiddleWeightPackages_MultipleItems_ExactOrder`**:

```
        [Test]
        public void GetSortedMiddleWeightPackages_MultipleItems_
ExactOrder()
        {
```

7. Arrange your test by creating three **`Package`** instances, each with increasing distances (all between **`10`** and **`20`** in weight):

```
            // ARRANGE
            var nearPackage = new Package(10_000, 10);
            var midPackage = new Package(20_000, 15);
            var farPackage = new Package(30_000, 20);
```

8. Create a **`List`** of packages with your three known packages interspersed.

9. The second argument to **`Package`** is the weight, so include some heavy and lightweight packages to test your middle-weight function correctly:

```
            var packages = new List<Package>()
            {
                new Package(100, 20),
                nearPackage,
                new Package(200, 30),
                midPackage,
                new Package(50_000, 1),// far distance but light
weight
                farPackage
            };
```

10. For readability, create an array containing the packages in the order that they are expected:

```
            var expected = new []
            {
                farPackage, midPackage, nearPackage
            };
```

11. Call **`GetSortedMiddleWeightPackages`** and pass in your source packages and the number **`3`** to get the top three middle-weight packages by distance:

```
            // ACT
            var actual = DeliveryFilters
                .GetSortedMiddleWeightPackages(packages, 3);
```

12. Use the **`Is.EqualTo`** constraint method to assert that two collections contain the **same** objects in exactly the **same** order. You could have used **`Is.EquivalentTo`** to assert that two collections are equivalent that is, they contain the same items in any order. However, you want to confirm that the packages are in the correct order—that is, **`farPackage`**, **`midPackage`**, and the **`nearPackage`** instance.

13. So, in this case, use **`Assert.That`** and pass the **`Is.EqualTo`** constraint method with the **`actual`** and **`expected`** values:

```
            // ASSERT
            Assert.That(actual, Is.EqualTo(expected));
        }
    }
}
```

Notice how you are comparing an **`array`** and a **`List`**. It does not matter that they are of different types but they both implement the standard **`Ienumerable`** interface, as seen in *Chapter 4*, *Data Structures and LINQ*. As long as they both contain the same items in the same order, this test will pass. You don't need to walk through the results comparing references, as **`Is.EqualTo`** will do that for you.

14. Running the tests using the **`dotnet test`** produces a successful result:

```
Chapter10\Chapter10.Tests>dotnet test
Passed!  - Failed:     0, Passed:    40, Skipped:     0, Total:
40, Duration: 403 ms - Chapter10.Tests.dll (net5.0)
```

15. Try one more collection verification example. Add a new method called **`GetLastTwoPackages`** that uses a range to return the last two items from a collection.

16. In the **`DeliveryFilters`** class (in **`Exercise03.cs`**), add the following **`GetLastTwoPackages`** method:

```
        public static Package[] GetLastTwoPackages(Package[] items)
        {
```

```
            return items[^2..^0];
        }
```

The **index from end operator** (**^**) is used to return elements relative to the end of the sequence. So here when you are using **^2..^0**, you get the last two elements back.

17. The **GetLastTwoPackages** is a small method and could easily be changed incorrectly. So, you will add a test for it. Add a new test method **GetLastTwoPackages_MultipleItems_ExactOrder** to the **DeliveryFilterTests** class (**Exercise03Tests.cs** in the **Chapter10.Tests** project) to check if this works as expected:

```
        [Test]
        public void GetLastTwoPackages_MultipleItems_ExactOrder()
        {
            // ARRANGE
            var package1 = new Package(20_000, 15);
            var package2 = new Package(30_000, 20);
```

18. Create an array of **Package** instances:

```
            var packages = new []
            {
                new Package(100, 1),
                new Package(200, 1),
                new Package(300, 1),
                package1,
                package2
            };
```

You will start off with three new **Package** instances and add **package1** and **package2** variables to the end of an array. You could have added any number of new instances before, but you want to test that the last two are the correct instances returned.

19. Create an array of items named **expected**. This will contain the package instances that you expect to be returned—that is, **package1** and **package2**:

```
            var expected = new []
            {
                package1, package2
            };
```

20. Call the **`GetLastTwoPackages`** method returning the results to a variable called **`actual`**:

```
            // ACT
            var actual = DeliveryFilters.GetLastTwoPackages(packages);
```

21. Finally, verify whether the **`actual`** and **`expected`** variables contain the same package instances using the **`Is.EqualTo`** constraint:

```
            // ASSERT
            Assert.That(actual, Is.EqualTo(expected));
        }
```

    Remember that **`Is.EqualTo`** will check whether the items are present and are sorted in the correct order. It is not checking the returned array's reference.

22. Run the tests once more using the **`dotnet test`** to get the following successful result:

```
Chapter10\Chapter10.Tests>dotnet test
Passed!  - Failed:     0, Passed:    41, Skipped:     0, Total:
41, Duration: 220 ms - Chapter10.Tests.dll (net5.0)
```

With this exercise, you have seen how **`NUnit`** can be used to confirm that object properties and collections are correct when applied to potentially complex methods.

> **NOTE**
>
> The library code for this exercise can be found at https://packt.link/QjwSC. You can find the unit test code for this exercise at https://packt.link/DlsuK.

The next section will look at how object interactions can be verified using similar assert techniques.

VERIFYING INTERACTIONS BETWEEN OBJECTS

Often you need to pass a source object into a class and want to be certain that a method on your source object is called a certain number of times. For example, a shopping basket application may have an **`ApplyDiscount`** method that reduces the value of an order by a certain percentage. You probably would not want **`ApplyDiscount`** to be called more than once as doing so would indicate a bug somewhere.

By using interfaces, you can create **mock** objects that inherit from an interface. This interface definition can be declared in a target class method. Passing a mock object based on an interface allows you to make a note of how many times a certain method is called by a target. Perform the following steps to do so:

1. In order to make a shopping basket application, first create the **`EmailMessage`** class that represents an email that is to be sent. It contains **`To`** and **`Body`** properties:

```
public class EmailMessage
{
    public string To { get; set; }
    public string Body { get; set; }
}
```

2. Now create an **`IMailSender`** interface that contains a single **`SendMessage`** method as follows:

```
public interface IMailSender
{
    void SendMessage(EmailMessage message);
}
```

3. Define a **`MarketingService`** class whose constructor is passed an instance of the class that inherits from the **`IMailSender`** interface:

```
public class MarketingService
{
    private readonly IMailSender _sender;

    public MarketingService(IMailSender sender)
    {
        _sender = sender;
    }
```

4. Add a **`SendMailShot`** method that invokes the **`IMailSender.SendMessage`** when called:

```
    public void SendMailShot(string to, string body)
    {
        _sender.SendMessage(new EmailMessage{To=to, Body = body});
        _sender.SendMessage(new EmailMessage{To=to, Body = "test
message"});
    }
}
```

   You may notice that something is not quite correct with this method, as it's calling **`SendMessage`** twice. If you then proceeded to create an instance of the **`MarketingService`** class, passing in an **`EmailSender`** instance that sends a real email message, you may find that your customers become rather annoyed.

   They would receive two emails each time the **`SendMailShot`** is called—one containing the correct body text and the other with the words **`test message`**.

   By using mock objects and interfaces, you can easily create tests that verify how many times the **`IMailSender.SendMessage`** method is called by **`MarketingService`** without the risk of real emails being sent out.

> **NOTE**
>
> You can find the code used for this example at https://packt.link/JAvz0.

There are many excellent mocking frameworks available for .NET that can be used within unit tests to verify correctness. Such frameworks are also known as **isolation frameworks**. This will not be covered in detail in this chapter.

> **NOTE**
>
> Some effective isolation frameworks are FakeItEasy (https://fakeiteasy.github.io), Moq (https://packt.link/lpC5W), and Microsoft Fakes (https://docs.microsoft.com/en-us/visualstudio/test/isolating-code-under-test-with-microsoft-fakes).

This concludes the theoretical portion of this topic. In the following section, you will put this into practice with an exercise.

EXERCISE 10.04: VERIFYING INTERACTIONS BETWEEN OBJECTS

This exercise aims to show you how to verify interactions between objects. You will create a class named **`ShoppingBasket`** that is passed an **`ICustomerDiscount`** interface. The shopping basket may call the **`ICustomerDiscount.Apply`** method.

For your test to pass, **`Apply`** must have been called by the shopping basket either **once** or **not at all**. Anything else is a failure. In this example, a valid discount code must be any of the following characters: **`a`**, **`b`**, or **`c`**.

Perform the following steps to complete this exercise:

1. In **`Chapter10.Lib`** project, add a new class file called **`Exercise04.cs`**:

```
using System;

namespace Chapter10.Lib
{
```

2. Create an **`Order`** record that has **`TotalValue`** and **`Quantity`** properties set via the constructor.
3. To make the **`Order`** immutable, use C# init-only set properties:

```
    public record Order
    {
        public Order (double totalValue, int quantity)
            => (TotalValue, Quantity) = (totalValue, quantity);

        public double TotalValue {get; init;}
        public int Quantity { get; init; }
    }
```

   Any changes will result in a new **`Order`** instance being created.

4. Create the **`IDiscounter`** interface:

```
    public interface IDiscounter
    {
        Order Apply(Order order);
    }
```

   The callers of this interface do not need to know what type of discounter is being used; just that it follows the **`IDiscounter`** interface contract.

5. Create a class for the shopping basket. The constructor should take **`IDiscounter`** and the initial **`Order`**, as follows:

```
    public class ShoppingBasket
    {
        private readonly IDiscounter _discounter;

        public ShoppingBasket(IDiscounter discounter, Order order)
        {
            _discounter = discounter;
            Order = order;
        }

        public Order Order {get; private set;}
```

6. Make the basket's **`ApplyDiscount`** method to only call the discounter's **`Apply`** method, if the code is correct (for this, you are using a C# pattern) and a discount has not already been applied:

```
        private bool _discountApplied;

        public void ApplyDiscount(char code)
        {
            if (_discountApplied)
                return;

            if (code is (>= 'a' and <= 'c'))
            {
                Order = _discounter.Apply(Order);
                _discountApplied = true;
            }
        }
```

In the **`Chapter10.Tests`** project, add a new class called **`Exercise04Tests.cs`**.

7. To test the **`ApplyDiscount`** behavior, create a **`MockDiscounter`** class as follows (based on **`IDiscounter`**):

```
using NUnit.Framework;
using Chapter10.Lib;

namespace Chapter10.Tests;
```

```
internal class MockDiscounter : IDiscounter
{
  public int ApplyCalls {get; private set;}
  public Order NewOrder {get; private set;}
  public Order Apply(Order order)
  {
    ApplyCalls ++;
```

The **MockDiscounter** class simply counts the number of times the **Apply** method is called by incrementing **ApplyCalls** once each time.

8. Call **Apply** in order to return a new **Order** instance:

```
            NewOrder = order with {};
            return NewOrder;
        }
    }
```

Here a C# **with** expression is passed along with an empty property list **{ }** as you want an exact copy of the original **Order**. This is assigned to the **NewOrder** property before returning.

This test checks whether **Apply** is called rather than checking that an order is created with altered properties. If you wanted to test whether the discount is applied correctly, create a test specifically to test the order's **TotalValue** property.

9. For the actual test class and test method, use four **TestCase** attributes, passing a voucher code character along with the number of times you expect the mock discounter's **Apply** method to be called by the shopping basket:

```
    [TestFixture]
    public class ShoppingBasketTests
    {
        [TestCase('a', 1)]
        [TestCase('b', 1)]
        [TestCase('c', 1)]
        [TestCase('d', 0)]

        public void ApplyDiscount_TestCase_CallsApply(char code,
            int expectedCalls)
        {
```

10. To arrange the test, first declare a new **`MockDiscounter`** using the **`new()`** expression and an initial **`Order`**. Both of these are passed to the new **`ShoppingBasket`** instance:

```
            // ARRANGE
            MockDiscounter discounter = new();
            var order = new Order(19.99, 2);
            var basket = new ShoppingBasket(discounter, order);
```

11. Next, intentionally call **`ApplyDiscount`** twice:

```
            // ACT
            basket.ApplyDiscount(code);
                  basket.ApplyDiscount(code);
```

    This is because you want to ensure that no matter how many times **`ApplyDiscount`** is called, the **`ShoppingBasket`** only ever calls the discounter's **`Apply`** method once and only if the code is valid.

12. Now define the assertion:

```
            // ASSERT
            Assert.That(discounter.ApplyCalls,
Is.EqualTo(expectedCalls));
        }
    }
```

    You compare the expected number of calls to the **`MockDiscounter.Apply instance`** with the actual number of calls.

13. Run the four test cases using the **`Run All Tests`** option:

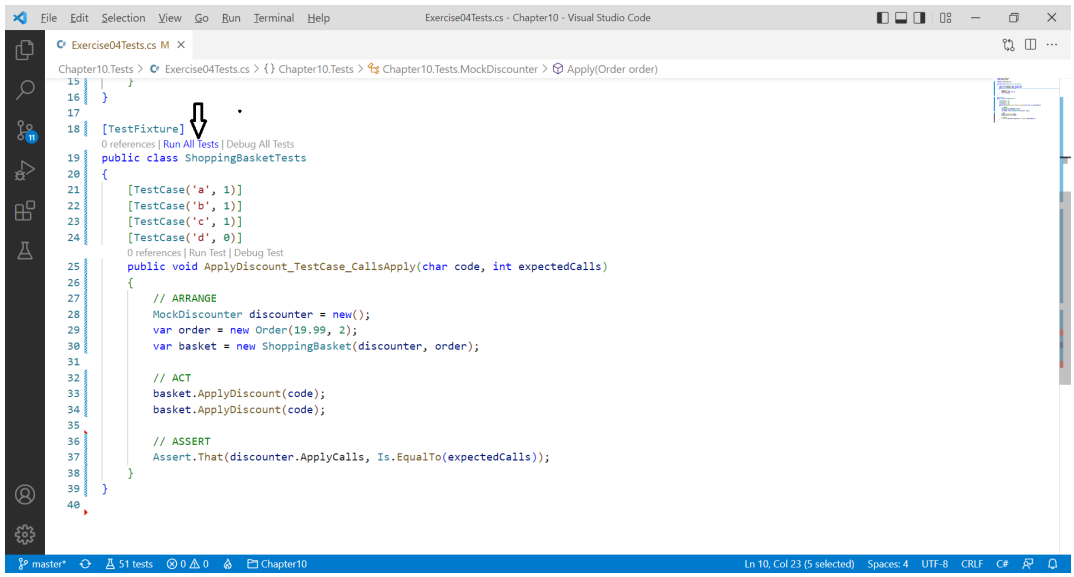

Figure 10.7: Running Test Cases with the Run All Tests option

The following output gets displayed:

```
NUnit3TestExecutor discovered 4 of 4 NUnit test cases using Current
Discovery mode, Non-Explicit run
NUnit Adapter 4.1.0.0: Test execution complete

----- Test Execution Summary -----

Chapter10.Tests.ShoppingBasketTests.ApplyDiscount_TestCase_
CallsApply('a',1):    Outcome: Passed
Chapter10.Tests.ShoppingBasketTests.ApplyDiscount_TestCase_
CallsApply('b',1):    Outcome: Passed
Chapter10.Tests.ShoppingBasketTests.ApplyDiscount_TestCase_
CallsApply('c',1):    Outcome: Passed
Chapter10.Tests.ShoppingBasketTests.ApplyDiscount_TestCase_
CallsApply('d',0):    Outcome: Passed
Total tests: 4. Passed: 4. Failed: 0. Skipped: 0
```

> **NOTE**
>
> The library code for this exercise can be found at https://packt.link/8RPMh. You can find the unit test code for this exercise at https://packt.link/GWJKM.

You have now covered the key aspects of unit testing with C#. In the following activity, you will see how multiple assertions can be used to verify the state of classes in an application.

ACTIVITY 10.01: CREATING A BANK TELLER

You are asked to create a **`BankTeller`** class that allows money to be transferred from one account to another. It is essential that the following business rules are always followed:

- The application must throw an **`ApplicationException`**, if the amount to transfer is less than or equal to zero.
- The application must throw an **`ApplicationException`**, if the from account does not have a sufficient balance to cover the amount requested.
- Each account must contain a list of numeric transactions that record the amount deducted or added to the account.
- Each account must have an opening balance.
- Each account must have a balance calculated from the opening balance and the sum of the transactions.
- For the **`BankTeller`** to be accepted as correct, you will need to create a class library and a suite of **`NUnit`**-based C# tests that verify whether the preceding rules are applied.

Perform the following steps to complete this activity:

1. In **`Chapter10.Lib`** project, add a **`BankAccount`** class.
2. Ensure that the **`BankAccount`** class has an **init-only** double **`OpeningBalance`** property, set via the constructor.
3. The **`BankAccount`** class should have a **`List<double> Transactions`** property and **`Balance`** function to return the current balance.
4. Add a static class named **`BankTeller`** that has a **`Transfer`** method. This method should accept a **`from`** and **`to BankAccount`** instance and a **`double`** value for **`amount`**, for the value to transfer. The **`from`** account should have a negative-signed amount added to its **`Transactions`** and the **`to`** account the same amount but positive-signed.

5. The transfer should throw an **`ApplicationException`** as per rule one (the amount to transfer is less than or equal to zero) and rule two (the account has insufficient balance to cover the amount requested). So add the code for this.
6. In **`Chapter10.Tests`** project, add a **`BankTellerTests`** class.
7. Use **`dotnet add reference`**, if you haven't already added a project reference to the **`Chapter10.Lib`** and **`Chapter10.Test projects`**:

```
dotnet add Chapter10.Tests\Chapter10.Tests.csproj reference
Chapter10.Lib\Chapter10.Lib.csproj
```

8. Add a suitably named test method (such as **`Transfer_BalanceTooLow_ThrowsException`**) that verifies the **`BankTeller.Transfer`** method correctly adds **one** negative-signed numeric value to the **`from`** account and adds **one** positively signed amount to the **`to`** account.
9. Both accounts' **`Balance`** methods must correctly calculate the value on demand. Ideally, each account should be passed a random double for the opening balance.
10. Add a test method that detects an **`ApplicationException`** being thrown for rule one violations.
11. Add a test method that detects an **`ApplicationException`** being thrown for rule two violations.
12. To run the **`BankTeller`** tests, click the **`Run All Tests`** option at the top of the **`Activity01Tests.cs`** code window.

    You will see an output similar to this:

```
----- Running tests in class "Chapter10.Tests.BankTellerTests" -----
NUnit Adapter 4.1.0.0: Test execution complete
----- Test Execution Summary -----
Chapter10.Tests.BankTellerTests.Transfer_BalanceTooLow_
ThrowsException:
    Outcome: Passed


Chapter10.Tests.BankTellerTests.Transfer_NegativeAmount_
ThrowsException:
    Outcome: Passed
```

```
    Chapter10.Tests.BankTellerTests.Transfer_ValidBalance_
    AddsToTransactions:
        Outcome: Passed


    Total tests: 3. Passed: 3. Failed: 0. Skipped: 0
```

> **NOTE**
>
> The solution to this activity can be found at https://packt.link/qclbF.

In completing this activity, you have seen how business rules can be defined and applied to multiple target classes, and how exceptions can be correctly caught and verified.

SUMMARY

In this chapter, you saw how production results can be verified for correctness. Small unit test methods act as an automated first line of defense against subtle bugs. The two main unit testing frameworks, **`MSTest`** and **`NUnit`**, were introduced and you saw how similar their implementations are when creating unit tests. **`NUnit`** was then covered in further detail due to it being the more popular of the two frameworks.

By applying some of the common **`NUnit`** constraints to variables and collections, you verified relatively small code paths using tests that were simple to run, fast to execute, and always gave the same results no matter how random the test parameters were. You extended this to show that interactions between classes can also be verified using interfaces and mock objects. From this, you can see how automated testing can be applied to large and complex projects with many interacting classes, particularly in the early stages where efficient design and tests can help document a system.