Bonus Chapter 19: Automated Testing

In this chapter, we will cover the following:

- Generating and running a default test script class
- Performing a simple unit test
- Parameterizing tests with a data provider method
- Unit testing a simple health script class
- Creating and executing a unit test in PlayMode
- PlayMode testing a door animation
- PlayMode and unit testing a player health bar with events, logging, and exceptions

Introduction

For a very simple computer program, we can write code, then run it, entering a variety of valid and invalid data, and see whether the program behaves as we expect it to. This is known as a code-then-test approach. However, this approach has several significant weaknesses:

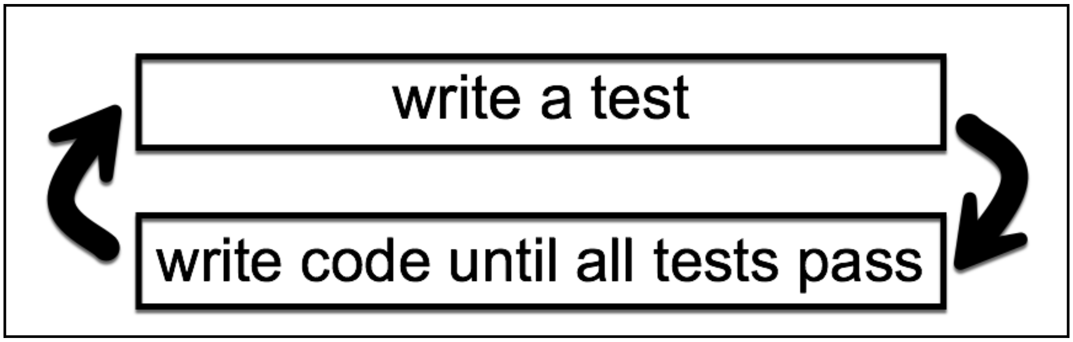
- Each time we change the code, as well as running new tests relating to the code we are improving, we have to run all the old tests to ensure that no unexpected modified behaviors have been introduced (in other words, our new code has not **broken** another part of our program)
- Running tests manually is time consuming
- We are relying on a human to rerun the test each time, and this test may be run using different data, or some data may be omitted, or different team members may take a different approach to running tests

Therefore, even for simple programs (and most are not simple), some kind of fast, automated testing system makes a lot of sense.

The big picture

There is an approach to software development called **Test-Driven Development** (**TDD**), whereby code is **only** written until all tests pass. So, if we want to add or improve the behavior of our game program, we must specify what we want in terms of tests, and then the programmers write code to pass the tests. This avoids a situation whereby programmers write code and features that are not needed, or spend time over-optimizing things that would have been fine, and so on. It means that the game development team directs its work toward agreed goals understood by all, since they have been specified as tests.

The following diagram illustrates basic TDD in that we only write code until all tests pass. Then it's time to write more tests:



Another way that TDD is often summarized is as red-green-refactor:

- **red:** We write code that fails the test (in other words, for the new feature/improved behavior we wish to add to our system)
- **green:** We write code that passes the new test (and all the existing ones)
- **refactor:** We (may) choose to improve the code (and ensure that the improved code passes all the tests)

Two kinds of software test are the following:

- **Unit tests**
- **Integration tests**

Unit tests

A **Unit Test** tests a "unit" of code, which can be a single method, but which may include some other computer work being executed between the method being tested and the end result(s) being checked.

"A unit test is a piece of code that invokes a unit of work and checks one specific end result of that unit of work. If the assumptions on the end result turn out to be wrong, the unit test has failed."
—Roy Oshergrove (p. 5, The Art of Unit Testing (Second edition).

Unit tests should be as follows:

- automated (runnable at the "push of a button")
- fast
- easy to implement
- easy to read
- executed in isolation (tests should be independent from one another)
- assessed as either having being passed or failed
- relevant tomorrow
- consistent (the same results each time!)
- able to easily pinpoint what was at fault for each test that fails

Most computer languages have an xUnit unit testing system available, for example:

- C#: NUnit
- Java: JUnit
- PHP: PHPUnit

Unity offers an easy way to write and execute NUnit tests in its editor (and at the command line).

Typically, each unit test will be written in three sections, a sequence of:

- **Arrange**: Set any initial values needed (sometimes, we are just giving a value to a variable in order to improve code readability)
- **Act**: Invoke some code (and, if appropriate, store the results)
- **Assert**: Make assertions for what should be true about the code invoked (and any stored results)

Observe that the naming of a unit test method (by convention) is quite verbose—it is made up of lots of words that describe what it does. For example, you might have a unit test method named `TestHealthNotGoAboveOne()`. The idea is that if a test fails, the name of the test should give a programmer a very good idea of what behavior is being tested and, therefore, how to quickly establish whether the test is correct and, if so, where to look in your program code for what was being tested. Another part of the convention of naming unit tests is that numerals are not used—ust words—so we write "one", "two", and so on, in the name of the test method.

Integration tests (PlayMode tests in Unity)

An **Integration Test** involves checking the behavior of interacting software components, for example, ones that use real time, or a real filesystem, or that communicate with the web or other applications running on the computer. Integration tests are usually not as fast as unit tests, and may not produce consistent results (since the components may interact in different ways at different times).

Both **Unit** and **Integration Test** are important, but they are different and should be treated differently.

Unity offers **Play Mode** testing, allowing integration testing as Unity scenes execute with testing code in them.

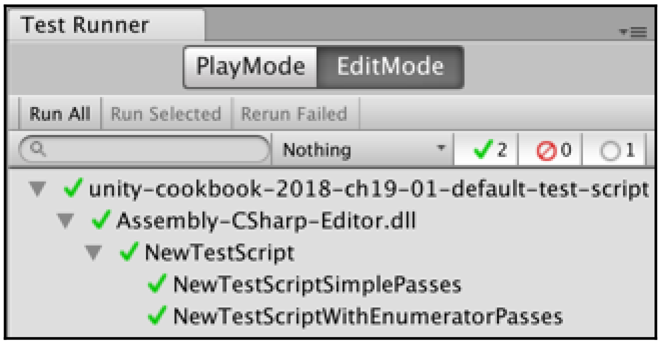
Places where you can learn more about Unity Testing include the following:

- Unity Test Runner and PlayMode documentation pages:
    - `https://docs.unity3d.com/Manual/testing-editortestsrunner.html`
    - `https://docs.unity3d.com/Manual/PlaymodeTestFramework.html`
- A website for the book "The Art of Unit Testing" (and lots of other learning resources associated with testing): `http://artofunittesting.com/`
- A great dual article tutorial about Unity testing by Tomek Paszek of Unity (talking about the old Unity test tools, but most of the content is still very relevant): `https://blogs.unity3d.com/2014/06/03/unit-testing-part-2-unit-testing-monobehaviours/`
- YouTube, where you can learn lots about Unity testing (and other topics) from Infalliblecode: `https://www.youtube.com/infalliblecode`
- CodeProject.com's introduction to TDD and NUnit: `https://www.codeproject.com/Articles/162041/Introduction-to-NUnit`

```
    -and-TDD
```

Generating a default test script class

Unity can create a default C# test script for you, thereby enabling you to quickly start creating and executing tests on your project:



unity-cookbook-2018-ch19-01-default-test-script
Assembly-CSharp-Editor.dll
NewTestScript
NewTestScriptSimplePasses
NewTestScriptWithEnumeratorPasses

How to do it...

To generate a default test script class, follow these steps:

1. In the **Project** panel, create a folder called **Editor**
2. Display the Test Runner panel by choosing the following menu: **Window | General | Test Runner**
3. Ensure that the **EditMode** button is selected in the **Test Runner** panel
4. Ensure that your new **Editor** folder is selected in the **Project** panel
5. In the **Test Runner** panel, click the **Create Test Script in the current folder** button
6. You should now have a new C# script added to your **Editor** folder
7. To run the tests in your script class, click the **Run All** button in the **Test Running panel**
8. You should now see all green ticks (check marks) in the panel

How it works...

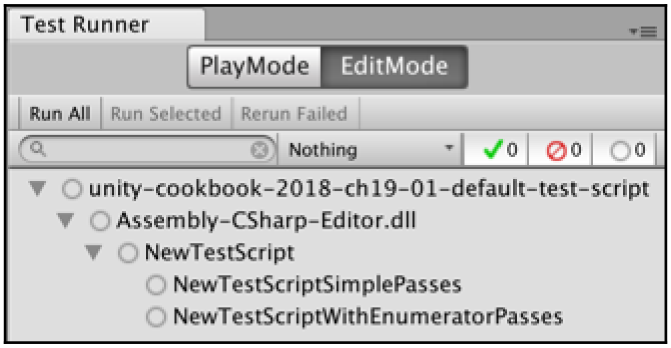
Unity checks that you have a folder named **Editor** selected in the **Project** panel, and then creates a C# NewTestScript script class for you containing the following:

```
using UnityEngine;
 using UnityEngine.TestTools;
 using NUnit.Framework;
 using System.Collections;

 public class NewTestScript {
     [Test]
     public void NewTestScriptSimplePasses() {
         // Use the Assert class to test conditions.
     }

     // A UnityTest behaves like a coroutine in PlayMode
     // and allows you to yield null to skip a frame in EditMode
     [UnityTest]
     public IEnumerator NewTestScriptWithEnumeratorPasses() {
         // Use the Assert class to test conditions.
         // yield to skip a frame
         yield return null;
     }
 }
```

In the **Test Runner** panel, you should see the script class and its two methods listed. Note that the first line in the **Test Runner** panel is the Unity project name, the second line will say `Assembly-CSharp-Editor.dll`, followed by your script class name, and then each of the test methods:



unity-cookbook-2018-ch19-01-default-test-script
Assembly-CSharp-Editor.dll
NewTestScript
NewTestScriptSimplePasses
NewTestScriptWithEnumeratorPasses

There are three symbols to indicate the status of each test/class:

- **Empty circle**: Test not executed since the script class was last changed

- **Green tick** (check mark): The test was passed successfully
- **Red cross**: The test was failed

There's more...

Here are some details that you won't want to miss.

Create a default test script from the Project panel's Create menu

Another way of creating a default **Unit Test** script is as follows:

- From the **Project** panel, chose the following menu: **Create** | **Testing** | **C# Test Script**

Edit mode minimum skeleton unit test script

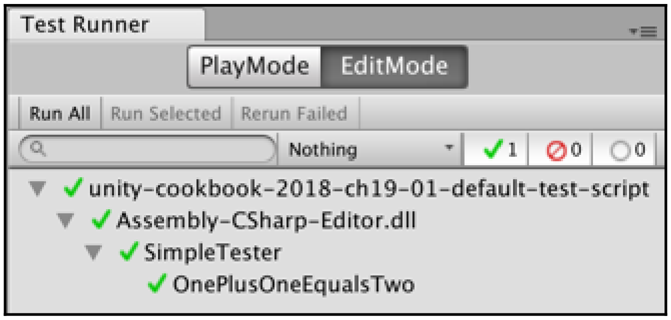
Be aware that if you are only going to use this script class for testing in **EditMode**, you can delete the second method and some of the using statements as follows, so as to give you a minimal skeleton to work from:

```
using NUnit.Framework;

 public class UnitTestSkeleton
 {
     [Test]
     public void NewTestScriptSimplePasses()
     {
         // write your assertion(s) here
     }
 }
```

A simple unit test

In the same way as printing "hello world" is most programmers first program statement, asserting that 1 + 1 = 2 is perhaps the most common first test executed for those learning unit testing. That's what we'll create in this recipe:



unity-cookbook-2018-ch19-01-default-test-script
Assembly-CSharp-Editor.dll
SimpleTester
OnePlusOneEqualsTwo

How to do it...

To create and execute a simple unit test, follow these steps:

1. In the **Project** panel, create a folder called **Editor**.
2. Inside your **Editor** folder, create a new C# `SimpleTester.cs` script class containing the following:

```
using NUnit.Framework;

 class SimpleTester
 {
     [Test]
     public void TestOnePlusOneEqualsTwo()
     {
         // Arrange
         int n1 = 1;
         int n2 = 1;
         int expectedResult = 2;

         // Act
         int result = n1 + n2;

         // Assert
         Assert.AreEqual(expectedResult, result);
     }
 }
```

3. Display the **Test Runner** panel by choosing the following menu: **Window** | **General** | **Test Runner**.
4. Ensure that the `EditMode` button is selected in the **Test Runner** panel.
5. Click **Run All**.

6. You should see the results of your **Unit Test** being executed – if the test was concluded successfully, it should have a green 'tick' next to it.

How it works...

You have declared that the `TestOnePlusOneEqualsTwo()` method in the C# `SimpleTester.cs` script class is a test method. When executing this test method, the **Unity Test Runner** executes each statement in sequence, so variables `n1`, `n2`, and `expectedResult` are set, then the calculation of 1 + 1 is stored in the variable result, and finally (the most important bit), we make an assertion of what should be true after executing that code. Our assertion states that the value of the `expectedResult` variable should be equal to the value of the variable result.

If the assertion is true, the test is passed, otherwise it is failed. Generally, as programmers, we expect our code to pass, so we inspect each fail very carefully, first to see whether we have an obvious error, then perhaps to check whether the test itself is correct (especially if it's a new test), and then to begin to debug and understand why our code behaved in such a way that it did not yield the anticipated result.

There's more...

Here are some details that you won't want to miss.

Shorter tests with values in the assertion

For simple calculations, some programmers prefer to write less test code by putting the values directly into the assertion. So, as shown below, our 1 + 1 = 2 test could be expressed in a single assertion, where the expected value of 2, and the expression 1 + 1, are entered directly into an `AreEqual(...)` method invocation:

```
using NUnit.Framework;

 class SimpleTester
 {
     [Test]
     public void TestOnePlusOneEqualsTwo()
     {
         // Assert
         Assert.AreEqual(2, 1 + 1);
     }
```

```
        }
```

However, if you are new to testing, you may prefer the previous approach, whereby the preparation, code execution, and storage of results, and the assertion of properties about those results, are structured clearly in a sequence of **Arrange/Act/Assert**.

Expected value followed by the actual value

When comparing values with assertions, it is customary for the expected (correct) value to be given first, followed by the actual value:

```
       Assert.AreEqual( <expectedValue>, <actualValue> );
```

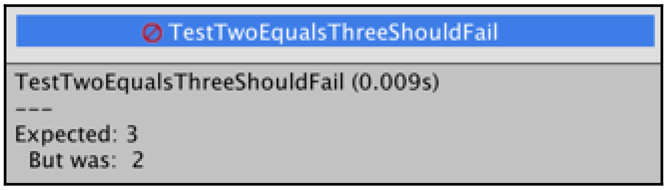
While it makes no difference to the true or false nature of equality, and so on, it can make a difference to messages when tests fail with some testing frameworks (for example, "got 2 but expected 3" has a very different meaning to "got 3 but expected 2"). Hence, the following assertion would output a message that would be confusing, since 2 was our expected result:

```
       public void TestTwoEqualsThreeShouldFail() {
            // Arrange
            int expectedResult = 2;

            // Act
            int result = 1 + 2; // 3 !!!

            // Assert
            Assert.AreEqual(result, expectedResult);
        }
```

Refer to the following screenshot:

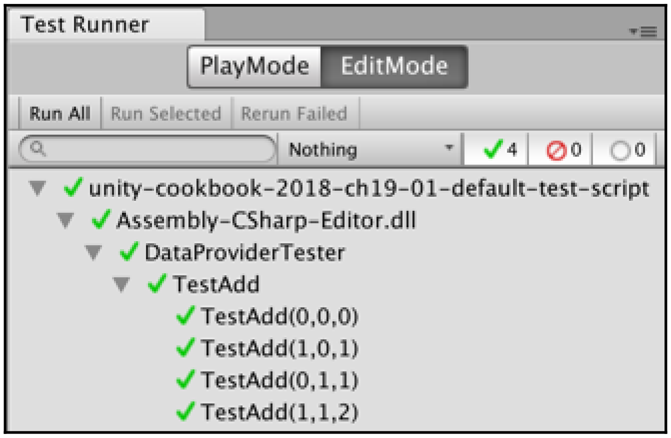


```
TestTwoEqualsThreeShouldFail
TestTwoEqualsThreeShouldFail (0.009s)
---
Expected: 3
  But was:  2
```

Parameterizing tests with a data provider

method

If we are testing our code using a range of test data, then sometimes there is little difference between each test apart from the the values. Rather than duplicating our Arrange/Act/Assert statements, we can re-use a single method, and the Unity Test Runner will loop through a collection of test data, running the test method for each set of test data. The special method that provides multiple sets of test data to a test method is known as a **DataProvider**, and we'll create one in this recipe:



```
unity-cookbook-2018-ch19-01-default-test-script
  Assembly-CSharp-Editor.dll
    DataProviderTester
      TestAdd
        TestAdd(0,0,0)
        TestAdd(1,0,1)
        TestAdd(0,1,1)
        TestAdd(1,1,2)
```

How to do it...

To parameterize tests with a data provider method, follow these steps:

1. In the **Project** panel, create a folder called **Editor**.
2. Inside your **Editor** folder, create a new C# `DataProviderTester.cs` script class containing the following:

```
using NUnit.Framework;

 class DataProviderTester
 {
     [Test, TestCaseSource("AdditionProvider")]
     public void TestAdd(int num1, int num2, int expectedResult)
     {
         // Arrange
         // (not needed - since values coming as arguments)

         // Act
```

```
            int result = num1 + num2;

            // Assert
            Assert.AreEqual(expectedResult, result);
        }

        // the data provider
        static object[] AdditionProvider =
        {
            new object[] { 0, 0, 0 },
            new object[] { 1, 0, 1 },
            new object[] { 0, 1, 1 },
            new object[] { 1, 1, 2 }
        };
    }
```

3. Display the **Test Runner** panel by choosing the following menu: **Window** | **General** | **Test Runner**.
4. Ensure that the **EditMode** button is selected in the **Test Runner** panel.
5. Click **Run All**.
6. You should see the results of your **Unit Test** being executed. You should see four sets of results for the `TestAdd(...)` test method, one for each of the datasets provided by the `AdditionProvider` method.

How it works...

We have indicated that the `TestAdd(...)` method is a test method with a compiler attribute [Test]. However, in this case, we have added additional information to state that the data source for this method is the `AdditionProvider` method.

This means that the **Unity Test Runner** will retrieve the data objects from the additional provider, and create multiple tests for the `TestAdd(...)` method, one for each set of data from the `AdditionProvider()` method.

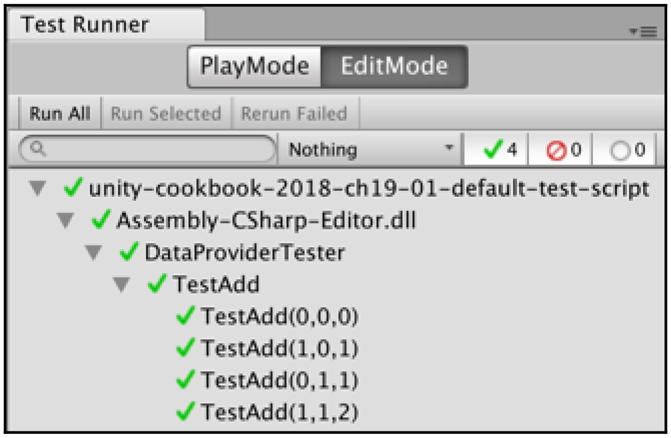
In the **Test Runner** panel, we can see a line for each of these tests:

```
TestAdd(0,0,0)
 TestAdd(1,0,1)
 TestAdd(0,1,1)
 TestAdd(1,1,2)
```

Unit testing a simple health script class

Let's create something that might be used in a game, and that can easily be unit tested. Classes that do **not** subclass from Monobehavior are much easier to unit test, since instance objects can be created using the keyword **new**. If the class is carefully designed with private data and public methods with clearly declared dependencies as parameters, it becomes easy to write a set of tests to make us confident that objects of this class will behave as expected in terms of default values, as well as valid and invalid data.

In this recipe, we will create a health script class, and a set of tests for this class. This kind of class can be reused for both the health of human players, and also Artificial Intelligence (AI)-controlled enemies in a game:



```
Test Runner
PlayMode  EditMode
Run All | Run Selected | Rerun Failed
Nothing   ✓4  ⊘0  ○0
▼ ✓ unity-cookbook-2018-ch19-01-default-test-script
  ▼ ✓ Assembly-CSharp-Editor.dll
    ▼ ✓ DataProviderTester
      ▼ ✓ TestAdd
          ✓ TestAdd(0,0,0)
          ✓ TestAdd(1,0,1)
          ✓ TestAdd(0,1,1)
          ✓ TestAdd(1,1,2)
```

How to do it...

To unit test a health script class, follow these steps:

1. In the **Project** panel, create a **_Scripts** folder.
2. Inside your `_Scripts` folder, create a new C# `Health.cs` script class containing the following:

```
using UnityEngine;
 using System.Collections;

 public class Health
 {
     private float health = 1;

     public float GetHealth()
     {
         return health;
     }

     public bool AddHealth(float heathPlus)
     {
         if(heathPlus > 0){
             health += heathPlus;
             return true;
         } else {
             return false;
         }
     }

     public bool KillCharacter()
     {
         health = 0;
         return true;
     }
 }
```

3. Inside your **_Scripts** folder, create a new folder named **Editor**.
4. Inside your **Editor** folder, create a new C# `TestHealth.cs` script class containing the following:

```
using NUnit.Framework;

class TestHealth {
   [Test]
   public void TestReturnsOneWhenCreated()   {
      // Arrange
      Health h = new Health ();
      float expectedResult = 1;

      // Act
      float result = h.GetHealth ();

      // Assert
      Assert.AreEqual (expectedResult, result);
   }

    [Test]
    public void TestPointTwoAfterAddPointOneTwiceAfterKill()
{
        // Arrange
```

```
            Health h = new Health();
            float healthToAdd = 0.1f;
            float expectedResult = 0.2f;

            // Act
            h.KillCharacter();
            h.AddHealth(healthToAdd);
            h.AddHealth(healthToAdd);
            float result = h.GetHealth();


            // Assert
            Assert.AreEqual(expectedResult, result);
        }

        [Test]
        public void
    TestNoChangeAndReturnsFalseWhenAddNegativeValue()     {
            // Arrange
            Health h = new Health();
            float healthToAdd = -1;
            bool expectedResultBool = false;
            float expectedResultFloat = 1;

            // Act
            bool resultBool = h.AddHealth(healthToAdd);
            float resultFloat = h.GetHealth();

            // Assert
            Assert.AreEqual(expectedResultBool, resultBool);
            Assert.AreEqual(expectedResultFloat, resultFloat);
        }

        [Test]
        public void TestReturnsZeroWhenKilled()    {
            // Arrange
            Health h = new Health();
            float expectedResult = 0;

            // Act
            h.KillCharacter();
            float result = h.GetHealth();

            // Assert
            Assert.AreEqual(expectedResult, result);
        }
```

```
            [Test]
            public void TestHealthNotGoAboveOne()    {
                // Arrange
                Health h = new Health();
                float expectedResult = 1;

                // Act
                h.AddHealth(0.1f);
                h.AddHealth(0.5f);
                h.AddHealth(1);
                h.AddHealth(5);
                float result = h.GetHealth();

                // Assert
                Assert.AreEqual(expectedResult, result);
            }
        }
```

5. Display the **Test Runner** panel by choosing the following menu: **Window | Debug | Test Runner**.
6. Ensure that the **EditMode** button is selected in the **Test Runner** panel.
7. Click **Run All**.
8. You should see the results of your unit tests being executed.

How it works...

Each of the C# script-classes is described below.

Script-class `Health.cs`

This script class has one private property; as it is private, it can only be changed by methods. Its initial value is 1.0, in other words, 100% health:

- `health` (float): The valid range is from 0 (dead!) to 1.0 (100% health)

There are 3 public methods:

- `GetHealth()`: This returns the current value of the health float number (which should be between 0 and 1.0)
- `AddHealth(float)`: This takes as input a float (the amount to add to the health), and returns a Boolean true/false, as to whether the value was valid. Note the logic of this method is that it accepts values of 0 or more (and will

return true), but it will ensure that the value of health is never more than 1
- `KillCharacter()`: This method sets health to zero, and returns true, since it is always successful in this action

Script-class `TestHealth.cs`

This script class has five methods:

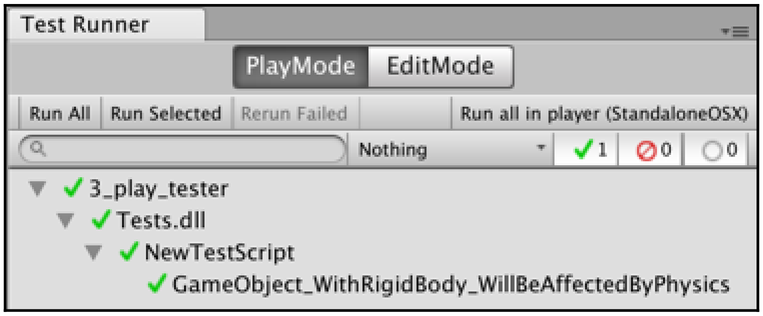
- `TestReturnsOneWhenCreated()`: This tests that the initial value of health is 1, when a new Health object is created.
- `TestPointTwoAfterAddPointOneTwiceAfterKill()`: This tests that after a kill (health set to zero), and then adding 0.1 on two occasions, the health should be 0.2.
- `TestReturnsZeroWhenKilled()`: This tests that the health value is set to zero immediately after the `KillCharacter()` method has been called.
- `TestNoChangeAndReturnsFalseWhenAddNegativeValue()`: This tests that attempting to add a negative value to health should return false and that the value of health should not have changed. This method is an example of a test with more than one assertion (but both are related to the actions.
- `TestHealthNotGoAboveOne()`: This test verifies that even when lots of values are added to health, totaling more than 1.0, the value returned from `GetHealth()` is one.

Hopefully, all the tests pass when you run them, giving some confidence that the logic implementation in the `Health.cs` script class does behave as intended.

Creating and executing a unit test in Play mode

It's a good idea to write as much of the logic for a game as isolated, non-Monobehavior classes, that are easy to unit test in Edit mode. However, some of the logic in a game relates to things that happen when the game is running. Examples include physics, collisions, and timing-based events. We test these parts of our games in Play Mode.

In this recipe, we'll create one very simple Play Mode test, to check that physics affects a RigidBody (based on an example from the Unity documentation):

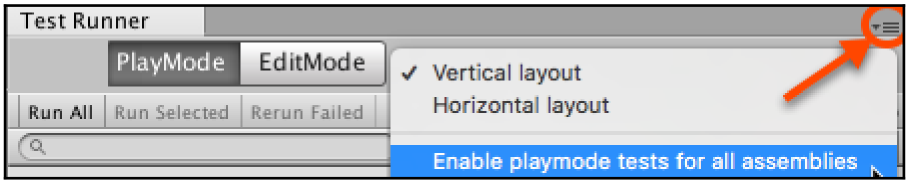


3_play_tester
Tests.dll
NewTestScript
GameObject_WithRigidBody_WillBeAffectedByPhysics

How to do it...

To create and execute a unit test in **Play** mode, follow these steps:

1. Display the **Test Runner** panel by choosing the following menu: **Window** | **General** | **Test Runner**
2. Enable PlayMode tests for all assemblies. Do this by displaying the drop-down menu in the top-right corner of the **Test Runner** panel, and then selecting **Enable playmode tests for all assemblies** (click OK to any message concerned with restarting the editor):



3. Now, restart the Unity Editor (just close the application and then reopen it with your project).

It is very important that you restart the Unity Editor application after enabling **PlayMode**. If you fail to do this, then you may not be able to locate your **PlayMode** test script classes where they can see (and refer to) your Monobehavior classes.

4. Ensure that the **PlayMode** button is selected in the **Test Runner** panel.
5. In the **Test Runner** panel, click the **Create PlayMode Test Assembly Folder** button. A new folder, named **Tests**, should have been created.
6. In the **Project** panel, open the `Tests` folder. It should contain an assembly

definition file **Tests.asmdef**.

7. In the **Test Runner** panel, click the **Create Test Script in the current folder** button – you may wish to rename this script from the default name, `NewTestScript`.
8. Edit your new test script, replacing the content with the following:

```
      using UnityEngine;
       using UnityEngine.TestTools;
       using NUnit.Framework;
       using System.Collections;

       public class NewTestScript
       {
           [UnityTest]
           public IEnumerator
  GameObject_WithRigidBody_WillBeAffectedByPhysics()
           {
               // Arrange
               var go = new GameObject();
               go.AddComponent<Rigidbody>();
               var originalPosition = go.transform.position.y;

               // Act
               yield return new WaitForFixedUpdate();

               // Assert
               Assert.AreNotEqual(originalPosition,
  go.transform.position.y);
           }
       }
```

9. Click **Run All**.
10. In the Hierarchy, you'll see that a temporary scene is created (named something along the lines of InitTestScene6623462364), and that a **GameObject** named **Code Based Test Runner** is created.
11. In the **Game** panel, you will briefly see the message **Display 1 No Cameras Rendering**.
12. You should see the results of your unit test being executed – if the test is concluded successfully, it should have a green tick next to it.

How it works...

Methods marked with the `[UnityTest]` attribute are run as coroutines. A coroutine has the ability to pause execution (when it meets a yield statement) and return control to Unity, but then to continue where it left off when called again (for example, the next frame, second, or whatever). The yield statement indicates the statement after which, and for how long, execution of the method is to be paused. Examples of different types of yield include:

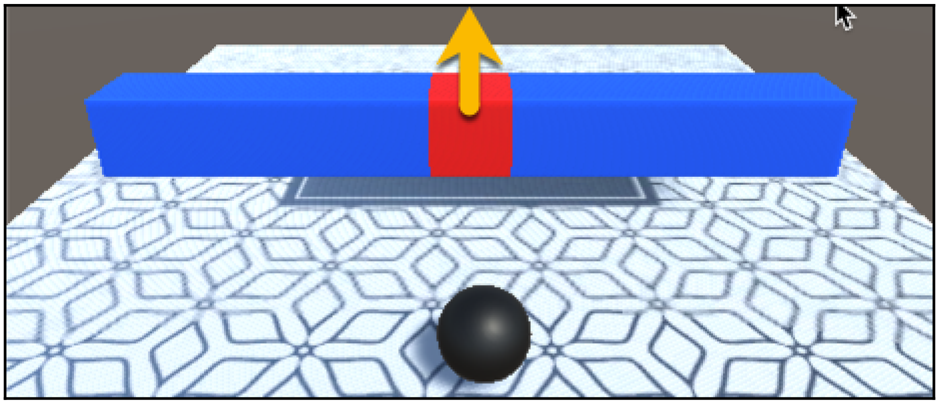
- Waiting until until the next frame: `null`
- Waiting for a given length of time: `WaitForSeconds(<seconds>)`
- Waiting until the next fixed-update time period (physics is not applied each frame (since the framerate varies), but after a fixed period of time): `WaitForFixedUpdate()`

Method `GameObject_WithRigidBody_WillBeAffectedByPhysics()` creates a new GameObject and attaches to it a RigidBody. It also stores the original Y position. The yield statement makes the **PlayMode Test Runner** wait until physics has begun at the next fixed update period. Finally, an assertion is made that the original Y position is not equal to the new Y position (after the physics fixed update). Since the defaults for a RigidBody are that gravity will be applied, this is a good test that physics is being applied to the new object (in other words, it should have started falling down once physics had been applied).

PlayMode testing a door animation

Having learned the basics of **PlayMode** testing in the previous recipe, now let's test something non-trivial that we might find in a game. In this recipe, we'll create a **PlayMode** test to ensure that a door opening animation plays when the player's sphere object enters a collider.

A scene has been provided with the player's sphere initialized to roll toward a red door. When the sphere hits the collider (`OnTriggerEnter` event), some code sets the door's **Animator Controller Opening** variable to true, which transitions the door from its closed state to its open state, as can be seen in the following screenshot:



Thanks should go to the creator of the ground texture; it was designed by Starline, and published at `Freepik.com`.

Getting ready

For this recipe, a Unity Package has been provided `(doorScene.unitypackage)` in the `19_06` folder.

How to do it...

To **PlayMode** test a door animation, follow these steps:

1. Create a new Unity project, and delete the default folder **Scenes**.
2. Import the Unity package provided (doorScene.unitypackage).
3. Add the following scenes – **doorScene** and **menuScene** – to the project Build (the sequence doesn't matter).
4. Ensure that the scene currently open is **menuScene**.
5. Display the **Test Runner** panel by choosing the following menu: **Window | General | Test Runner**
6. Enable playmode tests for all assemblies. Do this by displaying the drop-down menu in the top-right corner of the **Test Runner** panel, and selecting **Enable playmode tests for all assemblies** (click OK to any message concerned with restarting the editor).
7. Now restart the Unity Editor (just close the application and then reopen it with your project).

8. Ensure that the **PlayMode** button is selected in the **Test Runner** panel.
9. In the **Project** panel, select the top-level folder **Assets**.
10. In the **Test Runner** panel, click the "**Create PlayMode Test Assembly Folder**" button. A new folder, named Tests, should have been created.
11. In the **Project** panel, open the `Tests` folder. It should contain an assembly definition file `Tests.asmdef`.
12. In the **Test Runner** panel, click the "**Create Test Script in the current folder**" button. Rename this script class `DoorTest`.
13. Edit the `DoorTest.cs` script class, replacing the content with the following:

```
using System.Collections;
 using NUnit.Framework;
 using UnityEngine;
 using UnityEngine.SceneManagement;
 using UnityEngine.TestTools;

 public class DoorTest
 {
     const int BASE_LAYER = 0;
     private string initialScenePath;
     private Animator doorAnimator;
     private Scene tempTestScene;

     // name of scene being tested by this class
     private string sceneToTest = "doorScene";

     [SetUp]
     public void Setup()
     {
         // setup - load the scene
         tempTestScene = SceneManager.GetActiveScene();
     }
 }
```

14. Add the following test method to `DoorTest.cs`:

```
[UnityTest]
 public IEnumerator TestDoorAnimationStateStartsClosed()
 {
     // load scene to be tested
     yield return SceneManager.LoadSceneAsync(sceneToTest,
LoadSceneMode.Additive);
SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneT
oTest));
```

```
          // Arrange
          doorAnimator =
GameObject.FindWithTag("Door").GetComponent<Animator>();
          string expectedDoorAnimationState = "DoorClosed";

          // immediate next frame
          yield return null;

          // Act
          AnimatorClipInfo[] currentClipInfo =
doorAnimator.GetCurrentAnimatorClipInfo(BASE_LAYER);
          string doorAnimationState =
currentClipInfo[0].clip.name;

          // Assert
          Assert.AreEqual(expectedDoorAnimationState,
doorAnimationState);

          // teardown - reload original temp test scene
          SceneManager.SetActiveScene(tempTestScene);
          yield return
SceneManager.UnloadSceneAsync(sceneToTest);
      }
```

15. Add the following test method to `DoorTest.cs`:

```
     [UnityTest]
      public IEnumerator TestIsOpeningStartsFalse()
      {
          // load scene to be tested
          yield return SceneManager.LoadSceneAsync(sceneToTest,
LoadSceneMode.Additive);
SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneT
oTest));

          // Arrange
          doorAnimator =
GameObject.FindWithTag("Door").GetComponent<Animator>();

          // immediate next frame
          yield return null;

          // Act
          bool isOpening = doorAnimator.GetBool("Opening");

          // Assert
          Assert.IsFalse(isOpening);
```

```
            // teardown - reload original temp test scene
            SceneManager.SetActiveScene(tempTestScene);
            yield return
SceneManager.UnloadSceneAsync(sceneToTest);
       }
```

16. Add the following test method to `DoorTest.cs`:

```
      [UnityTest]
       public IEnumerator
TestDoorAnimationStateOpenAfterAFewSeconds()
       {
            // load scene to be tested
            yield return SceneManager.LoadSceneAsync(sceneToTest,
LoadSceneMode.Additive);
SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneT
oTest));

            // wait a few seconds
            int secondsToWait = 3;
            yield return new WaitForSeconds(secondsToWait);

            // Arrange
            doorAnimator =
GameObject.FindWithTag("Door").GetComponent<Animator>();
            string expectedDoorAnimationState = "DoorOpen";


            // Act
            AnimatorClipInfo[] currentClipInfo =
doorAnimator.GetCurrentAnimatorClipInfo(BASE_LAYER);
            string doorAnimationState =
currentClipInfo[0].clip.name;
            bool isOpening = doorAnimator.GetBool("Opening");

            // Assert
            Assert.AreEqual(expectedDoorAnimationState,
doorAnimationState);
            Assert.IsTrue(isOpening);

            // teardown - reload original temp test scene
            SceneManager.SetActiveScene(tempTestScene);
            yield return
SceneManager.UnloadSceneAsync(sceneToTest);
       }
```

17. Click **Run All**.
18. As the tests run, you will see first in the **Hierarchy**, **Game** and **Scene** panels

that a temporary scene is created, then the **doorScene** running, with the sphere rolling toward the red door.

19. You should see the results of your unit test being executed – if all tests are concluded successfully, there should be green ticks (check marks) next to each test.

How it works...

You added two scenes to the build, so they can be selected in our scripts using the **SceneManager** during **PlayMode** testing.

We opened the **menuScene** so that we can clearly see when Unity runs different scenes during our **PlayMode** testing – and we'll see the menu scene reopened after testing takes place.

There is a `SetUp()` method that is executed before each test. SetUp and TearDown methods are very useful for preparing things before each test, and resetting things back to how they were before the test took place. Unfortunately, aspects such as loading our door scene before running each test, and then reloading the menu after each test, involve waiting until the scene load process has completed. We can't place yield statements in our `SetUp()` and `TearDown()` methods, so you'll see each test has repeated scene loading at the beginning and end of each test:

```
// load scene to be tested
 yield return SceneManager.LoadSceneAsync(sceneToTest,
LoadSceneMode.Additive);
SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneToTest));

 // Arrange-Act-Assert goes here

 // teardown - reload original temp test scene
 SceneManager.SetActiveScene(tempTestScene);
 yield return SceneManager.UnloadSceneAsync(sceneToTest);
```

For each test, we wait, either for a single frame (yield `null`), or for a few seconds (yield `return new WaitForSeconds(...)`). This ensures that all objects have been created and physics is started before our test starts running. The first two tests check the initial conditions, in other words, that the door begins in the DoorClosed animation state, and that the **Animation Controller's** `isOpening` variable is false.

The final test waits for a few seconds (which is enough time for the sphere to roll up to the door and trigger the opening animation), and tests that the door is entering/has entered the **DoorOpen** animation state, and that the Animation Controller's `isOpening` variable is true.

As can be seen, there is quite a bit more to **PlayMode** testing than Unit Testing, but it means that we have a way to test actual **GameObject** interactions when features such as timers and physics are running. As this recipe demonstrates, we can also load our own scenes for **PlayMode** testing, be they special scenes created just to test interactions, or actual scenes that are to be included in our final game build.

PlayMode and Unit Testing a player health bar with events, logging, and exceptions

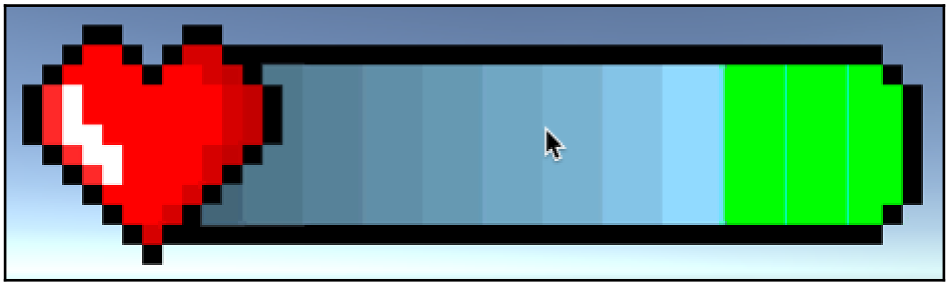
In this recipe, we combine many different kinds of tests on a feature of many games – a visual health bar representing the player's numeric health value (in this case, a float number from 0.0 - 1.0). Although far from comprehensively testing all aspects of the health bar, this recipe gives a good sample of how we can go about testing many different parts of a game using the Unity Testing tools.

A Unity Package is provided that contains the following:

- `Player.cs`: a player script class, managing values for player health, and using delegates-and-events to publish health changes to any listening **View** classes
- Two **View** classes that register to listen for player health change events:
    - `HealthBarDisplay.cs`: this updates the fillAmount for a UI image for each new player health value received
    - `HealthChangeLogger.cs`: this prints messages about the new player health value received to the Debug.Log file
- `PlayerManager.cs`: a manager script, which initializes player and HealthChangeLogger objects, and also allows the user to change the health of the player by pressing the Up and Down arrow keys (simulating healing/damage during a game)
- A scene that has 2 UI images – one is a health bar outline (red heart and a black outline), the second is the filler image – showing dark blue to light blue to green, for weak to strong health values)

This recipe allows several different kinds of testing to be demonstrated:

- **PlayMode** testing, to check that the actual `fillAmount` of the UI image displayed matches the 0.0 ... 1.0 range of the player's health
- **Unit Testing**, to check that player health starts with the correct default value, and correctly increases and decreases after calls to the `AddHealth(...)` and `ReduceHealth(...)` methods
- Unit testing, to check that health change events are published by the player object
- Unit testing, to check that expected messages are logged in the `Debug.Log`
- Unit testing, to check that argument out-of-range exceptions are thrown if negative values are passed to the player's `AddHealth(...)` or `ReduceHealth(...)` methods. This is demonstrated in the following screenshot:



Thanks to Pixel Art Maker for the health bar image:
`http://pixelartmaker.com/art/49e2498a414f221`

Getting ready

For this recipe, a Unity Package has been provided (`healthBarScene.unitypackage`) in the `19_07` folder.

How to do it...

To **PlayMode** and **Unit Test** a player health bar, follow these steps:

1. Create a new Unity project, create a new empty scene, and delete the default folder Scenes.
2. Import the Unity package provided (`healthBarScene.unitypackage`).
3. Open the **HealthBarScene** scene.

4. Add **HealthBarScene** to the project Build (menu: **File | Build Settings ...**).
5. Display the **Test Runner** panel by choosing the following menu: **Window | General | Test Runner**.
6. Enable PlayMode tests for all assemblies. Do this by displaying the drop-down menu in the top-right corner of the **Test Runner** panel, and selecting **Enable playmode tests for all assemblies** (click OK to any message concerned with restarting the editor).
7. Now restart the Unity Editor (just close the application and then reopen it with your project).
8. Ensure that the **PlayMode** button is selected in the **Test Runner** panel.
9. In the **Project** panel, select the top-level folder **Assets**.
10. In the **Test Runner** panel, click the "**Create PlayMode Test Assembly Folder**" button. A new folder, named **Tests**, should have been created.
11. Ensure that the **Assets** folder is selected in the **Project** panel. Create a new folder named **PlayModeTests** (this should now appear in the Assets folder).
12. Ensure that the **PlayModeTests** folder is selected in the **Project** panel. In the **Test Runner** panel, click the "**Create Test Script in the current folder**" button. Rename this script class `HealthBarPlayModeTests`.
13. Edit the `HealthBarPlayModeTests.cs` script class, replacing the content with the following:

```
using UnityEngine;
 using UnityEngine.UI;
 using UnityEngine.TestTools;
 using NUnit.Framework;
 using System.Collections;
 using UnityEngine.SceneManagement;


 [TestFixture]
 public class HealthBarPlayModeTests
 {
     private Scene tempTestScene;

     // name of scene being tested by this class
     private string sceneToTest = "HealthBar";

     [SetUp]
     public void Setup()
     {
         // setup - load the scene
         tempTestScene = SceneManager.GetActiveScene();
```

```
            }
        }
```

14. Add the following test in `HealthBarPlayModeTests.cs`:

```
        [UnityTest]
          public IEnumerator
    TestHealthBarImageMatchesPlayerHealth()
          {
              // load scene to be tested
              yield return SceneManager.LoadSceneAsync(sceneToTest,
    LoadSceneMode.Additive);
    SceneManager.SetActiveScene(SceneManager.GetSceneByName(sceneT
    oTest));

              // wait for one frame
              yield return null;

              // Arrange
              Image healthBarFiller = GameObject.Find("image-
    health-bar-filler").GetComponent<Image>();
              PlayerManager playerManager =
    GameObject.FindWithTag("PlayerManager").GetComponent<PlayerMan
    ager>();
              float expectedResult = 0.9f;

              // Act
              playerManager.ReduceHealth();

              // Assert
              Assert.AreEqual(expectedResult,
    healthBarFiller.fillAmount);

              // teardown - reload original temp test scene
              SceneManager.SetActiveScene(tempTestScene);
              yield return
    SceneManager.UnloadSceneAsync(sceneToTest);
          }
```

15. Click **Run All**.
16. As the tests run, you will see first in the **Hierarchy**, **Game** and **Scene** panels that a temporary scene is created, then the **HealthBarScene** running, with the visual health bar.
17. You should see the results of your **PlayMode** Test being executed – if the test is concluded successfully, there should be a green tick (check mark).
18. Ensure that the **Assets** folder is selected in the **Project** panel. Create a new

folder named **Editor** (this should now appear in the **Assets** folder).

19. Ensure that the **Editor** folder is selected in the **Project** panel. In the Test Runner panel, click the "**Create Test Script in the current folder**" button. Rename this script class `EditModeUnitTests`.
20. Edit the `EditModeUnitTests.cs` script class, replacing the content with the following:

```
using System;
 using UnityEngine.TestTools;
 using NUnit.Framework;
 using UnityEngine;

 public class EditModeUnitTests
 {

     // inner unit test classes go here

 }
```

21. Add the following class and basic tests inside the `EditModeUnitTests` class in `EditModeUnitTests.cs`:

```
public class TestCorrectValues
 {
     [Test]
     public void DefaultHealthOne()
     {
         // Arrange
         Player player = new Player();
         float expectedResult = 1;

         // Act
         float result = player.GetHealth();

         // Assert
         Assert.AreEqual(expectedResult, result);
     }

     [Test]
     public void HealthCorrectAfterReducedByPointOne()
     {
         // Arrange
         Player player = new Player();
         float expectedResult = 0.9f;

         // Act
```

```
            player.ReduceHealth(0.1f);
            float result = player.GetHealth();

            // Assert
            Assert.AreEqual(expectedResult, result);
        }

        [Test]
        public void HealthCorrectAfterReducedByHalf()
        {
            // Arrange
            Player player = new Player();
            float expectedResult = 0.5f;

            // Act
            player.ReduceHealth(0.5f);
            float result = player.GetHealth();

            // Assert
            Assert.AreEqual(expectedResult, result);
        }
    }
```

22. Add the following class and limit test inside the `EditModeUnitTests` class in `EditModeUnitTests.cs`:

```
  public class TestLimitNotExceeded
    {
        [Test]
        public void HealthNotExceedMaximumOfOne()
        {
            // Arrange
            Player player = new Player();
            float expectedResult = 1;

            // Act
            player.AddHealth(1);
            player.AddHealth(1);
            player.AddHealth(0.5f);
            player.AddHealth(0.1f);
            float result = player.GetHealth();

            // Assert
            Assert.AreEqual(expectedResult, result);
        }
    }
```

23. Add the following class and event tests inside the `EditModeUnitTests`

class in `EditModeUnitTests.cs`:

```
public class TestEvents
 {
     [Test]
     public void CheckEventFiredWhenAddHealth()
     {
         // Arrange
         Player player = new Player();
         bool eventFired = false;

         Player.OnHealthChange += delegate
         {
             eventFired = true;
         };

         // Act
         player.AddHealth(0.1f);

         // Assert
         Assert.IsTrue(eventFired);
     }

     [Test]
     public void CheckEventFiredWhenReduceHealth()
     {
         // Arrange
         Player player = new Player();
         bool eventFired = false;

         Player.OnHealthChange += delegate
         {
             eventFired = true;
         };

         // Act
         player.ReduceHealth(0.1f);

         // Assert
         Assert.IsTrue(eventFired);
     }
 }
```

24. Add the following class and exception tests inside the `EditModeUnitTests` class in `EditModeUnitTests.cs`:

```
public class TestExceptions
 {
```

```
            [Test]
            public void
    Throws_Exception_When_Add_Health_Passed_Less_Than_Zero()
            {
                // Arrange
                Player player = new Player();

                // Act

                // Assert
                Assert.Throws<ArgumentOutOfRangeException>(
                    delegate
                    {
                        player.AddHealth(-1);
                    }
                );
            }

            [Test]
            public void
    Throws_Exception_When_Reduce_Health_Passed_Less_Than_Zero()
            {
                // Arrange
                Player player = new Player();

                // Act

                // Assert
                Assert.Throws<ArgumentOutOfRangeException>(
                    () => player.ReduceHealth(-1)
                );
            }
        }
```

25. Add the following class and logging tests inside the `EditModeUnitTests` class in `EditModeUnitTests.cs`:

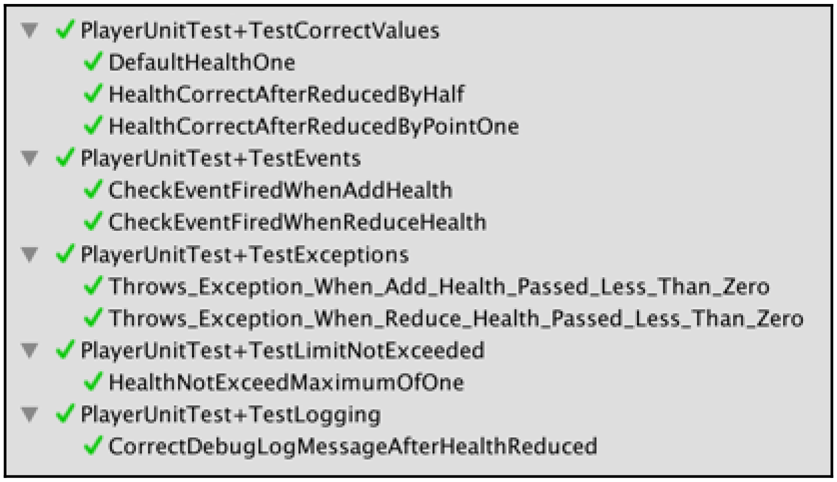
```
        public class TestLogging
         {
            [Test]
            public void
    Throws_Exception_When_Add_Health_Passed_Less_Than_Zero()
            {
                Debug.unityLogger.logEnabled = true;

                // Arrange
                Player player = new Player();
                HealthChangeLogger healthChangeLogger = new
    HealthChangeLogger();
                string expectedResult = "health = 0.9";

                // Act
                player.ReduceHealth(0.1f);

                // Assert
                LogAssert.Expect(LogType.Log, expectedResult);
            }
         }
```

You can see that the inner classes allow grouping of the unit tests visually in the Test Runner panel



```
▼ ✓ PlayerUnitTest+TestCorrectValues
     ✓ DefaultHealthOne
     ✓ HealthCorrectAfterReducedByHalf
     ✓ HealthCorrectAfterReducedByPointOne
▼ ✓ PlayerUnitTest+TestEvents
     ✓ CheckEventFiredWhenAddHealth
     ✓ CheckEventFiredWhenReduceHealth
▼ ✓ PlayerUnitTest+TestExceptions
     ✓ Throws_Exception_When_Add_Health_Passed_Less_Than_Zero
     ✓ Throws_Exception_When_Reduce_Health_Passed_Less_Than_Zero
▼ ✓ PlayerUnitTest+TestLimitNotExceeded
     ✓ HealthNotExceedMaximumOfOne
▼ ✓ PlayerUnitTest+TestLogging
     ✓ CorrectDebugLogMessageAfterHealthReduced
```

How it works...

Let's take a look at how it works in detail.

PlayMode testing

The PlayMode test `TestHealthBarImageMatchesPlayerHealth()` loads the `HealthBar` scene, gets a reference to the instance-object of **PlayerManager**, which is a component of the **GameObject** tagged **PlayerManager**, and invokes the `ReduceHealth()` method. This method reduces the player's health by 0.1, so from its starting value of 1.0, it becomes 0.9.

The **PlayerManager GameObject** also has as a component an instance object of the C# `HealthBarDisplay` script class. This object registers to listen to published events from the player class. It also has a public **UI Image** variable that has been linked to the **UI Image** of the health bar filler image in the scene.

When the player's health is reduced to 0.9, it publishes the `OnChangeHealth(0.9)` event. This event is received by the `HealthBarDisplay` object instance, which then sets the fillAmount property of the linked health bar filler image in the scene.

The `TestHealthBarImageMatchesPlayerHealth()` PlayMode test gets a reference to the object instance named image-health-bar-filler, storing this reference in the `healthBarFiller` variable. The test assertion made is that the expectedResult value of 0.9 matches that actual fillAmount property of the **UI Image** in the scene:

```
Assert.AreEqual(expectedResult, healthBarFiller.fillAmount);
```

Unit tests

There are several unit tests, grouped by placing them inside their own classes, inside the `EditModeUnitTests` script class.

- `TestCorrectValues` class:
    - `DefaultHealthOne()`: this tests that the default (initial value) of the player's health is 1
    - `HealthCorrectAfterReducedByPointOne()`: this tests that when the player's health is reduced by 0.1, it becomes 0.9
    - `HealthCorrectAfterReducedByHalf()`: this tests that when the player's health is reduced by 0.5 it becomes 0.5
- class `TestLimitNotExceeded`:
    - `HealthNotExceedMaximumOfOne()`: this tests that the value of the player's health does not exceed 1, even after attempts to add 1, 0.5, and 0.1 to its initial value of 1
- class `TestEvents`:
    - `CheckEventFiredWhenAddHealth()`: this tests that an OnChangeHealth() event is published when the player's health is increased
    - `CheckEventFiredWhenReduceHealth()`: this tests that an OnChangeHealth() event is published when the player's

health is decreased

- class `TestLogging`:
    - `CorrectDebugLogMessageAfterHealthReduced()`: this tests that a Debug.Log message is correctly logged after the player's heath is reduced by 0.1 to 0.9
- class `TestExceptions`:
    - `Throws_Exception_When_Add_Health_Passed_Less_Than_Zero()`: this tests that an ArgumentOutOfRangeException is thrown when a negative value is passed to the AddHealth(...) player method
    - `Throws_Exception_When_Reduce_Health_Passed_Less_Than_Zero()`: this tests that an ArgumentOutOfRangeException is thrown when a negative value is passed to the ReduceHealth(...) player method

These two tests illustrate one convention of naming tests that adds an underscore _ character between each word in the method name in order to improve readability.

See also

Learn more about the LogAssert Unity Script reference in the Unity documentation:

- `https://docs.unity3d.com/ScriptReference/TestTools.LogAssert.html`

The method for unit testing C# events is adapted from a post on philosophicalgeek.com:

- `http://www.philosophicalgeek.com/2007/12/27/easily-unit-testing-event-handlers/`

The delegate-event publishing of health change events in this health bar feature is an example of the Publisher-Subscriber design pattern. Learn more about design patterns and their implementations for Unity games in `Chapter 17`, Extra Features and Design Patterns.