

# 1

## Unity C# Refresher

This book is about mastering scripting for Unity, specifically mastering C# in the context of Unity game development. The concept of mastering needs a definition and qualification, before proceeding further. By mastering, I mean this book will help you transition from having intermediate and theoretical knowledge to having more fluent, practical, and advanced knowledge of scripting. Fluency is the keyword here. From the outset of learning any programming language, the focus invariably turns to language syntax and its rules and laws — the formal parts of a language. This includes concepts such as variables, loops, and functions. However, as a programmer gets experience, the focus shifts from language specifically to the creative ways in which language is applied to solve real-world problems. The focus changes from language-oriented problems to questions of context-sensitive application. Consequently, most of this book will not primarily be about the formal language syntax of C#.

After this chapter, I'll assume that you already know the basics. Instead, the book will be about case studies and real-world examples of the use of C#. However, before turning to that, this chapter will focus on the C# basics generally. This is intentional. It'll cover, quickly and in summary, all the C# foundational knowledge you'll need to follow along productively with subsequent chapters. I strongly recommend that you read it through from start to finish, whatever your experience. It's aimed primarily at readers who are reasonably new to C# but fancy jumping in at the deep end. However, it can also be valuable to experienced developers to consolidate their existing knowledge and, perhaps, pick up new advice and ideas along the way. In this chapter, then, I'll outline the fundamentals of C# from the ground up, in a step-by-step, summarized way. I will speak as though you already understand the very basics of programming generally, perhaps with another language, but have never encountered C#. So, let's go.

## Why C#?

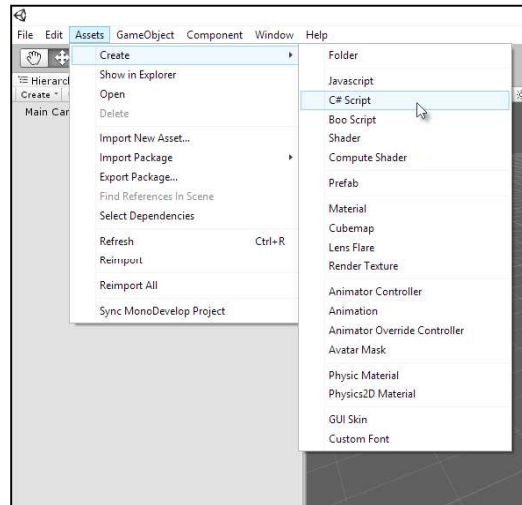
When it comes to Unity scripting, an early question when making a new game is which language to choose, because Unity offers a choice. The official choices are C# or JavaScript. However, there's a debate about whether JavaScript should more properly be named "JavaScript" or "UnityScript" due to the Unity-specific adaptations made to the language. This point is not our concern here. The question is which language should be chosen for your project. Now, it initially seems that as we have a choice, we can actually choose all two languages and write some script files in one language and other script files in another language, thus effectively mixing up the languages. This is, of course, technically possible. Unity won't stop you from doing this. However, it's a "bad" practice because it typically leads to confusion as well as compilation conflicts; it's like trying to calculate distances in miles and kilometers at the same time.

The recommended approach, instead, is to choose one of the three languages and apply it consistently across your project as the authoritative language. This is a slicker, more efficient workflow, but it means one language must be chosen at the expense of others. This book chooses C#. Why? First, it's not because C# is "better" than the others. There is no absolute "better" or "worse" in my view. Each and every language has its own merits and uses, and all the Unity languages are equally serviceable for making games. The main reason is that C# is, perhaps, the most widely used and supported Unity language, because it connects most readily to the existing knowledge that most developers already have when they approach Unity. Most Unity tutorials are written with C# in mind, as it has a strong presence in other fields of application development. C# is historically tied to the .NET framework, which is also used in Unity (known as Mono there), and C# most closely resembles C++, which generally has a strong presence in game development. Further, by learning C#, you're more likely to find that your skill set aligns with the current demand for Unity programmers in the contemporary games industry. Therefore, I've chosen C# to give this book the widest appeal and one that connects to the extensive body of external tutorials and literature. This allows you to more easily push your knowledge even further after reading this book.

## Creating script files

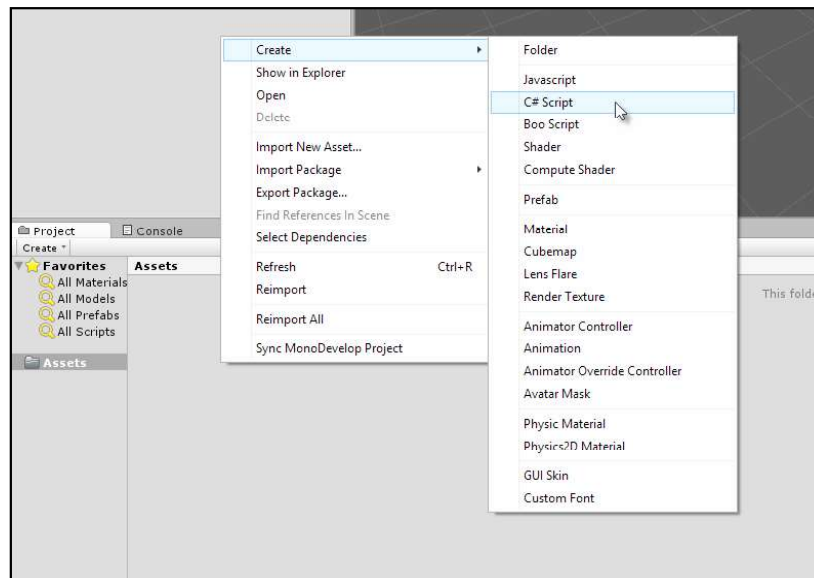
If you need to define a logic or behavior for your game, then you'll need to write a script. Scripting in Unity begins by creating a new script file, which is a standard text file added to the project. This file defines a program that lists all the instructions for Unity to follow. As mentioned, the instructions can be written in either C#, JavaScript, or Boo; for this book, the language will be C#. There are multiple ways to create a script file in Unity.

One way is to go to **Assets | Create | C# Script** from the application menu, as shown in the following screenshot:



Creating a script file via the application menu

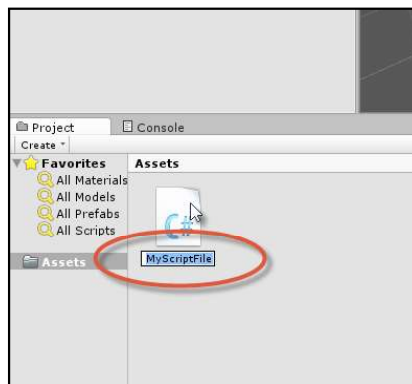
Another way is to right-click on the empty space anywhere within the **Project** panel and choose the **C# Script** option in the **Create** menu from the context menu, as shown in the following screenshot. This creates the asset in the currently open folder.



Creating a script file via the Project panel context menu

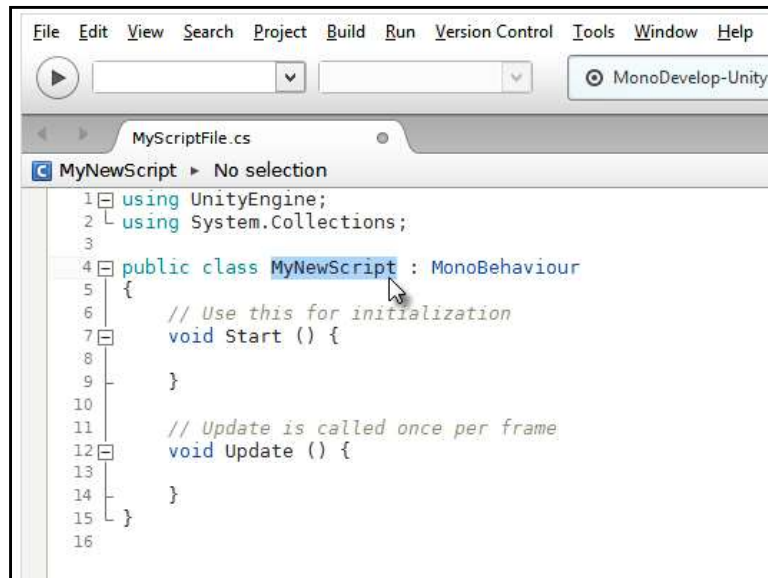
Once created, a new script file will be generated inside the `Project` folder with a `.cs` file extension (representing C Sharp). The filename is especially important and has serious implications on the validity of your script files because Unity uses the filename to determine the name of a C# class to be created inside the file. Classes are considered in more depth later in this chapter. In short, be sure to give your file a unique and meaningful name.

By unique, we mean that no other script file anywhere in your project should have the same name, whether it is located in a different folder or not. All the script files should have a unique name across the project. The name should also be meaningful by expressing clearly what your script intends to do. Further, there are rules of validity governing filenames as well as class names in C#. The formal definition of these rules can be found online at <http://msdn.microsoft.com/en-us/library/aa664670%28VS.71%29.aspx>. In short, the filename should start with a letter or underscore character only (numbers are not permitted for the first character), and the name should include no spaces, although underscores (`_`) are allowed:



Name your script files in a unique way and according to the C# class naming conventions

Unity script files can be opened and examined in any text editor or IDE, including Visual Studio and Notepad++, but Unity provides the free and open source editor, **MonoDevelop**. This software is part of the main Unity package included in the installation and doesn't need to be downloaded separately. By double-clicking on the script file from the **Project** panel, Unity will automatically open the file inside MonoDevelop. If you later decide to, or need to, rename the script file, you also need to rename the C# class inside the file to match the filename exactly, as shown in the following screenshot. Failure to do so will result in invalid code and compilation errors or problems when attaching the script file to your objects.



Renaming classes to match the renamed script files

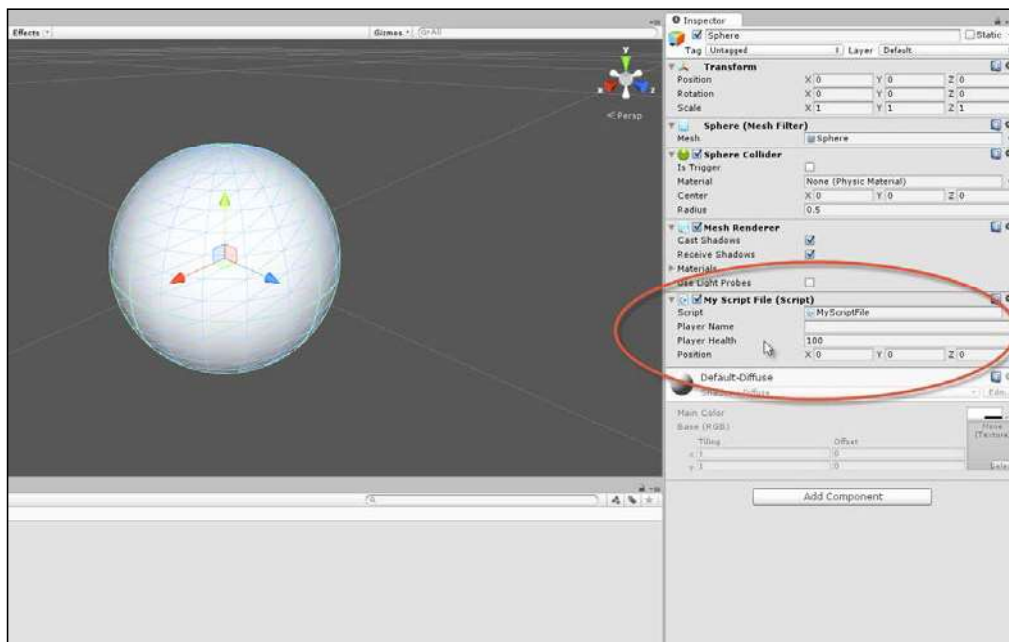
### Compiling code

To compile code in Unity, you just need to save your script file in MonoDevelop by choosing the **Save** option in the **File** menu from the application menu (or by pressing *Ctrl + S* on the keyboard) and then return to the main Unity Editor. On refocusing on the Unity window, Unity automatically detects code changes in the files and then compiles your code in response. If there are errors, the game cannot be run, and the errors are printed to the **Console** window. If the compile was successful, you don't need to do anything else, except press **Play** on the **Editor** toolbar and test run your game. Take care here; if you forget to save your file in MonoDevelop after making code changes, then Unity will still use the older, compiled version of your code. For this reason as well as for the purpose of backup, it's really important to save your work regularly, so be sure to press *Ctrl + S* to save in MonoDevelop.



## Instantiating scripts

Each script file in Unity defines one main class that is like a blueprint or design that can be instantiated. It is a collection of related variables, functions, and events (as we'll see soon). By default, a script file is like any other kind of Unity asset, such as meshes and audio files. Specifically, it remains dormant in the **Project** folder and does nothing until it's added to a specific scene (by being added to an object as a component), where it comes alive at runtime. Now, scripts, being logical and mathematical in nature, are not added to the scene as tangible, independent objects as meshes are. You cannot see or hear them directly, because they have no visible or audible presence. Instead, they're added onto existing game objects as components, where they define the behavior of those objects. This process of bringing scripts to life as a specific component on a specific object is known as instantiation. Of course, a single script file can be instantiated on multiple objects to replicate the behavior for them all, saving us from making multiple script files for each object, such as when multiple enemy characters must use the same artificial intelligence. The point of the script file, ideally, is to define an abstract formula or behavior pattern for an object that can be reused successfully across many similar objects in all possible scenarios. To add a script file onto an object, simply drag-and-drop the script from the **Project** panel onto the destination object in the scene. The script will be instantiated as a component, and its public variables will be visible in the **Object Inspector** whenever the object is selected, as shown in the following screenshot:



Attaching scripts onto game objects as components

Variables are considered in more depth in the next section.



More information on creating and using scripts in Unity can be found online at <http://docs.unity3d.com/412/Documentation/Manual/Scripting.html>.

## Variables

Perhaps, the core concept in programming and in C# is the variable. Variables often correspond to the letters used in algebra and stand in for numerical quantities, such as  $X$ ,  $Y$ , and  $Z$  and  $a$ ,  $b$ , and  $c$ . If you need to keep track of information, such as the player name, score, position, orientation, ammo, health, and a multitude of other types of quantifiable data (expressed by nouns), then a variable will be your friend. A variable represents a single unit of information. This means that multiple variables are needed to hold multiple units, one variable for each. Further, each unit will be of a specific type or kind. For example, the player's name represents a sequence of letters, such as "John", "Tom", and "David". In contrast, the player's health refers to numerical data, such as 100 percent (1) or 50 percent (0.5), depending on whether the player has sustained damage. So, each variable necessarily has a data type. In C#, variables are created using a specific kind of syntax or grammar. Consider the following code sample 1-1 that defines a new script file and class called `MyNewScript`, which declares three different variables with class scope, each of a unique type. The word "declare" means that we, as programmers, are telling the C# compiler about the variables required:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyNewScript : MonoBehaviour
05 {
06     public string PlayerName = "";
07     public int PlayerHealth = 100;
08     public Vector3 Position = Vector3.zero;
09
10     // Use this for initialization
11     void Start () {
12
13     }
14
15     // Update is called once per frame
16     void Update () {
17
18     }
19 }
```

### Variable data types

Each variable has a data type. A few of the most common ones include `int`, `float`, `bool`, `string`, and `Vector3`. Here, are a few examples of these types:



- `int` (integer or whole number) = -3, -2, -1, 0, 1, 2, 3...
- `float` (floating point number or decimal) = -3.0, -2.5, 0.0, 1.7, 3.9...
- `bool` (Boolean or true/false) = `true` or `false` (1 or 0)
- `string` (string of characters) = "hello world", "a", "another word..."
- `Vector3` (a position value) = (0, 0, 0), (10, 5, 0)...

Notice from lines 06-08 of code sample 1-1 that each variable is assigned a starting value, and its data type is explicitly stated as `int` (integer), `string`, and `Vector3`, which represent the points in a 3D space (as well as directions, as we'll see). There's no full list of possible data types, as this will vary extensively, depending on your project (and you'll also create your own!). Throughout this book, we'll work with the most common types, so you'll see plenty of examples. Finally, each variable declaration line begins with the keyword `public`. Usually, variables can be either `public` or `private` (and there is another one called `protected`, which is not covered here). The `public` variables will be accessible and editable in Unity's Object Inspector (as we'll see soon, you can also refer to the preceding screenshot), and they can also be accessed by other classes.

Variables are so named because their values might vary (or change) over time. Of course, they don't change in arbitrary and unpredictable ways. Rather, they change whenever we explicitly change them, either through direct assignment in code, from the Object Inspector, or through methods and function calls. They can be changed both directly and indirectly. Variables can be assigned values directly, such as the following one:

```
PlayerName = "NewName";
```

They can also be assigned indirectly using expressions, that is, statements whose final value must be evaluated before the assignment can be finally made to the variable as follows:

```
//Variable will result to 50, because: 100 x 0.5 = 50  
PlayerHealth = 100 * 0.5;
```



**Variable scope**

Each variable is declared with an implicit scope. The scope determines the lifetime of a variable, that is, the places inside a source file where a variable can be successfully referenced and accessed. Scope is determined by the place where the variable is declared. The variables declared in code sample 1-1 have class scope, because they are declared at the top of a class and outside any functions. This means they can be accessed everywhere throughout the class, and (being public) they can also be accessed from other classes. Variables can also be declared inside specific functions. These are known as local variables, because their scope is restricted to the function, that is, a local variable cannot be accessed outside the function in which it was declared. Classes and functions are considered later in this chapter.

More information on variables and their usage in C# can be found at <http://msdn.microsoft.com/en-us/library/aa691160%28v=vs.71%29.aspx>.

## Conditional statements

Variables change in potentially many different circumstances: when the player changes their position, when enemies are destroyed, when the level changes, and so on. Consequently, you'll frequently need to check the value of a variable to branch the execution of your scripts that perform different sets of actions, depending on the value. For example, if `PlayerHealth` reaches 0 percent, you'll perform a death sequence, but if `PlayerHealth` is at 20 percent, you might only display a warning message. In this specific example, the `PlayerHealth` variable drives the script in a specified direction. C# offers two main conditional statements to achieve a program branching like this. These are the `if` statement and the `Switch` statement. Both are highly useful.

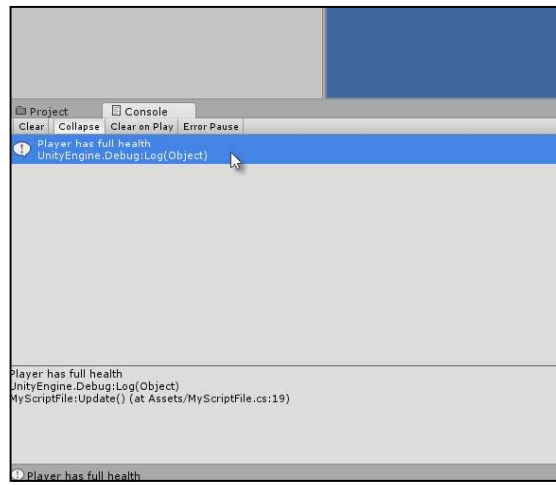
### The if statement

The `if` statement has various forms. The most basic form checks for a condition and will perform a subsequent block of code if, and only if, that condition is `true`. Consider the following code sample 1-2:

```
01 using UnityEngine;
02 using System.Collections;
03
```

```
04 public class MyScriptFile : MonoBehaviour
05 {
06     public string PlayerName = "";
07     public int PlayerHealth = 100;
08     public Vector3 Position = Vector3.zero;
09
10     // Use this for initialization
11     void Start () {
12     }
13
14     // Update is called once per frame
15     void Update ()
16     {
17         //Check player health - the braces symbol {} are option
           for one-line if-statements
18         if(PlayerHealth == 100)
19         {
20             Debug.Log ("Player has full health");
21         }
22     }
23 }
```

The preceding code is executed like all other types of code in Unity, by pressing the **Play** button from the toolbar, as long as the script file has previously been instantiated on an object in the active scene. The `if` statement at line 18 continually checks the `PlayerHealth` class variable for its current value. If the `PlayerHealth` variable is exactly equal to (`==`) 100, then the code inside the `{ }` braces (in lines 19–21) will be executed. This works because all conditional checks result in a Boolean value of either `true` or `false`; the conditional statement is really checked to see whether the queried condition (`PlayerHealth == 100`) is `true`. The code inside the braces can, in theory, span across multiple lines and expressions. However, here, there is just a single line in line 20: the `Debug.Log` Unity function outputs the **Player has full health** string to the console, as shown in the following screenshot. Of course, the `if` statement could potentially have gone the other way, that is, if `PlayerHealth` was not equal to 100 (perhaps, it was 99 or 101), then no message would be printed. Its execution always depends on the previous `if` statement evaluating to `true`.



The Unity Console is useful for printing and viewing debug messages

More information on the `if` statements, the `if-else` statement, and their usage in C# can be found online at <http://msdn.microsoft.com/en-GB/library/5011f09h.aspx>.

### Unity Console



As you can see in the preceding screenshot, the console is a debugging tool in Unity. It's a place where messages can be printed from the code using the `Debug.Log` statement (or the `Print` function) to be viewed by developers. They are helpful to diagnose issues at runtime and compile time. If you get a compile time or runtime error, it should be listed in the **Console** tab. The **Console** tab should be visible in the Unity Editor by default, but it can be displayed manually by selecting **Console** in the **Window** menu from the Unity application file menu. More information on the `Debug.Log` function can be found at <http://docs.unity3d.com/ScriptReference/Debug.Log.html>.

You can, of course, check for more conditions than just equality (`==`), as we did in code sample 1-2. You can use the `>` and `<` operators to check whether a variable is greater than or less than another value, respectively. You can also use the `!=` operator to check whether a variable is not equal to another value. Further, you can even combine multiple conditional checks into the same `if` statement using the `&&` (AND) operator and the `||` (OR) operator. For example, check out the following `if` statement. It performs the code block between the `{ }` braces only if the `PlayerHealth` variable is between 0 and 100 and is not equal to 50, as shown here:

```
if(PlayerHealth >= 0 && PlayerHealth <= 100 && PlayerHealth !=50)
{
```

```
Debug.Log ("Player has full health");
}
```

### The if-else statement

One variation of the `if` statement is the `if-else` statement. The `if` statement performs a code block if its condition evaluates to `true`. However, the `if-else` statement extends this. It would perform an X code block if its condition is `true` and a Y code block if its condition is `false`:



```
if (MyCondition)
{
    //X - perform my code if MyCondition is true
}
else
{
    //Y - perform my code if MyCondition is false
}
```

## The switch statement

As we've seen, the `if` statement is useful to determine whether a single and specific condition is `true` or `false` and to perform a specific code block on the basis of this. The `switch` statement, in contrast, lets you check a variable for multiple possible conditions or states, and then lets you branch the program in one of many possible directions, not just one or two as is the case with `if` statements. For example, if you're creating an enemy character that can be in one of the many possible states of action (`CHASE`, `FLEE`, `FIGHT`, `HIDE`, and so on), you'll probably need to branch your code appropriately to handle each state specifically. The `break` keyword is used to exit from a state returning to the end of the `switch` statement. The following code sample 1-3 handles a sample enemy using enumerations:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06     //Define possible states for enemy using an enum
07     public enum EnemyState {CHASE, FLEE, FIGHT, HIDE};
08
```

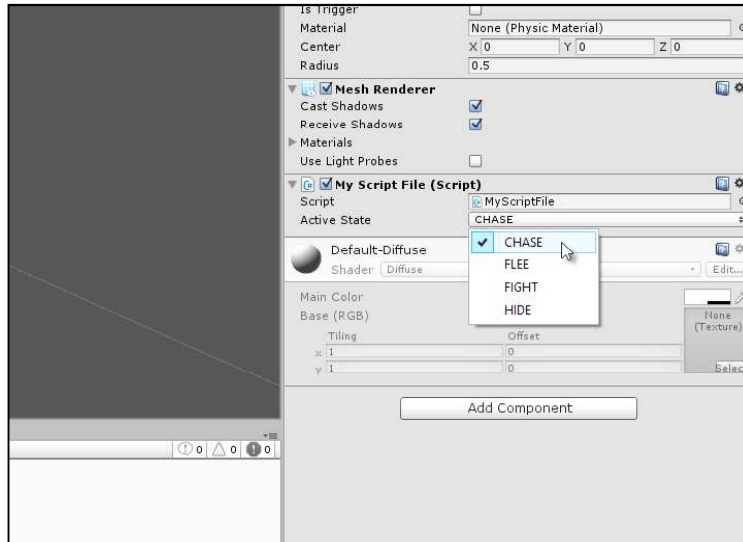
```
09    //The current state of enemy
10    public EnemyState ActiveState = EnemyState.CHASE;
11
12    // Use this for initialization
13    void Start () {
14    }
15
16    // Update is called once per frame
17    void Update ()
18    {
19        //Check the ActiveState variable
20        switch(ActiveState)
21        {
22            case EnemyState.FIGHT:
23            {
24                //Perform fight code here
25                Debug.Log ("Entered fight state");
26            }
27                break;
28
29
30            case EnemyState.FLEE:
31            case EnemyState.HIDE:
32            {
33                //Flee and hide performs the same behaviour
34                Debug.Log ("Entered flee or hide state");
35            }
36                break;
37
38            default:
39            {
40                //Default case when all other states fail
41                //This is used for the chase state
42                Debug.Log ("Entered chase state");
43            }
44                break;
45            }
46    }
47 }
```

### Enumerations



This line 07 in code sample 1-3 declares an enumeration (enum) named `EnemyState`. An enum is a special structure used to store a range of potential values for one or more other variables. It's not a variable itself per se, but a way of specifying the limits of values that a variable might have. In code sample 1-3, the `ActiveState` variable declared in line 10 makes use of `EnemyState`. Its value can be any valid value from the `ActiveState` enumeration. Enums are a great way of helping you validate your variables, limiting their values within a specific range and series of options.

Another great benefit of enums is that variables based on them have their values appear as selectable options from drop-down boxes in the Object Inspector, as shown in the following screenshot:



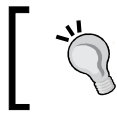
Enumerations offer you drop-down options for your variables from the Object Inspector

More information on enums and their usage in C# can be found online at <http://msdn.microsoft.com/en-us/library/sbdt4032.aspx>.

The following are the comments for code sample 1-3:

- **Line 20:** The `switch` statement begins. Parentheses, `()`, are used to select the variable whose value or state must be checked. In this case, the `ActiveState` variable is being queried.
- **Line 22:** The first case statement is made inside the `switch` statement. The following block of code (lines 24 and 25) will be executed if the `ActiveState` variable is set to `EnemyState.Fight`. Otherwise, the code will be ignored.

- **Lines 30 and 31:** Here, two case statements follow one another. The code block in lines 33 and 34 will be executed if, and only if, `ActiveState` is either `EnemyState.Flee` or `EnemyState.Hide`.
- **Line 38:** The default statement is optional for a `switch` statement. When included, it will be entered if no other case statements are `true`. In this case, it would apply if `ActiveState` is `EnemyState.Chase`.
- **Lines 27, 36, and 44:** The `break` statement should occur at the end of a case statement. When it is reached, it will exit the complete `switch` statement to which it belongs, resuming program execution in the line after the `switch` statement, in this case, line 45.



More information on the `switch` statement and its usage in C# can be found at <http://msdn.microsoft.com/en-GB/library/06tc147t.aspx>.

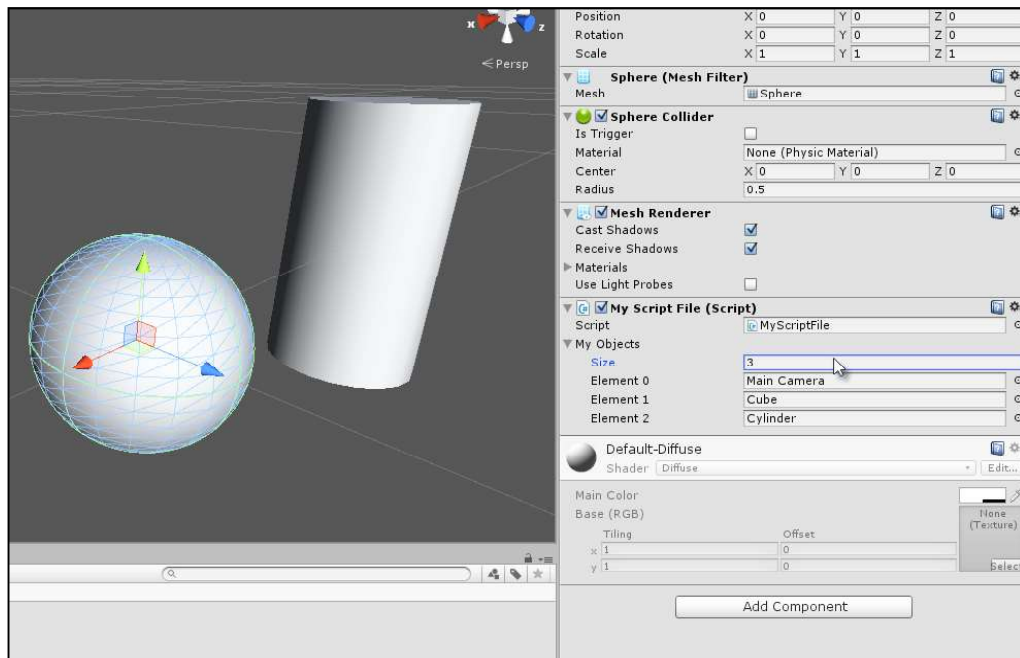
## Arrays

Lists and sequences are everywhere in games. For this reason, you'll frequently need to keep track of lists of data of the same type: all enemies in the level, all weapons that have been collected, all power ups that could be collected, all spells and items in the inventory, and so on. One type of list is the array. Each item in the array is, essentially, a unit of information that has the potential to change during gameplay, and so a variable is suitable to store each item. However, it's useful to collect together all the related variables (all enemies, all weapons, and so on) into a single, linear, and traversable list structure. This is what an array achieves. In C#, there are two kinds of arrays: static and dynamic. Static arrays might hold a fixed and maximum number of possible entries in memory, decided in advance, and this capacity remains unchanged throughout program execution, even if you only need to store fewer items than the capacity. This means some slots or entries could be wasted. Dynamic arrays might grow and shrink in capacity, on demand, to accommodate exactly the number of items required. Static arrays typically perform better and faster, but dynamic arrays feel cleaner and avoid memory wastage. This chapter considers only static arrays, and dynamic arrays are considered later, as shown in the following code sample 1-4:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06     //Array of game objects in the scene
07     public GameObject[] MyObjects;
08
```

```
09      // Use this for initialization
10      void Start ()
11      {
12      }
13
14      // Update is called once per frame
15      void Update ()
16      {
17      }
18 }
```

In code sample 1-4, line 07 declares a completely empty array of `GameObjects`, named `MyObjects`. To create this, it uses the `[]` syntax after the data type `GameObject` to designate an array, that is, to signify that a list of `GameObjects` is being declared as opposed to a single `GameObject`. Here, the declared array will be a list of all objects in the scene. It begins empty, but you can use the Object Inspector in the Unity Editor to build the array manually by setting its maximum capacity and populating it with any objects you need. To do this, select the object to which the script is attached in the scene and type in a **Size** value for the **My Objects** field to specify the capacity of the array. This should be the total number of objects you want to hold. Then, simply drag-and-drop objects individually from the scene hierarchy panel into the array slots in the Object Inspector to populate the list with items, as shown here:



Building arrays from the Unity Object Inspector



You can also build the array manually in code via the `Start` function instead of using the Object Inspector. This ensures that the array is constructed as the level begins. Either method works fine, as shown in the following code sample 1-5:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06     //Array of game objects in the scene
07     public GameObject[] MyObjects;
08
09     // Use this for initialization
10     void Start ()
11     {
12         //Build the array manually in code
13         MyObjects = new GameObject[3];
14         //Scene must have a camera tagged as MainCamera
15         MyObjects[0] = Camera.main.gameObject;
16
17         //Use GameObject.Find function to
18         //find objects in scene by name
19         MyObjects[1] = GameObject.Find("Cube");
20         MyObjects[2] = GameObject.Find("Cylinder");
21     }
22
23     // Update is called once per frame
24     void Update ()
25     {
26     }
```

The following are the comments for code sample 1-5:

- **Line 10:** The `Start` function is executed at level startup. Functions are considered in more depth later in this chapter.
- **Line 13:** The `new` keyword is used to create a new array with a capacity of three. This means that the list can hold no more than three elements at any one time. By default, all elements are set to the starting value of `null` (meaning nothing). They are empty.

- **Line 15:** Here, the first element in the array is set to the main camera object in the scene. Two important points should be noted here. First, elements in the array can be accessed using the array subscript operator `[]`. Thus, the first element of `MyObjects` can be accessed with `MyObjects[0]`. Second, C# arrays are "zero indexed". This means the first element is always at position 0, the next is at 1, the next at 2, and so on. For the `MyObjects` three-element array, each element can be accessed with `MyObjects[0]`, `MyObjects[1]`, and `MyObjects[2]`. Notice that the last element is 2 and not 3.
- **Lines 18 and 19:** Elements 1 and 2 of the `MyObjects` array are populated with objects using the function `GameObject.Find`. This searches the active scene for game objects with a specified name (case sensitive), inserting a reference to them at the specified element in the `MyObjects` array. If no object of a matching name is found, then `null` is inserted instead.



More information on arrays and their usage in C# can be found online at <http://msdn.microsoft.com/en-GB/library/9b9dty7d.aspx>.

## Loops

Loops are one of the most powerful tools in programming. Imagine a game where the entire level can be nuked. When this happens, you'll want to destroy almost everything in the scene. Now, you can do this by deleting each and every object individually in code, one line at a time. If you did this, then a small scene with only a few objects would take just a few lines of code, and this wouldn't be problematic. However, for larger scenes with potentially hundreds of objects, you'd have to write a lot of code, and this code would need to be changed if you altered the contents of the scene. This would be tedious. Loops can simplify the process to just a few lines, regardless of scene complexity or object number. They allow you to repeatedly perform operations on potentially many objects. There are several kinds of loops in C#. Let's see some examples.

## The foreach loop

Perhaps, the simplest loop type in C# is the `foreach` loop. Using `foreach`, you can cycle through every element in an array, sequentially from start to end, processing each item as required. Consider the following code sample 1-6; it destroys all `GameObjects` from a `GameObject` array:

```
01 using UnityEngine;
```

```

02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06     //Array of game objects in the scene
07     public GameObject[] MyObjects;
08
09     // Use this for initialization
10     void Start ()
11     {
12         //Repeat code for all objects in array, one by one
13         foreach(GameObject Obj in MyObjects)
14         {
15             //Destroy object
16             Destroy (Obj);
17         }
18     }
19
20     // Update is called once per frame
21     void Update ()
22     {
23     }
24 }

```

#### Downloading the example code



You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

The `foreach` loop repeats the code block `{ }` between lines 14–17, once for each element in the array `MyObjects`. Each pass or cycle in the loop is known as an iteration. The loop depends on array size; this means that larger arrays require more iterations and more processing time. The loop also features a local variable `obj`. This is declared in the `foreach` statement in line 13. This variable stands in for the selected or active element in the array as the loop passes each iteration, so `obj` represents the first element in the loop on the first iteration, the second element on the second iteration, and so on.



More information on the `foreach` loop and its usage in C# can be found at <http://msdn.microsoft.com/en-GB/library/ttw7t8t6.aspx>.

## The for loop

The `foreach` loop is handy when you need to iterate through a single array sequentially from start to end, processing each element one at a time. But sometimes you need more control over the iterations. You might need to process a loop backwards from the end to the start, you might need to process two arrays of equal length simultaneously, or you might need to process every alternate array element as opposed to every element. You can achieve this using the `for` loop, as shown here:

```
//Repeat code backwards for all objects in array, one by one
for(int i = MyObjects.Length-1; i >= 0; i--)
{
    //Destroy object
    DestroyMyObjects[i];
}
```

The following are the comments for the preceding code snippet:

- Here, the `for` loop traverses the `MyObjects` array backwards from the end to the start, deleting each `GameObject` in the scene. It does this using a local variable `i`. This is sometimes known as an `Iterator` variable, because it controls how the loop progresses.
- The `for` loop line has the following three main parts, each separated by a semicolon character:
  - `i`: This is initialized to `MyObjects.Length - 1` (the last element in the array). Remember that arrays are zero-indexed, so the last element is always `Array.Length - 1`. This ensures that loop iteration begins at the array end.
  - `i >= 0`: This expression indicates the condition when the loop should terminate. The `i` variable acts like a countdown variable, decrementing backwards through the array. In this case, the loop should end when `i` is no longer greater than or equal to 0, because 0 represents the start of the array.
  - `i--`: This expression controls how the variable `i` changes on each iteration of the loop moving from the array end to the beginning. Here, `i` will be decremented by one on each iteration, that is, a value of 1 will be subtracted from `i` on each pass of the loop. In contrast, the statement `++` will add 1.
- During the loop, the expression `MyObjects[i]` is used to access array elements.



More information on the for loop and its usage in C# can be found at <http://msdn.microsoft.com/en-gb/library/ch45axte.aspx>.

## The while loop

Both the `for` and `foreach` loops were especially useful when cycling through an array, performing specific operations on each iteration. The `while` loop, in contrast, is useful to continually repeat a specific behavior until a specified condition evaluates to `false`. For example, if you must deal damage to the player as long as they're standing on hot lava or continually move a vehicle until the breaks are applied, then a `while` loop could be just what you need, as shown in the following code sample 1-7:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06     // Use this for initialization
07     void Start ()
08     {
09         //Will count how many messages have been printed
10         int NumberOfMessages = 0;
11
12         //Loop until 5 messages have been printed to the console
13         while(NumberOfMessages < 5)
14         {
15             //Print message
16
17             Debug.Log ("This is Message: " +
18                 NumberOfMessages.ToString());
19
20             //Increment counter
21             ++NumberOfMessages;
22         }
23
24     // Update is called once per frame
25     void Update ()
26     {
27     }
```



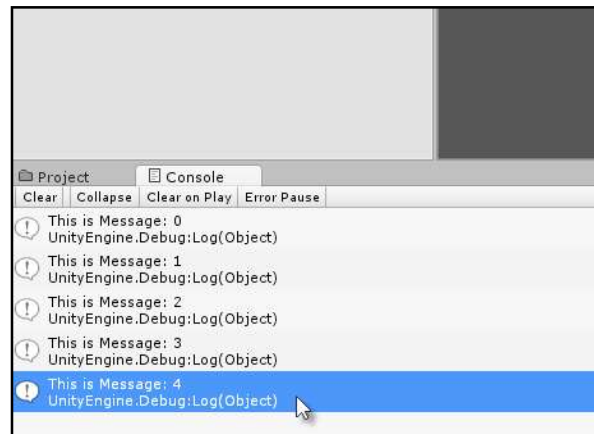
### ToString

Many classes and objects in Unity have a `ToString` function (see line 16 of code sample 1-7). This function converts the object, such as an integer (whole number), to a human-readable word or statement that can be printed to the **Console** or **Debugging** window. This is useful for printing objects and data to the console when debugging. Note that converting numerical objects to strings requires an implicit conversion.

The following are the comments for code sample 1-7:

- Line 13 begins the `while` loop with the condition that it repeats until the integer variable `NumberOfMessages` exceeds or equals 5
- The code block between lines 15 and 19 is repeated as the body of the `while` loop
- Line 19 increments the variable `NumberOfMessages` on each iteration

The result of code sample 1-7, when executed in the game mode, will be to print five text messages to the Unity Console when the level begins, as shown in the following screenshot:



Printing messages to Console in a while loop



More information on the `while` loop and its usage in C# can be found at <http://msdn.microsoft.com/en-gb/library/2aeyhxcd.aspx>.

## Infinite loops

One danger of using loops, especially `while` loops, is to accidentally create an infinite loop, that is, a loop that cannot end. If your game enters an infinite loop, it will normally freeze, perhaps permanently, requiring you to force a quit by terminating the application or even worse, causing a complete system crash! Often, Unity will catch the problem and exit but don't rely on this. For example, removing line 19 of the code sample 1-7 would create an infinite loop because the `NumberOfMessages` variable will never increment to a level that satisfies the `while` loop condition, thereby causing an exit. The message of this section, then, is first and foremost, "Take care when writing and planning loops to avoid infinite loops." The following is another classic example of an infinite loop that will certainly cause problems for your game, so be sure to avoid them:

```
//Loop forever
while(true)
{
}
```

However, believe it or not, there are times when infinite loops are technically what you need for your game under the right conditions! If you need a moving platform to travel up and down endlessly, a magical orb to continually spin round and round, or a day-night cycle to perpetually repeat, then an infinite loop can be serviceable, provided it's implemented appropriately. Later in this book, we'll see examples where infinite loops can be put to good use. Loops are powerful, fun structures, but when coded inappropriately, whether infinite or not, they can be the source of crashes, stalls, and performance issues, so take care. In this book, we'll see good practices for creating loops.

## Functions

We already used functions in this chapter, such as the `Start` and `Update` functions. However, now, it's time to consider them more formally and precisely. In essence, a function is a collection of statements bundled together as a single, identifiable block, which is given a collective name and can be executed on demand, each line of the function being executed in sequence. When you think about the logic of your game, there are times when you need to perform some operations repeatedly on your objects, such as, firing a weapon, jumping in the air, killing enemies, updating the score, and playing a sound. You can copy and paste your code throughout the source file, wherever you need to reuse it; this is not a good habit to cultivate. It's easier to consolidate the recyclable code into a function that can be executed by a name when you need it, as shown in the following code sample 1-8:

```
01 using UnityEngine;
```

```
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06     //Private variable for score
07     //Accessible only within this class
08     private int Score = 0;
09
10     // Use this for initialization
11     void Start ()
12     {
13         //Call update score
14         UpdateScore(5, false); //Add five points
15         UpdateScore (10, false); //Add ten points
16
17         int CurrentScore = UpdateScore (15, false); //Add fifteen
            points and store result
18
19         //Now double score
20         UpdateScore(CurrentScore);
21     }
22     // Update is called once per frame
23     void Update ()
24     {
25     }
26
27     //Update game score
28
29     public int UpdateScore (int AmountToAdd, bool
        PrintToConsole = true)
30     {
31         //Add points to score
32         Score += AmountToAdd;
33
34         //Should we print to console?
35
36         if(PrintToConsole){Debug.Log ("Score is: " +
            Score.ToString());}
```



---

```
36      //Output current score and exit function
37      return Score;
38  }
39 }
```

The following is the breakdown of the code present for code sample 1-8:

- **Line 08:** A private, integer class variable `Score` is declared to keep track of a sample score value. This variable will be used later in the function `UpdateScore`.
- **Lines 11, 23, and 28:** The class `MyScriptFile` has three functions (sometimes called methods or member functions). These are `Start`, `Update`, and `UpdateScore`. `Start` and `Update` are special functions that Unity provides, as we'll see shortly. `UpdateScore` is a custom function for `MyScriptFile`.
- **Line 28:** The `UpdateScore` function represents a complete block of code between lines 29 and 38. This specific function should be invoked every time the game score must change. When called, the code block (lines 29–38) will be executed sequentially. In this way, functions offer us code recyclability.
- **Lines 14-19:** The `UpdateScore` function is called several times during the `Start` function. For each call, the execution of the `Start` function pauses until the `UpdateScore` function completes. At this point, the execution resumes in the next line.
- **Line 28:** `UpdateScore` accepts two parameters or arguments. These are an integer `AmountToAdd` and a Boolean `PrintToConsole`. Arguments act like inputs we can plug in to the function to affect how they operate. The `AmountToAdd` variable expresses how much should be added to the current `Score` variable, and `PrintToConsole` determines whether the `Score` variable should be shown in the **Console** window when the function is executed. There is theoretically no limit to the number of arguments a function can have, and a function can also have no arguments at all, such as the `Start` and `Update` functions.
- **Lines 31-34:** Here, the score is actually updated and printed to **Console**, if required. Notice that the `PrintToConsole` argument has a default value of `true` already assigned to the function declaration in line 28. This makes the argument optional whenever the function is called. Lines 14, 15, and 16 explicitly override the default value by passing a value of `false`. Line 19, in contrast, omits a second value and thereby accepts the default of `true`.

- **Lines 28 and 37:** The `UpdateScore` function has a return value, which is a data type specified in line 28 before the function name. Here, the value is an `int`. This means on exiting or completion, the function will output an integer. The integer, in this case, will be the current `Score`. This is actually output in line 37 using the `return` statement. Functions don't have to return a value, it's not essential. If no return value is needed, the return type should be `void` as with `Start` and `Update`.



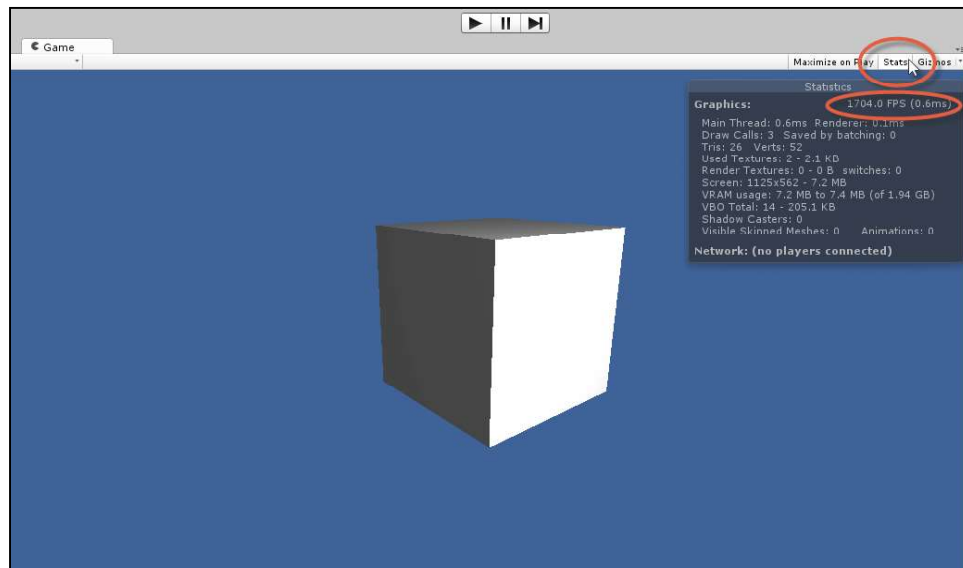
More information on functions and their usage in C# can be found at <http://csharp.net-tutorials.com/basics/functions/>.

## Events

Events are essentially functions used in a distinctive way. Both the `Start` and `Update` functions, which we have already seen, would more accurately be described as Unity-specific events. Events are functions called to notify an object that something significant has happened: the level has begun, a new frame has started, an enemy has died, the player has jumped, and others. In being called at these critical times, they offer objects the chance to respond if necessary. The `Start` function is called automatically by Unity when the object is first created, typically at level startup. The `Update` function is also called automatically, once on each frame. The `Start` function, therefore, gives us an opportunity to perform specific actions when the level begins, and the `Update` function on each frame many times per second. The `Update` function is especially useful, therefore, to achieve motion and animation in your games. Refer to code sample 1-9, which rotates an object over time:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06     // Use this for initialization
07     void Start ()
08     {
09     }
10
11     // Update is called once per frame
12     void Update ()
13     {
14         //Rotate object by 2 degrees per frame around the Y axis
15         transform.Rotate(new Vector3(0.0f, 2.0f, 0.0f));
16     }
17 }
```

Line 15 in code sample 1-9 is called once per frame. It continually rotates an object 2 degrees around the *y* axis. This code is frame rate dependent, which means that it'll turn objects faster when run on machines with higher frame rates, because `Update` will be called more often. There are techniques to achieve frame rate independence, ensuring that your games perform consistently across all machines, regardless of the frame rate. We'll see these in the next chapter. You can easily check the frame rate for your game directly from the Unity Editor **Game** tab. Select the **Game** tab and click on the **Stats** button in the top-right hand corner of the toolbar. This will show the **Stats** panel, offering a general, statistical overview of the performance of your game. This panels displays the game **frames per second (FPS)**, which indicates both how often `Update` is called on your objects and the general performance of your game on your system. In general, an FPS lower than 15 indicates a significant performance problem. Strive for FPS rates of 30 or above. Refer to the following screenshot to access the **Stats** panel:



Accessing the Stats panel for the Game tab to view FPS



There are too many event types to list comprehensively. However, some common events in Unity, such as `Start` and `Update`, can be found in the `MonoBehaviour` class. More information on `MonoBehaviour` is available at <http://docs.unity3d.com/ScriptReference/MonoBehaviour.html>.

## Classes and object-oriented programming

A class is an amalgam of many related variables and functions, all brought together into a self-contained unit or "thing". To put it another way, if you think about a game (such as a fantasy RPG), it's filled with many independent things such as wizards, orcs, trees, houses, the player, quests, inventory items, weapons, spells, doorways, bridges, force fields, portals, guards, and so on. Many of these objects parallel objects in the real world too. However, crucially, each of these things is an independent object; a wizard is different and separate from a force field, and a guard is different and separate from a tree. Each of these things, then, can be thought of as an object—a custom type. If we focus our attention on one specific object, an orc enemy, for example, we can identify the properties and behaviors in this object. The orc will have a position, rotation, and scale; these correspond to variables.

The orc might have several kinds of attacks too, such as a melee attack with an axe and a ranged attack with a crossbow. These attacks are performed through functions. In this way, a collection of variables and functions are brought together into a meaningful relationship. The process of bringing these things together is known as encapsulation. In this example, an orc has been encapsulated into a class. The class, in this case, represents the template for a general, abstract orc (the concept of an orc). Objects, in contrast, are particular, concrete instantiations of the `Orc` class in the level. In Unity, script files define a class. To instantiate the class as an object in the level, add it to `GameObject`. As we've seen, classes are attached to game objects as components. Components are objects, and multiple components together form a `GameObject`. Refer to code sample 1-10 for a sample `Orc` class stub:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Orc : MonoBehaviour
05 {
06     //Reference to the transform component of orc (position,
07     rotation, scale)
08     private Transform ThisTransform = null;
09
10     //Enum for states of orc
11     public enum OrcStates {NEUTRAL, ATTACK_MELEE, ATTACK_RANGE};
12
13     //Current state of orc
14     public OrcStates CurrentState = OrcStates.NEUTRAL;
15
16     //Movement speed of orc in meters per second
17     public float OrcSpeed = 10.0f;
```

```

17
18 //Is orc friendly to player
19 public bool isFriendly = false;
20
21 //-----
22 // Use this for initialization
23 void Start ()
24 {
25     //Get transform of orc
26     ThisTransform = transform;
27 }
28 //-----
29 // Update is called once per frame
30 void Update ()
31 {
32 }
33 //-----
34 //State actions for orc
35 public void AttackMelee()
36 {
37     //Do melee attack here
38 }
39 //-----
40 public void AttackRange()
41 {
42     //Do range attack here
43 }
44 //-----
45 }

```

The following are the comments for code sample 1-10:

- **Line 04:** Here, the class keyword is used to define a class named Orc. This class derives from MonoBehaviour. The next section of this chapter will consider inheritance and derived classes further.
- **Lines 09-19:** Several variables and an enum are added to the Orc class. The variables are of different types, but all are related to the concept of an orc.
- **Lines 35-45:** The orc has two methods: AttackMelee and AttackRange.



More information on classes and their usage in C# can be found at  
<http://msdn.microsoft.com/en-gb/library/x9afc042.aspx>.

## Classes and inheritance

Imagine a scenario where you create an `Orc` class to encode an orc object in the game. Having done so, you then decide to make two upgraded types. One is an `Orc Warlord`, with better armor and weapons, and the other is an `Orc Mage` who, as the name implies, is a spell caster. Both can do everything that the ordinary orc can do, but more besides. Now, to implement this, you can create three separate classes, `Orc`, `OrcWarlord`, and `OrcMage`, by copying and pasting common code between them.

The problem is that as `Orc Warlord` and `Orc Mage` share a lot of common ground and behaviors with `orc`, a lot of code will be wastefully copied and pasted to replicate the common behaviors. Furthermore, if you discovered a bug in the shared code of one class, you'd need to copy and paste the fix to the other classes to propagate it. This is both tedious and technically dangerous, as it risks wasting time, introducing bugs, and causing needless confusion. Instead, the object-oriented concept of inheritance can help us. Inheritance allows you to create a completely new class that implicitly absorbs or contains the functionality of another class, that is, it allows you to build a new class that extends an existing class without affecting the original one. When inheritance happens, two classes are brought into a relationship with each other. The original class (such as the `Orc` class) is known as the base class or ancestor class. The new class (such as the `Orc Warlord` or `Orc Mage`), which extends on the ancestor class, is called a super class or derived class.



More information on inheritance in C# can be found at <http://msdn.microsoft.com/en-gb/library/ms173149%28v=vs.80%29.aspx>.

By default, every new Unity script file creates a new class derived from `MonoBehaviour`. This means every new script contains all the `MonoBehaviour` functionality and has the potential to go beyond, based on the additional code that you add. To prove this, refer to the following code sample 1-11:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class NewScript : MonoBehaviour
05 {
06 //-----
07     // Use this for initialization
08     void Start ()
09     {
10         name = "NewObject";
11     }
```

```

12  //-----
13  // Update is called once per frame
14  void Update ()
15  {
16  }
17 }

```

The following are the comments for code sample 1-11:

- **Line 04:** The class `NewScript` is derived from `MonoBehaviour`. You can, however, substitute `MonoBehaviour` for almost any valid class name from which you want to derive.
- **Line 10:** Here, the variable name is assigned a string during the `Start` event. However, notice that the name is not explicitly declared as a variable anywhere in the `NewScript` source file. If `NewScript` were a completely new class with no ancestor defined in line 04, then line 10 would be invalid. However, because `NewScript` derives from `MonoBehaviour`, it automatically inherits all of its variables, allowing us to access and edit them from `NewScript`.



#### When to inherit

Only use inheritance where it's really appropriate; otherwise, you'll make your classes large, heavy, and confusing. If you're creating a class that shares a lot of common functionality with another and it makes sense to establish connection between them, then use inheritance. Another use of inheritance, as we'll see next, is when you want to override specific functions.

## Classes and polymorphism

To illustrate polymorphism in C#, let's start by considering the following code sample 1-12. This sample doesn't demonstrate polymorphism immediately but represents the start of a scenario where polymorphism will be useful, as we'll see. Here, a basic skeleton class is defined for a potential **non-player character** (NPC) in a generic RPG game. The class is intentionally not comprehensive and features basic variables that only mark the starting point for a character. The most important thing here is that the class features a `SayGreeting` function, which should be invoked when the player engages the NPC in conversation. It displays a generic welcome message to **Console** as follows:

```

01 using UnityEngine;
02 using System.Collections;

```

```
03
04 public class MyCharacter
05 {
06     public string CharName = "";
07     public int Health = 100;
08     public int Strength = 100;
09     public float Speed = 10.0f;
10     public bool isAwake = true;
11
12     //Offer greeting to the player when entering conversation
13     public virtual void SayGreeting()
14     {
15         Debug.Log ("Hello, my friend");
16     }
17 }
```

The first problem to arise relates to the diversity and believability of the `MyCharacter` class if we try imagining how it'd really work in a game. Specifically, every character instantiated from `MyCharacter` will offer exactly the same greeting when `SayGreeting` is invoked: men, women, orcs, and everybody. They'll all say the same thing, namely, "Hello, my friend". This is neither believable nor desirable. Perhaps, the most elegant solution would be to just add a public string variable to the class, thus allowing customization over the message printed. However, to illustrate polymorphism clearly, let's try a different solution. We could create several additional classes instead, all derived from `MyCharacter`, one for each new NPC type and each offering a unique greeting from a `SayGreeting` function. This is possible with `MyCharacter`, because `SayGreeting` has been declared using the `virtual` keyword (line 13). This allows derived classes to override the behavior of `SayGreeting` in the `MyCharacter` class. This means the `SayGreeting` function in derived classes will replace the behavior of the original function in the base class. Such a solution might look similar to the code sample 1-13:

```
01 using UnityEngine;
02 using System.Collections;
03 //-----
04 public class MyCharacter
05 {
06     public string CharName = "";
```



```
07     public int Health = 100;
08     public int Strength = 100;
09     public float Speed = 10.0f;
10     public bool isAwake = true;
11
12     //Offer greeting to the player when entering conversation
13     public virtual void SayGreeting()
14     {
15         Debug.Log ("Hello, my friend");
16     }
17 }
18 //-----
19 public class ManCharacter: MyCharacter
20 {
21     public override void SayGreeting()
22     {
23         Debug.Log ("Hello, I'm a man");
24     }
25 }
26 //-----
27 public class WomanCharacter: MyCharacter
28 {
29     public override void SayGreeting()
30     {
31         Debug.Log ("Hello, I'm a woman");
32     }
33 }
34 //-----
35 public class OrcCharacter: MyCharacter
36 {
37     public override void SayGreeting()
38     {
39         Debug.Log ("Hello, I'm an Orc");
40     }
41 }
42 //-----
```

With this code, some improvement is made, that is, different classes are created for each NPC type, namely, *ManCharacter*, *WomanCharacter*, and *OrcCharacter*. Each offers a different greeting in the *SayGreeting* function. Further, each NPC inherits all the common behaviors from the shared base class *MyCharacter*. However, a technical problem regarding type specificity arises. Now, imagine creating a tavern location inside which there are many NPCs of the different types defined, so far, all enjoying a tankard of grog. As the player enters the tavern, all NPCs should offer their unique greeting. To achieve this functionality, it'd be great if we could have a single array of all NPCs and simply call their *SayGreeting* function from a loop, each offering their own greeting. However, it seems, initially, that we cannot do this. This is because all elements in a single array must be of the same data type, such as *MyCharacter[]* or *OrcCharacter[]*. We cannot mix types for the same array. We could, of course, declare multiple arrays for each NPC type, but this feels awkward and doesn't easily allow for the seamless creation of more NPC types after the array code has been written. To solve this problem, we'll need a specific and dedicated solution. This is where polymorphism comes to the rescue. Refer to the following sample 1-14, which defines a new *Tavern* class in a completely separate script file:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Tavern : MonoBehaviour
05 {
06 //Array of NPCs in tavern
07 public MyCharacter[] Characters = null;
08 //-----
09 // Use this for initialization
10 void Start () {
11
12     //New array - 5 NPCs in tavern
13     Characters = new MyCharacter[5];
14
15     //Add characters of different types to array MyCharacter
16     Characters[0] = new ManCharacter();
17     Characters[1] = new WomanCharacter();
18     Characters[2] = new OrcCharacter();
19     Characters[3] = new ManCharacter();
20     Characters[4] = new WomanCharacter();
21
22     //Now run enter tavern functionality
23     EnterTavern();
24 }
25 //-----
```

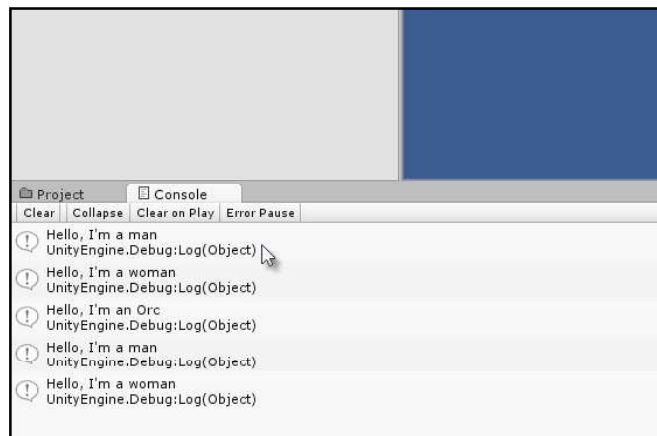
```

26 //Function when player enters Tavern
27 public void EnterTavern()
28 {
29     //Everybody say greeting
30     foreach(MyCharacter C in Characters)
31     {
32         //call SayGreeting in derived class
33         //Derived class is accessible via base class
34         C.SayGreeting();
35     }
36 }
37 //-----
38 }

```

The following are the comments for code sample 1-14:

- **Line 07:** To keep track of all NPCs in the tavern, regardless of the NPC type, a single array (Characters) of type MyCharacter is declared.
- **Lines 16-20:** The Characters array is populated with multiple NPCs of different types. This works because, though they are of different types, each NPC derives from the same base class.
- **Line 27:** The EnterTavern function is called at level startup.
- **Line 34:** A foreach loop cycles through all NPCs in the Characters array, calling the SayGreeting function. The result is shown in the following screenshot. The unique messages for each NPC are printed instead of the generic message defined in the base class. Polymorphism allows the overridden method in the derived classes to be called instead.



Polymorphism produces a backwards transparency between data types that share a common lineage



More information on polymorphism in C# can be found at <http://msdn.microsoft.com/en-GB/library/ms173152.aspx>.

## C# properties

When assigning values to class variables, such as `MyClass.x = 10;`, there are a couple of important things to take care of. First, you'll typically want to validate the value being assigned, ensuring that the variable is always valid. Typical cases include clamping an integer between a minimum and maximum range or allowing only a limited set of strings for a string variable. Second, you might need to detect when a variable changes, initiating other dependent functions and behaviors. C# properties let you achieve both these features. Refer to the following code sample 1-15, which limits an integer between 1 and 10 and prints a message to the console whenever it changes:

```
01 using UnityEngine;
02 using System.Collections;
03 //-----
04 //Sample class - can be attached to object as a component
05 public class Database : MonoBehaviour
06 {
07 //-----
08 //Public property for private variable iMyNumber
09 //This is a public property to the variable iMyNumber
10 public int MyNumber
11 {
12     //Called when retrieving value
13     get
14     {
15         return iMyNumber; //Output iMyNumber
16     }
17
18     //Called when setting value
19     set
20     {
21         //If value is within 1-10, set number else ignore
22         if(value >= 1 && value <= 10)
23         {
24             //Update private variable
25             iMyNumber = value;
26 }
```

---

```

27             //Call event
28             NumberChanged();
29         }
30     }
31 }
32 //-----
33 //Internal reference a number between 1-10
34 private int iMyNumber = 0;
35 //-----
36 // Use this for initialization
37 void Start ()
38 {
39     //Set MyNumber
40     MyNumber = 11; //Will fail because number is > 10
41
42     //Set MyNumber
43     MyNumber = 7; //Will succeed because number is between 1-10
44 }
45 //-----
46 //Event called when iMyNumber is changed
47 void NumberChanged()
48 {
49     Debug.Log("Variable iMyNumber changed to : " +
50             iMyNumber.ToString());
51 }
52 }
53 //-----

```

The following are the comments for code sample 1-15:

- **Line 10:** A public integer property is declared. This property is not an independent variable but simply a wrapper and accessor interface for the private variable `iMyNumber`, declared in line 34.
- **Line 13:** When `MyNumber` is used or referenced, the internal `get` function is called.
- **Line 14:** When `MyNumber` is assigned a value, the internal `set` function is called.
- **Line 25:** The `set` function features an implicit argument value that represents the value to be assigned.
- **Line 28:** The event `NumberChanged` is called when the `iMyNumber` variable is assigned a value.



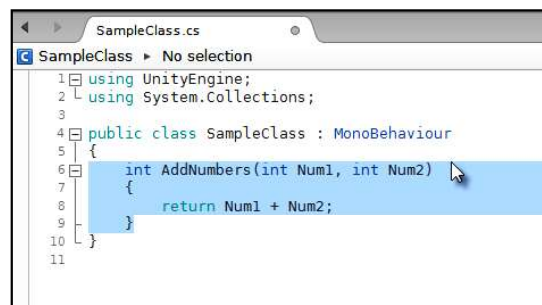
### Properties and Unity

Properties are useful to validate and control the assignment of values to variables. The main problem with using them in Unity concerns their visibility in the Object Inspector. Specifically, C# properties are not shown in the Object Inspector. You can neither get nor set their values in the editor. However, community-made scripts and solutions are available that can change this default behavior, for example exposing C# properties. These scripts and solutions can be found at [http://wiki.unity3d.com/index.php?title=Expose\\_properties\\_in\\_inspector](http://wiki.unity3d.com/index.php?title=Expose_properties_in_inspector).

More information on **Properties** in C# can be found at <http://msdn.microsoft.com/en-GB/library/x9fsa0sw.aspx>.

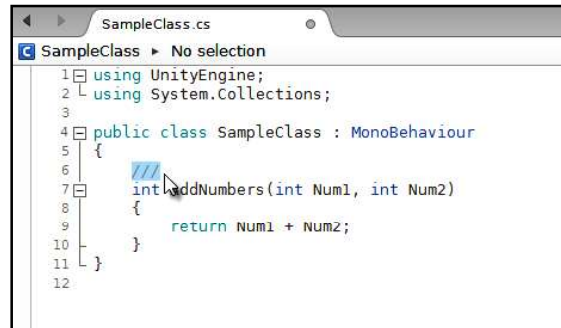
## Commenting

Commenting is the practice of inserting human readable messages into your code, purely for annotation, description, and to make things clearer to the reader. In C#, one-line comments are prefixed with the `//` symbol, and multiline comments begin with `/*` and end with `*/`. Comments are used throughout the code samples in this book. Comments are important, and I recommend that you get into the habit of using them if you're not already in the habit. They benefit not only other developers in your team (if you work with others), but you too! They help remind you of what your code is doing when you return to it weeks or months later, and they even help you get clear and straight about the code you're writing right now. Of course, all these benefits depend on you writing concise and meaningful comments and not long essays filled with irrelevance. However, MonoDevelop offers XML-based comments too to describe functions and arguments specifically and which integrates with code completion. It can significantly boost your workflow, especially when working in teams. Let's see how to use this. Start by writing your function or any function, as shown in the following screenshot:



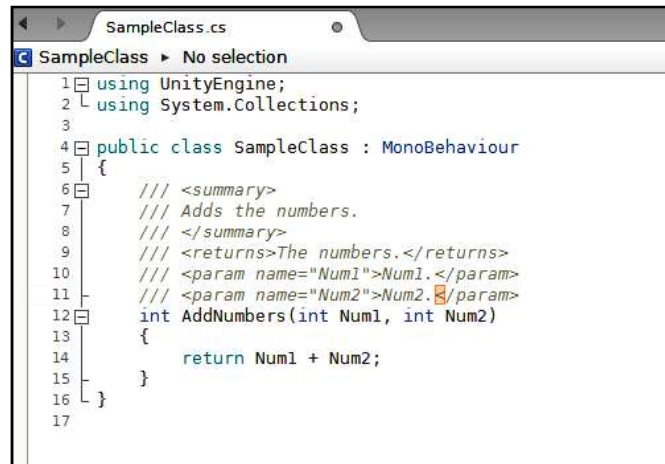
Writing a function (AddNumbers) in MonoDevelop (preparing for code commenting)

Then insert three forward-slash characters on the line above the function title (///), as shown in the following screenshot:



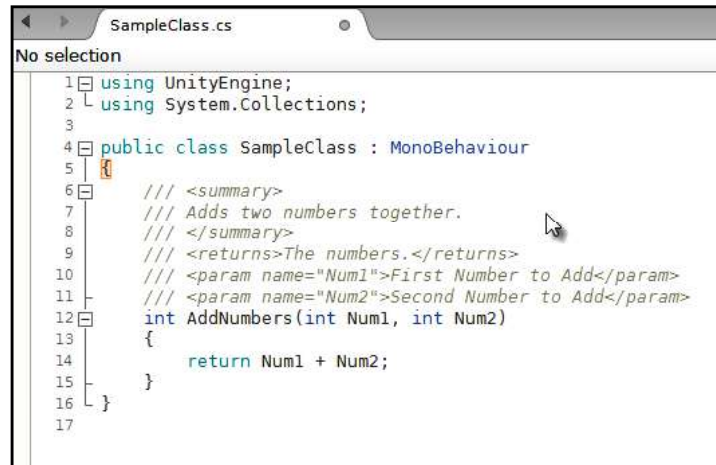
Inserting /// above the function title to create an XML comment

When you do this, MonoDevelop automatically inserts a template XML comment ready for you to complete with appropriate descriptions. It creates a summary section that describes the function generally and param entries for each argument in the function, as shown in the following screenshot:



Inserting /// above the function title will autogenerate an XML comment

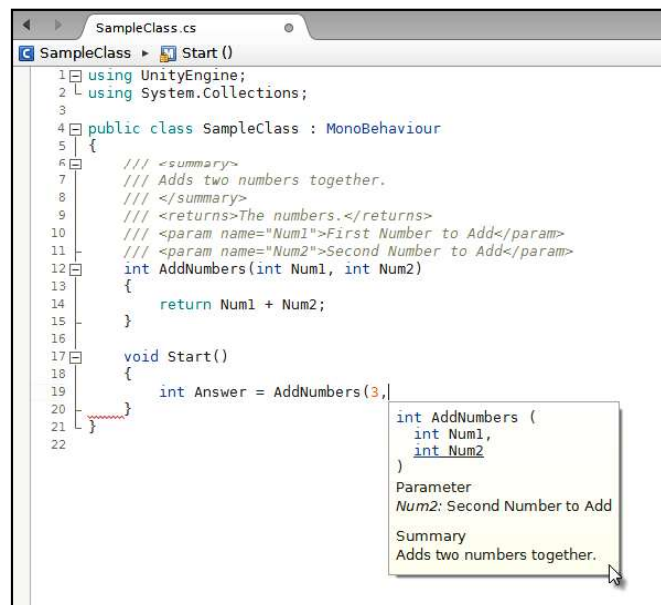
Next, fill in the XML template completely with comments for your function. Be sure to give each parameter an appropriate comment too, as shown in the following screenshot:



```
1 using UnityEngine;
2 using System.Collections;
3
4 public class SampleClass : MonoBehaviour
5 {
6     /// <summary>
7     /// Adds two numbers together.
8     /// </summary>
9     /// <returns>The numbers.</returns>
10    /// <param name="Num1">First Number to Add</param>
11    /// <param name="Num2">Second Number to Add</param>
12    int AddNumbers(int Num1, int Num2)
13    {
14        return Num1 + Num2;
15    }
16 }
17
```

Commenting your functions using XML comments

Now, when calling the AddNumbers function elsewhere in code, the code-completion pop-up helper will display both the summary comment for the function as well as the parameter comments' context sensitively, as shown here:



```
1 using UnityEngine;
2 using System.Collections;
3
4 public class SampleClass : MonoBehaviour
5 {
6     /// <summary>
7     /// Adds two numbers together.
8     /// </summary>
9     /// <returns>The numbers.</returns>
10    /// <param name="Num1">First Number to Add</param>
11    /// <param name="Num2">Second Number to Add</param>
12    int AddNumbers(int Num1, int Num2)
13    {
14        return Num1 + Num2;
15    }
16
17    void Start()
18    {
19        int Answer = AddNumbers(3,
20
21    }
22
```

```
int AddNumbers (
    int Num1,
    int Num2
)
Parameter
Num2: Second Number to Add
Summary
Adds two numbers together.
```

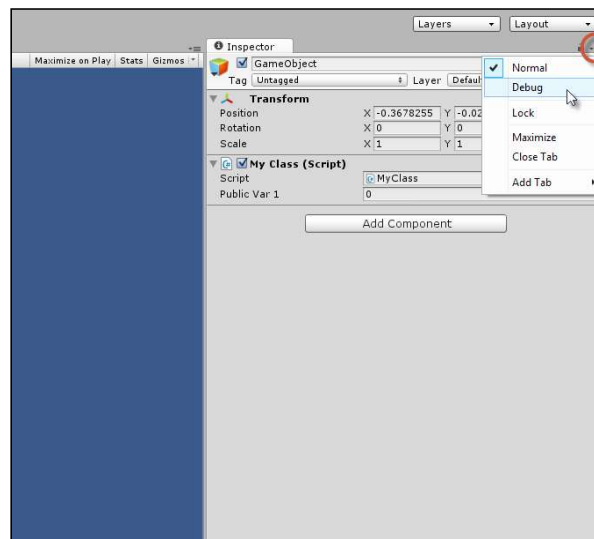
Viewing comments when making function calls



## Variable visibility

One excellent feature of Unity specifically is that it exposes (shows) public class variables inside the Object Inspector in the Unity Editor, allowing you to edit and preview variables, even at runtime. This is especially convenient for debugging. However, by default, the Object Inspector doesn't expose private variables. They are typically hidden from the inspector. This isn't always a good thing because there are many cases where you'll want to debug or, at least, monitor private variables from the inspector without having to change their scope to public. There are two main ways to overcome this problem easily.

The first solution would be useful if you want to view all public and private variables in a class. You can toggle the Object Inspector in the **Debug** mode. To do this, click on the context menu icon in the top-right corner of the **Inspector** window and select **Debug** from the context menu, as shown in the following screenshot. When **Debug** is selected, all the public and private variables for a class will show.



Enabling the Debug mode in the Object Inspector will show all the variables in a class

The second solution is useful for displaying specific private variables, variables that you mark explicitly as wanting to display in the Object Inspector. These will show in both the **Normal** and **Debug** modes. To achieve this, declare the private variable with the attribute `[SerializeField]`. C# attributes are considered later in this book, as shown here:

```
01 using UnityEngine;
02 using System.Collections;
03
```

```
04 public class MyClass : MonoBehaviour
05 {
06     //Will always show
07     public int PublicVar1;
08
09     //Will always show
10     [SerializeField]
11     private int PrivateVar1;
12
13     //Will show only in Debug Mode
14     private int PrivateVar2;
15
16     //Will show only in Debug Mode
17     private int PrivateVar3;
18 }
```



You can also use the `[HideInInspector]` attribute to hide a global variable from the inspector.

## The ? operator

The if-else statements are so common and widely used in C# that a specialized shorthand notation is available for writing simpler ones, without resorting to the full multiline if-else statements. This shorthand is called the ? operator. The basic form of this statement is as follows:

```
//If condition is true then do expression 1, else do expression 2
(condition) ? expression_1 : expression_2;
```

Let's see the ? operator in a practical example as shown here:

```
//We should hide this object if its Y position is above 100 units
bool ShouldHideObject = (transform.position.y > 100) ? true :
false;

//Update object visibility
gameObject.SetActive(!ShouldHideObject);
```



The ? operator is useful for shorter statements, but for long and more intricate statements, it can make your code harder to read.

## SendMessage and BroadcastMessage

The `MonoBehaviour` class included in the Unity API, which acts as the base class for most new scripts, offers the `SendMessage` and `BroadcastMessage` methods. Using these, you can easily execute functions by name on all components attached to an object. To invoke a method of a class, you typically need a local reference to that class to access and run its functions as well as to access its variables. However, the `SendMessage` and `BroadcastMessage` functions let you run functions using string values by simply specifying the name of a function to run. This is very convenient and makes your code look a lot simpler and shorter at the cost of efficiency, as we'll see later. Refer to the following code sample 1-16:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyClass : MonoBehaviour
05 {
06     void start()
07     {
08         //Will invoke MyFunction on ALL components/scripts
           attached to this object (where the function is present)
09
           SendMessage("MyFunction",
           SendMessageOptions.DontRequireReceiver);
10     }
11
12     //Runs when SendMessage is called
13     void MyFunction()
14     {
15         Debug.Log ("hello");
16     }
17 }
```

The following are the comments for code sample 1-16:

- **Line 09:** `SendMessage` is called to invoke the function `MyFunction`. `MyFunction` will be invoked not only on this class but on all other components attached to `GameObject`, if they have a `MyFunction` member, including the `Transform` component as well as others.
- **Line 09:** The parameter `SendMessageOptions.DontRequireReceiver` defines what happens if `MyFunction` is not present on a component. Here, it specifies that Unity should ignore the component and move on to the next calling `MyFunction` wherever it is found.



The term function and member function mean the same thing when the function belongs to a class. A function that belongs to a class is said to be a member function.

We've seen that `SendMessage` invokes a specified function across all components attached to a single `GameObject`. `BroadcastMessage` incorporates the `SendMessage` behavior and goes a stage further, that is, it invokes a specified function for all components on `GameObject` and then repeats this process recursively for all child objects in the scene hierarchy, cascading downwards to all children.

More information on `SendMessage` and `BroadcastMessage` can be found at <http://docs.unity3d.com/ScriptReference/GameObject.SendMessage.html> and <http://docs.unity3d.com/ScriptReference/Component.BroadcastMessage.html>.



### Reflection

`SendMessage` and `BroadcastMessage` are effective ways to facilitate inter-object communication and inter-component communication, that is, it's a great way to make components talk to one another if they need to, to synchronize behavior and recycle functionality. However, both `SendMessage` and `BroadcastMessage` rely internally on a C# feature known as **reflection**. By invoking a function using a string, your application is required to look at itself at runtime (to reflect), searching its code for the intended function to run. This process is computationally expensive compared to running a function in the normal way. For this reason, seek to minimize the usage of `SendMessage` and `BroadcastMessage`, especially during `Update` events or other frame-based scenarios, as the impact on performance can be significant. This doesn't mean you should never use them. There might be times when their use is rare, infrequent, and convenient and has practically no appreciable impact. However, later chapters in this book will demonstrate alternative and faster techniques using delegates and interfaces.

If you'd like more information on C# and its usage before proceeding further with this book, then I recommend the following sources:

- *Learning C# by Developing Games with Unity 3D Beginner's Guide*, Terry Norton, Packt Publishing

- *Intro to C# Programming and Scripting for Games in Unity, Alan Thorn*  
(3DMotive video course found at <https://www.udemy.com/3dmotive-intro-to-c-programming-and-scripting-for-games-in-unity/>)
- *Pro Unity Game Development with C#, Alan Thorn, Apress*

The following are a few online resources:

- <http://msdn.microsoft.com/en-gb/library/aa288436%28v=vs.71%29.aspx>
- <http://www.csharp-station.com/tutorial.aspx>
- <http://docs.unity3d.com/ScriptReference/>

## Summary

This chapter offered a general, Unity-specific overview of Unity's C#, exploring the most common and widely-used language features for game development. Later chapters will revisit some of these subjects in a more advanced way, but everything covered here will be critical for understanding and writing the code featured in subsequent chapters.

