HOUR 7

Scripting, Part 1

What You'll Learn in This Hour:

- The basics of scripts in Unity
- How to use variables
- How to use operators
- How to use conditionals
- How to use loops

You have so far in this book learned how to make objects in Unity. However, those objects have been a bit boring. How useful is a cube that just sits there? It would be much better to give the cube some custom action to make it interesting in some way. For this you need scripts. *Scripts* are files of codes that are used to define complex or nonstandard behaviors for objects. In this hour, you'll learn about the basics of scripting. You'll begin by looking at how to start working with scripts in Unity. You'll learn how to create scripts and use the scripting environment. Then you'll learn about the various components of a scripting language, including variables, operators, conditionals, and loops.

TIP

Sample Scripts

Several of the scripts and coding structures mentioned in this hour are available in the book assets for Hour 7. Be sure to check them out for additional learning.

CAUTION

New to Programming

If you have never programmed before, this lesson might seem strange and confusing. As you work through this hour, try your best to focus on how things are structured and why they are structured that way. Remember that programming is purely logical. If a program is not doing something you want it to, it is because you have not told it how to do it correctly. Sometimes it is up to you to change the way you think. Take this hour slowly and be sure to practice.

Scripts

As mentioned earlier in this hour, using scripts is a way to define behavior. Scripts attach to objects in Unity just like other components and give them interactivity. There are generally three steps involved in working with scripts in Unity:

1. Create the script.
2. Attach the script to one or more game objects.
3. If the script requires it, populate any properties with values or other game objects.

The rest of this lesson discusses these steps.

Creating Scripts

Before creating scripts, it is best to create a Scripts folder under the Assets folder in the Project view. Once you have a folder to contain all your scripts, simply right-click the folder and select **Create > C# Script**. Then give your script a name before continuing.

NOTE

Scripting Language

Unity allows you to write scripts in C# or JavaScript. This book uses the C# language for all scripts because it is a little more versatile and powerful in Unity. It is worth noting that JavaScript is being phased out, and the option to create new JavaScript files doesn't even appear in the editor anymore. At

some point in the future, support for the language will be dropped entirely.

Once a script is created, you can view and modify it. Clicking a script in the Project view enables you to see the contents of the script in the Inspector view (see Figure 7.1). Double-clicking the script in the Project view opens your default editor, where you can add code to the script. Assuming that you have installed the default components and haven't changed anything, double-clicking a file opens the Visual Studio development environment (see Figure 7.2).

FIGURE 7.1
The Inspector view of a script.

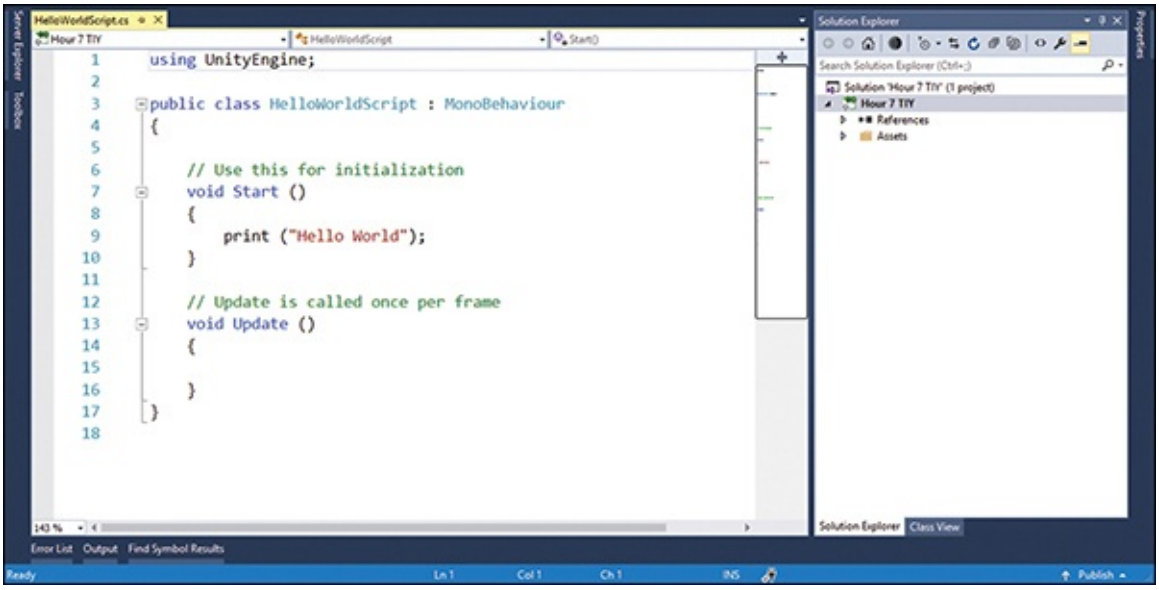

FIGURE 7.2
The Visual Studio software with the editor window showing.

▼ TRY IT YOURSELF

Creating a Script

Follow these steps to create a script for use in this section:

1. Create a new project or scene and add a **Scripts** folder to the Project view.
2. Right-click the Scripts folder and choose **Create > C# Script**. Name the script **HelloWorldScript**.
3. Double-click the new script file and wait for Visual Studio to open. In the editor window of Visual Studio (refer to Figure 7.2), erase all the text and replace it with the following code:

Click here to view code image

```
using UnityEngine;

public class HelloWorldScript : MonoBehaviour
{
    // Use this for initialization
    void Start ()
    {
```

```
        {
        print ("Hello World");
        }

        // Update is called once per frame
        void Update ()
        {

        }
}
```

4. Save your script by selecting **File > Save** or by pressing **Ctrl+S** (**Command+S** on a Mac). Back in Unity, confirm in the Inspector view that the script has been changed and run the scene. Notice that nothing happens. The script was created, but it does not work until it is attached to an object, as discussed in the next section.

NOTE

Script Names

You just created a script named HelloWordScript; the name of the actual script file is important. In Unity and C#, the name of the file must match the name of the class that is inside it. Classes are discussed later in this hour, but for now, suffice it to say that if you have a script containing a class named MyAwesomeClass, the file that contains it will be named MyAwesomeClass.cs. It is also worth noting that classes, and therefore script filenames, cannot contain spaces.

NOTE

IDEs

Visual Studio is a robust and complex piece of software that comes bundled with Unity. Editors like this are known as IDEs (integrated development environments), and they assist you with writing the code for games. Since IDEs are not actually part of Unity, this book does not cover them in any depth. The only part of Visual Studio you need to be familiar with right now is the editor window. If there is anything else you need to know about IDEs, it is covered in the hour where it is needed. (*Note:* Prior to Unity 2018.1, an IDE called MonoDevelop also came packaged with Unity. You can still acquire and use this software individually, but MonoDevelop is no longer

shipped with the engine.)

Attaching a Script

To attach a script to a game object, just click the script in the Project view and drag it onto the object (see Figure 7.3). You can drag the script onto the object in the Hierarchy view, the Scene view, or the Inspector view (assuming that the object is selected). Once attached to an object, the script becomes a component of that object and is visible in the Inspector view.

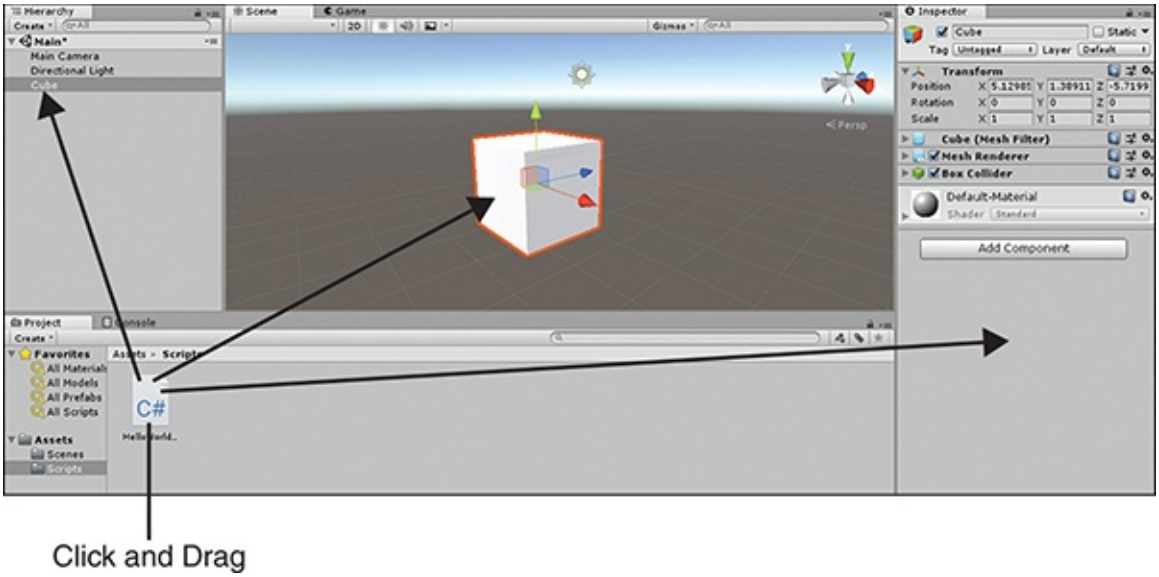

FIGURE 7.3
Clicking and dragging the script onto the desired object.

To see this in action, attach the script HelloWorldScript that you created earlier to the Main Camera. You should now see a component named Hello World Script (Script) in the Inspector view. If you run the scene, you see Hello World appear at the bottom of the editor, underneath the Project view (see Figure 7.4).

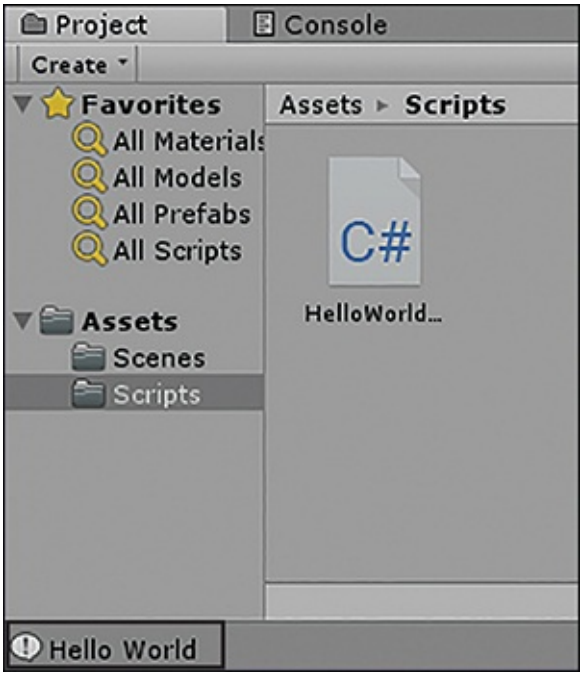

FIGURE 7.4
The words Hello World output when running the scene.

Anatomy of a Basic Script

In the preceding section, you modified a script to output some text to the screen, but the contents of the script were not explained. In this section, you'll look at the default template that is applied to every new C# script. (Note that scripts written in JavaScript have the same components even if they look a little different.) Listing 7.1 contains the full code that is generated for you by Unity when you make a new script named HelloWorldScript.

Listing 7.1 Default Script Code

Click here to view code image

```
using UnityEngine;
using System.Collections;

public class HelloWorldScript : MonoBehaviour
{
    // Use this for initialization
    void Start () {

    }
    // Update is called once per frame
    void Update () {
```

```
    }
}
```

This code can be broken down into three parts: the using section, the class declaration section, and the class contents.

The Using Section

The first part of the script lists the libraries that the script will be using. It looks like this:

Click here to view code image

```
using UnityEngine;
using System.Collections;
```

Generally speaking, you don't change this section too often and should just leave it alone for the time being. These lines are usually added for you when you create a script in Unity. The System.Collections library is optional and is often omitted if the script doesn't use any functionality from it.

The Class Declaration Section

The next part of the script is called a *class declaration*. Every script contains a class that is named after the script. It looks like this:

Click here to view code image

```
public class HelloWorldScript : MonoBehaviour { }
```

All the code in between the opening bracket `{` and closing bracket `}` is part of this class and therefore is part of the script. All your code should go between these brackets. As with the using section, you rarely change the class declaration section and should just leave it alone for now.

The Class Contents

The section in between the opening and closing brackets of the class is considered to be "in" the class. All your code goes here. By default, a script contains two methods inside the class, `Start` and `Update`:

Click here to view code image

```
// Use this for initialization
```

```
void Start () {

}
 // Update is called once per frame
void Update () {

}
```

Methods are covered in greater detail in Hour 8, “Scripting, Part 2.” For now, just know that any code inside the `Start` method runs when a scene first starts. Any code inside the `Update` method runs as fast as possible—even hundreds of times a second.

TIP

Comments

Programming languages enable code authors to leave messages for those who read the code later. These messages are called *comments*. Any words that follow two forward slashes (//) are “commented out.” This means that the computer will skip over them and not attempt to read them as code. You can see an example of commenting in the Try It Yourself “Creating a Script,” earlier in this hour.

NOTE

The Console

There is another window in the Unity editor that has not been mentioned until now: the Console. Basically, the Console is a window that contains text output from your game. Often, when there is an error or output from a script, messages get written to the Console. Figure 7.5 shows the Console. If the Console window isn’t visible, you can access it by selecting **Window > Console.**

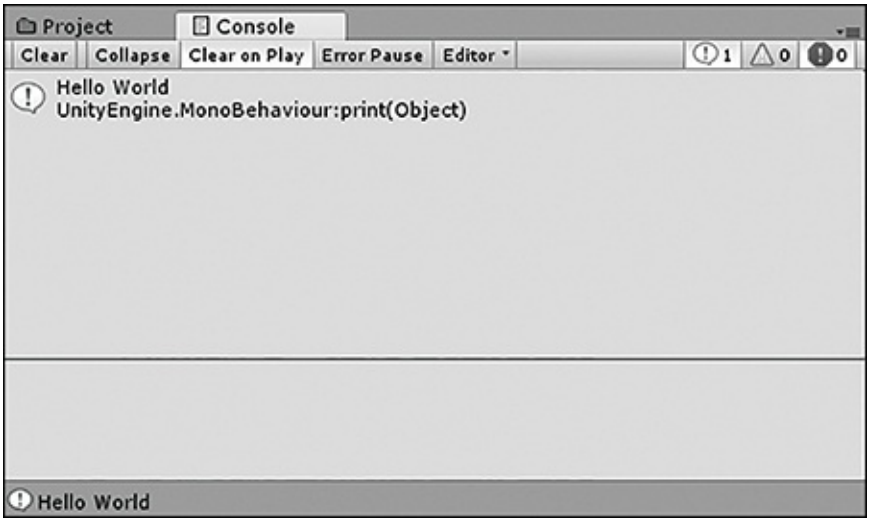

FIGURE 7.5
The Console window.

TRY IT YOURSELF

Using the Built-in Methods

Now you're ready to try out the built-in methods `Start` and `Update` and see how they work. The completed ImportantFunctions script is available in the book assets for Hour 7. Try to complete the exercise that follows on your own, but if you get stuck, refer to the book assets:

1. Create a new project or scene. Add a script to the project named **ImportantFunctions**. Double-click the script to open it in your code editor.

2. Inside the script, add the following line of code to the `Start` method:

Click here to view code image

```
print ("Start runs before an object Updates");
```

3. Save the script, and in Unity, attach it to the Main Camera. Run the scene and notice the message that appears in the Console window.

4. Back in Visual Studio, add the following line of code to the `Update` method:

Click here to view code image

```
print ("This is called once a frame");
```

5. Save the script and quickly start and stop the scene in Unity. Notice how, in the Console, there is a single line of text from the `Start` method and there are a bunch of lines from the `Update` method.

Variables

Sometimes you want to use the same bit of data more than once in a script. In such a case, you need a placeholder for data that can be reused. Such placeholders are called *variables*. Unlike in traditional math, variables in programming can contain more than just numbers. They can hold words, complex objects, or other scripts.

Creating Variables

Every variable has a name and a type that are given to the variable when it is created. You create a variable with the following syntax:

```
<variable type> <name>;
```

So, to create an integer named num1, you type the following:

```
int num1;
```

Table 7.1 lists all the primitive (or basic) variable types and the types of data they can hold.

NOTE

Syntax

The term *syntax* refers to the rules of a programming language. Syntax dictates how things are structured and written so that the computer knows how to read them. You may have noticed that every statement, or command, in a script so far has ended with a semicolon. This is also a part of the C# syntax. Forgetting the semicolon causes your script to not work. If you want to know more about the syntax of C#, check out the C# Guide at https://docs.microsoft.com/en-us/dotnet/csharp/.

TABLE 7.1 C# Variable Types

Type	Description

`int`	Short for integer, `int` stores positive or negative whole numbers.
`float`	`float` stores floating-point data (such as 3.4) and is the default number type in Unity. `float` numbers in Unity are always written with an f after them, such as 3.4f, 0f, .5f, and so on.
`double`	`double` also stores floating-point numbers; however, it is not the default number type in Unity. It can generally hold bigger numbers than `float`.
`bool`	Short for Boolean, `bool` stores true or false (actually written in code as `true` or `false`).
`char`	Short for character, `char` stores a single letter, space, or special character (such as a, 5, or !). `char` values are written with single quotes (`'A'`).
`string`	The `string` type holds entire words or sentences. String values are written with double quotes (`"Hello World"`).

Variable Scope

Variable scope refers to where a variable is able to be used. As you have seen in scripts, classes and methods use open and close brackets to denote what belongs to them. The area between the two brackets is often referred to as a *block*. The reason this is important is that variables are only able to be used in the blocks in which they are created. So if a variable is created inside the `Start` method of a script, it is not available in the `Update` method because they are two different blocks. Attempting to use a variable where it is not available results in an error. If a variable is created in a class but outside a method, it will be available to both methods because both methods are in the same block as the variable (the class block). Listing 7.2 demonstrates this.

Listing 7.2 Demonstration of Class and Local Block Levels

Click here to view code image

```
// This is in the "class block" and will
// be available everywhere in this class
private int num1;

void Start ()
{
    // this is in a "local block" and will
```

```
    // this is in a "local block" and will
    // only be available in the Start method
    int num2;
}
```

Public and Private

In Listing 7.2, you can see that the keyword `private` appears before `num1`. This is called an *access modifier*, and it is needed only for variables declared at the class level. There are two access modifiers you need to use: `private` and `public`. A lot can be said about the two access modifiers, but what you really need to know at this point is how they affect variables. Basically, a private variable (a variable with the word `private` before it) is usable only inside the file in which it is created. Other scripts and the editor cannot see it or modify it in any way. Private variables are intended for internal use only. Public variables, in contrast, are visible to other scripts and even the Unity editor. This makes it easy for you to change the values of your variables on-the-fly within Unity. If you do not mark a variable as public or private, it defaults to private.

TRY IT YOURSELF

Modifying Public Variables in Unity

Follow these steps to see how public variables are visible in the Unity editor:

1. Create a new C# script and in Visual Studio add the following line in the class above the `Start` method:

```
public int runSpeed;
```

2. Save the script and then, in Unity, attach it to the Main Camera.

3. Select the Main Camera and look in the Inspector view. Notice the script you just attached as a component. Now notice that the component has a new property: Run Speed. You can modify that property in the Inspector view, and the change will be reflected in the script at runtime. Figure 7.6 shows the component with the new property. This figure assumes that the script created was named ImportantFunctions.

FIGURE 7.6
The new Run Speed property of the script component.

Operators

All the data in variables is worthless if you have no way of accessing or modifying it. Operators are special symbols that enable you to perform modifications on data. They generally fall into one of four categories: arithmetic operators, assignment operators, equality operators, and logical operators.

Arithmetic Operators

Arithmetic operators perform some standard mathematical operation on variables. They are generally used only on number variables, although a few exceptions exist. Table 7.2 describes the arithmetic operators.

TABLE 7.2 Arithmetic Operators

Operator	Description
`+`	Addition. Adds two numbers together. In the case of strings, the + sign concatenates, or combines, them. The following is an example: `"Hello" + "World"; // produces "HelloWorld"`
`-`	Subtraction. Reduces the number on the left by the number on the right.
`*`	Multiplication. Multiplies two numbers together.
`/`	Division. Divides the number on the left by the number on the right.

`%`	Modulus. Divides the number on the left by the number on the right but does not return the result. Instead, the modulus returns the remainder of the division. Consider the following examples:
	`10 % 2; // returns 0`
	`6 % 5; // returns 1`

```
24 % 7; // returns 3
```

Arithmetic operators can be cascaded together to produce more complex math strings, as in this example:

```
x + (5 * (6 – y) / 3);
```

Arithmetic operators work in the standard mathematic order of operations. Math is done left to right, with anything in parentheses calculated first, multiplication and division done second, and addition and subtraction done third.

Assignment Operators

Assignment operators are just what they sound like: They assign values to variables. The most notable assignment operator is the equals sign, but there are more assignment operators that combine multiple operations. All assignment in C# is right to left. This means that whatever is on the right side gets moved to the left. Consider these examples:

Click here to view code image

```
x = 5; // This works. It sets the variable x to 5.
5 = x; // This does not work. You cannot assign a variable to a value (5
```

Table 7.3 describes the assignment operators.

TABLE 7.3 Assignment Operators

Operator	Description
=	Assigns the value on the right to the variable on the left.

+ =, – =, *=, /=	Shorthand assignment operator that performs some arithmetic operation based on the symbol used and then assigns the result to whatever is on the left. Consider these examples:
	`x = x + 5; // Adds 5 to x and then assigns it to x`
	`x += 5;  // Does the same as above, only shorthand`

++, –	Shorthand operators called the increment and decrement operators. They increase or decrease a number by 1. Consider these examples: `x = x + 1; // Adds 1 to x and then assigns it`

```
to x
x++;        // Does the same as above, only
shorthand
```

Equality Operators

Equality operators compare two values. The result of an equality operator is always either `true` or `false`. Therefore, the only variable type that can hold the result of an equality operator is a Boolean. (Remember that Booleans can only contain `true` or `false`.) Table 7.4 describes the equality operators.

TABLE 7.4 Equality Operators

Operator	Description

==	Not to be confused with the assignment operator (=), this operator returns `true` only if the two values are equal. Otherwise, it returns `false`. Consider these examples:
	`5 == 6; // Returns false`
	`9 == 9; // Returns true`

>, <	These are the "greater than" and "less than" operators. Consider these examples:
	`5 > 3; // Returns true`
	`5 < 3; // Returns false`

>=, <=	These are similar to "greater than" and "less than" except that they are the "greater than or equal to" and "less than or equal to" operators. Consider these examples:
	`3 >= 3; // Returns true`
	`5 <= 9; // Returns true`

!=	This is the "not equal" operator, and it returns `true` if the two values are not the same. Otherwise, it returns `false`. Consider these examples:
	`5 != 6; // Returns true`
	`9 != 9; // Returns false`

TIP

TIP

Additional Practice

In the book assets for Hour 7 is a script called EqualityAndOperations.cs. Be sure to look through it for some additional practice with the various operators.

Logical Operators

Logical operators enable you to combine two or more Boolean values (`true` or `false`) into a single Boolean value. They are useful for determining complex conditions. Table 7.5 describes the logical operators.

TABLE 7.5 Logical Operators

Operator	Description

`&&`	Known as the AND operator, this compares two Boolean values and determines whether they are both true. If either, or both, of the values is false, this operator returns `false`. Consider these examples:
	`true && false; // Returns false`
	`false && true; // Returns false`
	`false && false; // Returns false`
	`true && true;  // Returns true`

`||`	Known as the OR operator, this compares two Boolean values and determines whether either of them is true. If either or both of the values are true, this operator returns `true`. Consider these examples:
	`true || false; // Returns true`
	`false || true; // Returns true`
	`false || false; // Returns false`
	`true || true;  // Returns true`

`!`	Known as the NOT operator, this returns the opposite of a Boolean value. Consider these examples: `!true;  // Returns false`