HOUR 8

Scripting, Part 2

What You'll Learn in This Hour:

- How to write methods
- How to capture user input
- How to work with local components
- How to work with game objects

In Hour 7, "Scripting, Part 1," you learned about the basics of scripting in Unity. In this hour, you'll use what you have learned to complete more meaningful tasks. First, you'll examine what methods are, how they work, and how to write them. Then you'll get some hands-on practice with user input. After that, you'll examine how to access components from scripts. You'll wrap up the hour by learning how to access other game objects and their components with code.

TIP

Sample Scripts

Several of the scripts and coding structures mentioned in this hour are available in the book assets for Hour 8. Be sure to check them out for additional learning.

Methods

Methods, often called *functions*, are modules of code that can be called and used independently of each other. Each method generally represents a single task or

purpose, and often many methods can work together to achieve complex goals. Consider the two methods you have seen so far: `Start` and `Update`. Each represents a single and concise purpose. The `Start` method contains all the code that is run for an object when the scene first begins. The `Update` method contains the code that is run every update frame of the scene (which is different from a render frame, or "frames per second").

NOTE

Method Shorthand

You have seen so far that whenever the `Start` method is mentioned, the word *method* has followed it. It can become cumbersome to always have to specify that a word used is a method. You can't write just `Start`, though, because people wouldn't know if you meant the word, a variable, or a method. A shorter way of handling this is to use parentheses with the word. So, the method `Start` can be rewritten as just `Start()`. If you ever see something written like `SomeWords()`, you know instantly that the writer is talking about a method named `SomeWords`.

Anatomy of a Method

Before working with methods, you should look at the different parts that compose them. The following is the general format of a method:

Click here to view code image

```
<return type> <name> (<parameter list>)
{
    <Inside the method's block>
}
```

Method Name

Every method must have a unique name (sort of…see the "Method Signature" section below). Though the rules that govern proper names are determined by the language used, good general guidelines for method names include the following:

- Make a method name descriptive. It should be an action or a verb.
- Spaces are not allowed.
- Avoid using special characters (for example, `!`, `@`, `*`, `%`, `$`) in method

names. Different languages allow different characters. By not using any, you avoid risking problems.

Method names are important because they allow you to both identify and use them.

Return Type

Every method has the ability to return a variable back to whatever code called it. The type of this variable is called the *return type*. If a method returns an integer (a whole number), the return type is an `int`. Likewise, if a method returns a `true` or a `false`, the return type is `bool`. If a method doesn't return any value, it still has a return type. In that instance, the return type is `void` (meaning nothing). Any method that returns a value will do so with the keyword `return`.

Parameter List

Just as methods can pass a variable back to whatever code called it, the calling code can pass variables in. These variables are called *parameters*. The variables sent into the method are identified in the parameter list section of the method. For example, a method named `Attack` that takes an integer called `enemyID` would look like this:

Click here to view code image

```
void Attack(int enemyID)
{}
```

As you can see, when specifying a parameter, you must provide both the variable type and the name. Multiple parameters are separated with commas.

Method Signature

The combination of a method's return type, name, and parameters list is often referred to as a method's *signature*. Earlier in this hour I mentioned that a method must have a unique name, but that isn't exactly true. What is true is that a method must have a unique signature. Therefore, consider these two methods:

Click here to view code image

```
void MyMethod()
{}

void MyMethod(int number)
{}
```

Even though these two methods have the same name, they have different parameter lists and thus are different. This practice of having different methods with the same name can also be called *overloading* a method.

Method Block

The method block is where the code of the method goes. Every time a method is used, the code inside the method block executes.

TRY IT YOURSELF

Identifying Method Parts

Take a moment to review the different parts of a method and then consider the following method:

Click here to view code image

```
int TakeDamage(int damageAmount)
{
    int health = 100;
    return health - damageAmount;
}
```

Can you identify the following pieces?

1. What is the method's name?
2. What variable type does the method return?
3. What are the method's parameters? How many are there?
4. What code is in the method's block?

TIP

Methods as Factories

The concept of methods can be confusing for someone who is new to programming. Often, mistakes are made regarding method parameters and what methods return. A good way to keep it straight is to think of a method as a factory. Factories receive raw materials that they use to make products. Methods work the same way. The parameters are the materials you are passing in to the "factory," and the return is the final product of that factory. Just think of methods that don't take parameters as factories that don't

require raw goods. Likewise, think of methods that don't return anything as factories that don't produce final products. By imagining methods as little factories, you can work to keep the flow of logic straight in your head.

Writing Methods

When you understand the components of a method, writing them is easy. Before you begin writing methods yourself, take a moment to answer three main questions:

- What specific task will the method achieve?
- Does the method need any outside data to achieve its task?
- Does the method need to give back any data?

Answering these questions will help you determine a method's name, parameters, and return data.

Consider this example: A player has been hit with a fireball. You need to write a method to simulate this by removing 5 health points. You know what the specific task of this method is. You also know that the task doesn't need any data (because you know it takes 5 points) and should probably return the new health value back. You could write the method like this:

Click here to view code image

```
int TakeDamageFromFireball()
{
    int playerHealth = 100;
    return playerHealth - 5;
}
```

As you can see in this method, the player's health is 100, and 5 is deducted from it. The result (which is 95) is passed back. Obviously, this can be improved. For starters, what if you want the fireball to do more than 5 points of damage? You would then need to know exactly how much damage a fireball is supposed to do at any given time. You would need a variable, or in this case a parameter. Your new method could be written as follows:

Click here to view code image

```
int TakeDamageFromFireball(int damage)
{
    int playerHealth = 100;
    return playerHealth - damage;
}
```

Now you can see that the damage is read in from the method and applied to the health. Another place where this can be improved is with the health itself. Currently, players can never lose because their health will always refresh back to 100 before having damage deducted. It would be better to store the player's health elsewhere so that its value is persistent. You could then read it in and remove the damage appropriately. Your method could then look like this:

Click here to view code image

```
int TakeDamageFromFireball(int damage, int playerHealth)
{
    return playerHealth - damage;
}
```

By examining your needs, you can build better, more robust methods for your game.

NOTE

Simplification

In the preceding example, the resulting method simply performs basic subtraction. This is oversimplified for instruction's sake. In a more realistic environment, there are many ways to handle this task. A player's health could be stored in a variable belonging to a script. In this case, it would not need to be read in. Another possibility would be to use a complex algorithm in the `TakeDamageFromFireball` method to reduce the incoming damage by some armor value, a player's dodging ability, or a magical shield. If the examples here seem silly, just bear in mind that they are meant to demonstrate various elements of the topic.

Using Methods

Once a method is written, all that is left is to use it. Using a method is often referred to as *calling* or *invoking* the method. To call a method, you just need to write the method's name followed by parentheses and any parameters. So, if you were trying to use a method named `SomeMethod`, you would write the following:

```
SomeMethod();
```

If `SomeMethod()` requires an integer parameter, you call it like this:

Click here to view code image

```
// Method call with a value of 5
SomeMethod(5);
// Method call passing in a variable
int x = 5;
SomeMethod(x); // do not write "int x" here.
```

Note that when you call a method, you do not need to supply the variable type with the variable you are passing in. If `SomeMethod()` returns a value, you want to *catch* it in a variable. The code could look something like this (with a Boolean return type assumed; in reality, it could be anything):

```
bool result = SomeMethod();
```

Using this basic syntax is all there is to calling methods.

▼ TRY IT YOURSELF

Calling Methods

Let's work further with the `TakeDamageFromFireball` method described in the previous section. This exercise shows how to call the various forms of the method. (You can find the solution for this exercise as FireBallScript in the book assets for Hour 8.) Follow these steps:

1. Create a new project or scene. Create a C# script called FireBallScript and enter the three `TakeDamageFromFireball` methods described earlier. These should go inside the class definition, at the same level of indent as the `Start()` and `Update()` methods, but outside those two methods.

2. In the `Start` method, call the first `TakeDamageFromFireball()` method by typing the following:

Click here to view code image

```
int x = TakeDamageFromFireball();
print ("Player health: " + x);
```

3. Attach the script to the Main Camera and run the scene. Notice the output in the Console. Now call the second `TakeDamageFromFireball()` method in `Start()` by typing the following (placing it below the first bit of code you typed; no need to remove it):

Click here to view code image

```
int y = TakeDamageFromFireball(25);
print ("Player health: " + y);
```

4. Again, run the scene and note the output in the Console. Finally, call the last `TakeDamageFromFireball()` method in `Start()` by typing the following:

Click here to view code image

```
int z = TakeDamageFromFireball(30, 50);
print ("Player health: " + z);
```

5. Run the scene and note the final output. Notice how the three methods behave a little differently from one another. Also notice that you called each one specifically, and the correct version of the `TakeDamageFromFireball()` method was used based on the parameters you passed in.

TIP

Help Finding Errors

If you are getting errors when you try to run your script, pay attention to the reported line number and character number in the Console, at the end of the error message. Furthermore, you can "build" your code inside Visual Studio by using **Ctrl+Shift+B** (**Command+Shift+B** on a Mac). When you do this, Visual Studio checks your code and points out any errors in context, showing you exactly where the troublesome line is. Try it.

Input

Without player input, *video games* would just be *video*. Player input can come in many different varieties. Inputs can be physical—for example, gamepads, joysticks, keyboards, and mice. There are capacitive controllers such as the relatively new touch screens in modern mobile devices. There are also motion devices like the Wii Remote, the PlayStation Move, and the Microsoft Kinect. Rarer is the audio input that uses microphones and a player's voice to control a game. In this section, you'll learn all about writing code to allow the player to interact with your game by using physical devices.

Input Basics

With Unity (as with most other game engines), you can detect specific key presses in code to make it interactive. However, doing so makes it difficult to allow players to remap the controls to their preference, so it is a good idea to avoid doing that. Thankfully, Unity has a simple system for generically mapping controls. With Unity, you look for a specific *axis* to know whether a player intends a certain action. Then, when the player runs the game, he can choose to make different controls mean different axes.

You can view, edit, and add different axes by using the Input Manager. To access the Input Manager, click **Edit > Project Settings > Input**. In the Input Manager, you can see the various axes associated with different input actions. By default, there are 18 input axes, but you can add your own if you want. Figure 8.1 shows the default Input Manager with the horizontal axis expanded.

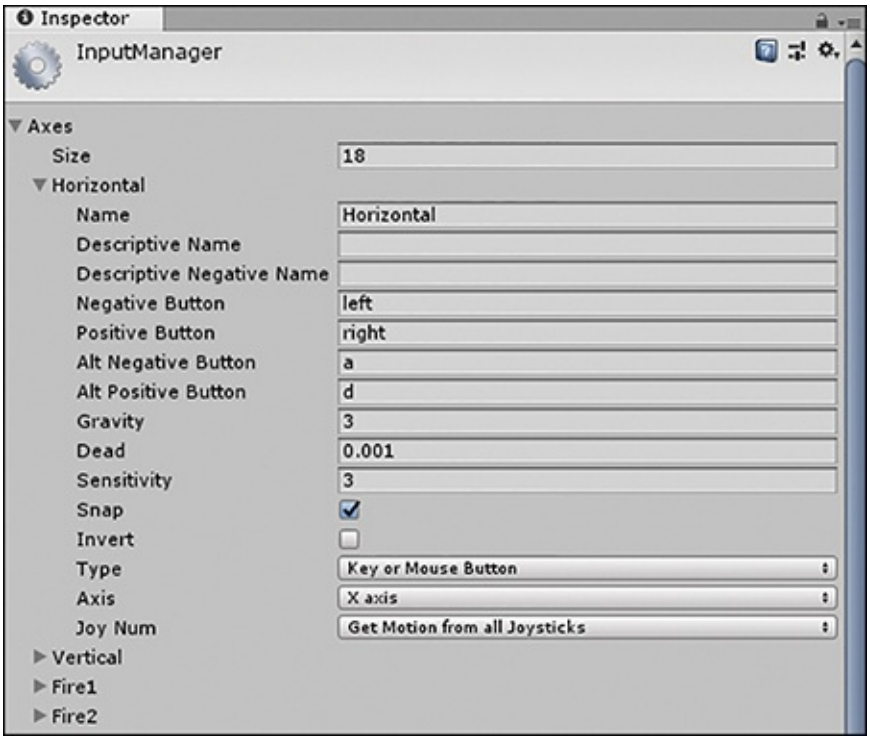

FIGURE 8.1
The Input Manager.

While the horizontal axis doesn't directly control anything (you will write scripts to do that later), it represents that player going sideways. Table 8.1 describes the properties of an axis.

TABLE 8.1 Axis Properties

Property	Description

Name	The name of the axis. This is how you reference it in code.
Descriptive Name/ Descriptive Negative Name	A verbose name for the axis that will appear to the player in the game configuration. The negative is the opposite name. For example: “Go left” and “Go right” would be a name and negative name pair.
Negative Button/ Positive Button	The buttons that pass negative and positive values to the axis. For the horizontal axis, these are the left arrow and the right arrow keys.
Alt Negative Button/ Alt Positive Button	Alternate buttons to pass values to the axis. For the horizontal axis, these are the A and D keys.
Gravity	How quickly the axis returns to 0 once the key is no longer pressed.
Dead	The value below which any input will be ignored. This helps prevent jittering with joystick devices.
Sensitivity	How quickly the axis responds to input.
Snap	When checked, Snap causes the axis to immediately go to 0 when the opposite direction is pressed.
Invert	Checking this property inverts the controls.
Type	The type of input. The types are keyboard/mouse buttons, mouse movement, and joystick movement.
Axis	The corresponding axis from an input device. This doesn’t apply to buttons.
Joy Num	Which joystick to get input from. By default, this property gets input from all joysticks.

Input Scripting

Once your axes are set up in the Input Manager, working with them in code is simple. To access any of the player’s input, you will be using the `Input` object.

More specifically, you will be using the `GetAxis` method of the `Input` object. `GetAxis()` reads the name of the axis as a string and returns the value of that axis. So, if you want to get the value of the horizontal axis, you type the following:

Click here to view code image

```
float hVal = Input.GetAxis("Horizontal");
```

In the case of the horizontal axis, if the player is pressing the left arrow key (or the A key), `GetAxis()` returns a negative number. If the player is pressing the right arrow key (or the D key), the method returns a positive value.

▼ TRY IT YOURSELF

Reading in User Input

Follow these steps to work with the vertical and horizontal axes and get a better idea of how to use player input:

1. Create a new project or scene. Add a script named PlayerInput to the project and attach the script to the Main Camera.
2. Add the following code to the `Update` method in the PlayerInput script:

Click here to view code image

```
float hVal = Input.GetAxis("Horizontal");
float vVal = Input.GetAxis("Vertical");
if(hVal != 0)
    print("Horizontal movement selected: " + hVal);
if(vVal != 0)
    print("Vertical movement selected: " + vVal);
```

This code must be in `Update` so that it continuously reads the input.

3. Save the script and run the scene. Notice what happens on the Console when you press the arrow keys. Now try out the W, A, S, and D keys. If you don’t see anything, click in the Game window with the mouse and try again.

Specific Key Input

Although you generally want to deal with the generic axes for input, sometimes you want to determine whether a specific key has been pressed. To do so, you will again be using the `Input` object. This time, however, you will use the `GetKey` method, which reads in a special code that corresponds to a specific key. It then returns `true` if the key is currently down or `false` if the key is not currently down. To determine whether the K key is currently pressed, you type the following:

Click here to view code image

```
bool isKeyDown = Input.GetKey(KeyCode.K);
```

TIP

Finding Key Codes

Each key has a specific key code. You can determine the key code of the specific key you want by reading the Unity documentation. Alternatively, you can use the built-in tools of Visual Studio to find it. Whenever you are working on a script in Visual Studio, you can always type in the name of an object followed by a period. When you do, a menu with all the possible options pops up. Likewise, if you type an open parenthesis after typing a method name, the same menu pops up, showing you the various options. Figure 8.2 illustrates using this pop-up menu to find the key code for the Esc key.

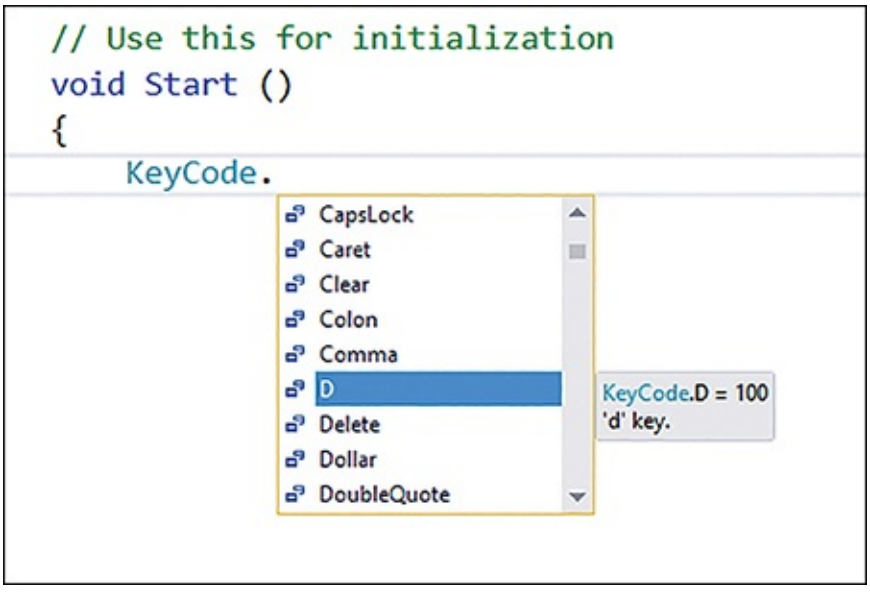

FIGURE 8.2
The automatic pop-up in Visual Studio.

TRY IT YOURSELF

Reading in Specific Key Presses

Follow these steps to write a script that determines whether a specific key is pressed:

1. Create a new project or scene. Add to the project a script named PlayerInput (or modify the existing one) and attach it to the Main Camera.

2. Add the following code to the `Update` method in the PlayerInput script:

Click here to view code image

```
if(Input.GetKey(KeyCode.M))
    print("The 'M' key is pressed down");
if(Input.GetKeyDown(KeyCode.O))
    print("The 'O' key was pressed");
```

3. Save the script and run the scene. Notice what happens when you press the M key versus what happens when you press the O key. In particular, note that the M key causes output the entire time it is held, while the O key outputs only when it is first pressed.

TIP

"Gotta Gotta Get Up to Get Down"

In the Try It Yourself exercise "Reading in Specific Key Presses," you checked for key input in two different ways. Namely, you used `Input.GetKey()` to test whether a key was pressed at all. You also used `Input.GetKeyDown()` to test whether a key was pressed during this frame. This second version only registered the first time the key was pressed and ignored you holding down the key. Generally speaking, Unity looks for three types of key press events: `GetKey()`, `GetKeyDown()`, and `GetKeyUp()`. This is consistent across the other similar methods as well: `GetButton()`, `GetButtonDown()`, `GetButtonUp()`, `GetMouseButton()`, `GetMouseButtonDown()`, and so on. Knowing when to look for a first key press versus a held button is important.

Whichever type of input you need, though, there's a method for it.

Mouse Input

Besides capturing key presses, you want to capture mouse input from the user. There are two components to mouse input: mouse button presses and mouse movement. Determining whether mouse buttons are pressed is much like detecting key presses, covered earlier this hour. In this section, you will again be using the `Input` object. This time you'll use the `GetMouseButtonDown` method, which takes an integer between 0 and 2 to dictate which mouse button you are asking about. The method returns a Boolean value indicating whether the button is pressed. The code to get the mouse button presses looks like this:

Click here to view code image

```
bool isButtonDown;
isButtonDown = Input.GetMouseButtonDown(0); // left mouse button
isButtonDown = Input.GetMouseButtonDown(1); // right mouse button
isButtonDown = Input.GetMouseButtonDown(2); // center mouse button
```

Mouse movement is only along two axes: x and y. To get the mouse movement, you use the `GetAxis` method of the input object. You can use the names Mouse X and Mouse Y to get the movement along the x axis and the y axis, respectively. The code to read in the mouse movement would look like this:

Click here to view code image

```
float value;
value = Input.GetAxis("Mouse X"); // x axis movement
value = Input.GetAxis("Mouse Y"); // y axis movement
```

Unlike button presses, mouse movement is measured by the amount the mouse has moved since the last frame only. Basically, holding a key causes a value to increase until it maxes out at -1 or 1 (depending on whether it is positive or negative). The mouse movement, however, generally has smaller numbers because it is measured and reset each frame.

▼ TRY IT YOURSELF

Reading Mouse Movement

In this exercise, you'll read in mouse movement and output the results to the Console: