Learning C# by Developing Games with Unity 5.x
Second Edition

Learning C# by Developing Games with Unity 5.x is a step-by-step guide that will help you to get started with programming behaviors in C# by creating a 2D game in Unity. Starting with the installation of Unity and learning its features, followed by creating a C# script, we will then deal with topics such as Unity scripting to understand how code work so you can create and use C# variables and methods. Moving forward, you will find out how to create, store, and retrieve data from collections of objects.

You will also develop an understanding of loops and their uses, and you’ll perform object-oriented programming. This will help you to turn your idea into a ready-to-code project and set up a Unity project for production. Finally, you will discover how to create the GameManager class to manage the gameplay loop, generate game levels, and develop a simple UI for the game.

Who this book is written for

The book is targeted at beginner level Unity developers with no programming experience. If you are a Unity developer and you wish to learn how to write C# scripts and code by creating games, then this book is for you.

What you will learn from this book

- Understand the fundamentals of variables, methods, and code syntax in C#
- Get acquainted to the techniques that will turn your game idea into a working project
- Use loops and collections efficiently in Unity to reduce the amount of code
- Develop a game using object-oriented programming principles
- Generate infinite levels for your game
- Create and code a good-looking functional UI system for your game
- Publish and share your game with users

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Getting into the Details of Methods'
- A synopsis of the book’s content
- More information on Learning C# by Developing Games with Unity 5.x Second Edition
About the Author

Greg Lukosek was born and raised in the Upper Silesia region of Poland. When he was about 8 years old, his amazing parents bought him and his brother a Commodore C64. That was when his love of programming started. He would spend hours writing simple basic code, and when he couldn't write it on the computer directly, he used a notepad.

Greg completed his mechanical engineering diploma at ZSTiO Meritum—Siemianowice Slaskie, Poland. He has learned all his programming skills through determination and hard work at home.

Greg met the love of his life, Kasia, in 2003, which changed his life forever. They both moved to London in search of adventure and decided to stay there.

He started work as a 3D artist and drifted away from programming for some years. Deep inside, he still felt the urge to come back to game programming. During his career as a 3D artist, he discovered Unity and adopted it for an interactive visualizations project. At that very moment, he started programming again.

His love for programming overcomes his love for 3D graphics. Greg ditched his 3D artist career and came back to writing code professionally. He is now doing what he really wanted to do since he was 8 years old—developing games.

These days, Greg lives in a little town called Sandy in the UK with Kasia and their son, Adam.
Hello, future game developers! If you are reading this book, you are probably a curious person trying to learn more about a great game engine — Unity — and specifically, programming in C#. This book will take you on a learning journey. We will go through it together, beginning with the fundamentals of programming and finishing with a functional 2D platform game.

What this book covers

Chapter 1, Discovering Your Hidden Scripting Skills and Getting Your Environment Ready, puts you at ease with writing scripts for Unity.

Chapter 2, Introducing the Building Blocks for Unity Scripts, helps you develop the skill of writing your first executable code.

Chapter 3, Getting into the Details of Variables, teaches you about creating and using C# variables, followed editing them in Unity Inspector.

Chapter 4, Getting into the Details of Methods, helps you learn more in detail about methods and how to use them to understand the importance of code blocks and the variables used in them.

Chapter 5, Lists, Arrays, and Dictionaries, introduces slightly more complex ideas of handling, lists, arrays, and dictionaries, which allow you to store many values at once.

Chapter 6, Conditions and Looping, helps you learn how to “ask” Unity to loop through a section of code and do something useful.

Chapter 7, Objects, a Containers with Variables and Methods, dives into the subjects of organizing your code and object-oriented programming.

Chapter 8, Let’s Make a Game! – From Idea to Development, shows you how to turn an idea into a ready-to-code project and how to break down complex mechanics into pieces.
Preface

Chapter 9, Starting Your First Game, helps us transform an idea into a real Unity project.

Chapter 10, Writing GameManager, gets you acquainted with the basics of the singleton approach and also helps you work through the gameplay loop.

Chapter 11, The Game Level, helps you learn how to create reusable pieces of a level and also how to populate them to create the illusion of an endlessly running game.

Chapter 12, The User Interface, explains how to construct and implement the user interface in our game.

Chapter 13, Collectables — What Next?, focuses on collectables and storing some data between Unity sessions.
Getting into the Details of Methods

In the previous chapter, you were introduced to a variable's scope, within which a variable exists and is allowed to be used. The scope is determined by the opening and closing curly braces. The purpose of those curly braces is to act as a container for a block of executable code—a code block. In the second chapter, you understood that a method is a code block that can execute by just calling the method's name. It's time to understand the importance of code blocks and the variables used in them. A method defines a code block that begins and ends with curly braces.

In this chapter, we will cover the following topics:

- Using methods in a script
- Naming methods the good way
- Defining a method
- Calling a method
- Returning a value from a method

Variables are the first major building block of C# and methods are the second, so let's dive into methods.

Using methods in a script

There are two reasons to use methods in a script:

- To provide a behavior to GameObject
- To create reusable sections of code
Getting into the Details of Methods

All of the executable code in a script is inside methods. The first purpose of a method is to work with the member variables of the class. The member variables store data that is needed for a component to give a GameObject its behavior. The whole reason for writing a script is to make a GameObject do something interesting. A method is the place where we make a behavior come to life.

The second purpose of a method is to create code blocks that will be used over and over again. You don't want to be writing the same code over and over. Instead, you place the code in a code block and give it a name so that you can call it whenever needed.

Let's take a quick look at this example:

```csharp
void AddAndPrintTwoNumbers(int number1, int number2) {
    int result = number1 + number2;
    Debug.Log(result);
}
```

This is a perfect example of the function that does something useful. It might look a bit strange to you as it takes two parameters. Don't worry about it too much as of now; we will cover it in detail soon. All I want you to notice right now is that the preceding method can take some data and do something useful with it. In this case, it is adding two numbers and printing the result on the Unity console. Now, the best part now—we can call this method as many times as we want, passing different parameters, without repeating the code every time we need it. If you feel confused, don't worry. Just remember that a function can save you from repeating code over and over again.

Methods can also return some data. We will cover this at a later stage in this chapter.

**Naming methods properly**

Always use meaningful names for your methods. Just as I explained for variables, if you don't use good names, then six months from now, you will be confused.

Since methods make GameObject do something useful, you should give your method a name that sounds like an *action*, for example, JumpOverTheFence or ClimbTheWall. You can look at those names and know exactly what the method is going to do.
Don't make them too simple. Suppose you name a method Wiggle. Sure, you know what Wiggle means right now, but six months later, you'll look at that and say "Wiggle? Wiggle what?" It takes only a moment more to be a little more precise and write WiggleDogsTail. Now, when you see this method name, you'll know exactly what it's going to do.

**Beginning method names with an uppercase letter**

Why? We do this to make it easier to tell the difference between a class or method and a variable. Also, Microsoft recommends beginning method names with an uppercase letter. If someone else ever looks at your code, they will expect to see method names beginning with an uppercase letter.

**Using multiword names for a method**

Let's use this example again:

```csharp
void AddTwoNumbers ()
{
    // Code goes here
}
```

You can see that the name is actually three words squished together. Since method names can have only one word, the first word begins with an uppercase, and then we just capitalize the first letter of every additional word, for example, PascalCasing.

**Parentheses are part of the method's name**

The method name always includes a pair of parentheses at the end. These parentheses not only let you know that the name is of a method, but also serve an important purpose of allowing you to input some data into the method when needed.

**Defining a method the right way**

Just as with variables, we have to let Unity know about a method before we can use it. Depending on who you talk to, some will say "We have to declare a method," others will say "We have to define a method," or even "We have to implement a method." Which is correct? In C#, it doesn't make any difference. Use whichever term helps you learn more easily. I like to say I'm defining a method's code block, nothing like declaring a simple variable on a one-line statement.
The minimum requirements for defining a method

There are three minimum requirements for defining a method:

- The type of information, or data, that a method will return to the place from where it was called
- The name of the method should be followed by a pair of parentheses
- A pair of curly braces should be present to contain the code block:

  ```
  returnType NameOfTheMethod()
  {
  }
  ```

Looking at LearningScript once again, or any Unity-generated script, you can see that the `Start()` method has the three minimum requirements for a method:

  ```
  void Start()
  {
  }
  ```

Here’s what we have:

- Our first requirement is the type of data that the method will return to the place in the code that called this method. This method isn’t returning any value, so instead of specifying an actual type of data, the `void` keyword is used. This informs Unity that nothing is being returned from the method.
- The second requirement is the method name, which is `Start()`.
- The last requirement is the curly braces. They contain the code that defines what the method is going to do.
This example fulfills the bare minimum requirements for a method. However, as you can see, there's no code in the code block, so when `Start()` is called by Unity, it doesn't do anything at all. Yet it's a method. Normally, if we aren't going to use a method by adding code to a skeleton method created by Unity, we can simply remove them from our script. It's normally best to remove unused code after the script has been written.

Here's what we know about this bare-minimum method definition as far as Unity is concerned:

- There's no public modifier, which means that this method is private by default. Therefore, this method cannot be called from other scripts.
- There's no code in the code block. Therefore, this method doesn't do anything. So, it can be removed if we wish to remove it.

Methods that do not return any data use the `void` keyword instead of `datatype`.

**Understanding parentheses – why are they there?**

One thing for sure is that parentheses make it easy to recognize that it's a method, but why are they part of a method's name?

We already know that a method is a code block that is going to be called multiple times. That's one of the reasons a method is created in the first place—so that we don't have to write the same code over and over. Remember the `AddAndPrintTwoNumbers()` example method? We have mentioned that a method can take some input parameters. Why is this useful?
Getting into the Details of Methods

A script may need to add two numbers several times, but they probably won't always be the same two numbers. We can have possibly hundreds of different combinations of two numbers to add together. This means that we need to let the method know which two numbers need to be added together at the moment when we call the method. Let's write a code example to make sure you fully understand it:

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

public class LearningReusableMethods : MonoBehaviour {

    public int number1 = 2;
    public int number2 = 3;
    public int number3 = 7;

    void Start () {
        AddAndPrintTwoNumbers(number1, number2);
        AddAndPrintTwoNumbers(number1, number3);
        AddAndPrintTwoNumbers(number2, number3);
    }

    void AddAndPrintTwoNumbers(int firstNumber, int secondNumber) {
        int result = firstNumber + secondNumber;
        Debug.Log(result);
    }
}
```

Lines 7, 8, and 9 should be quite clear to you—simple declarations of variables.

Let's take a look at the AddAndPrintTwoNumbers method. It's a void function. Again, this means the function does something but does not return any data. Inside the parentheses, our method takes two variables: `firstNumber` and `secondNumber`.

Line 25 contains the declaration and assignment of the local variable that we will be printing on line 26.
So, AddAndPrintTwoNumbers is written the universal way. We can reuse this function as many times as we want, passing different parameters.

Lines 15, 16, and 17 call our function three times, each time passing different parameters to the function. Let's test whether it works! Go ahead, add the LearningReusableMethods component to any GameObject in the Unity scene, and click on Play.

As this script executes, the AddAndPrintTwoNumbers method is called three times on lines 15, 16, and 17. The method's code block adds two numbers and displays the result in the Unity Console tab:

As expected! The console will print out the values. There's a special name for information between the parentheses of a method definition, such as line 23—the code is called method parameters.

**Specifying a method's parameters**

If you look up the word "parameters" in the dictionary, your brain will probably seize up. All it means is that the method has to be able to use the information you send it, so you simply have to specify the type of data that the method is allowed to use. That's it! It's very simple.

In the earlier screenshot, on line 23, we declared the firstNumber and secondNumber variables. The type is int. Now notice our member variables: number1, number2, and number3. They are also of the int type. These variables have to be of the int type since they store the numbers that will be added in the method, which the parameters specify will be of int the type.

So now, go look in the dictionary again. You will probably see the word limit in there somewhere. That's what you did when you specified the type of data that the method will use, an integer in this case. You set some limits on what is allowed.
Okay, so you're setting parameters, or limits, on the type of data the method can use, but what exactly is a parameter? Well, the first parameter is called `firstNumber`. And what is `firstNumber` doing? It stores a value that will be used in the code block on line 25. What do we call things that store data? That's right, variables! Variables are used everywhere.

As you can see on line 25 of the code block, those variables are being added and stored in the `result` variable.

### How many parameters can a method have?

We can have as many parameters as we need to make a method work properly. Whether we write our own custom methods or use the methods of the scripting reference, the parameters that are defined are what the method will require to be able to perform its specified task.

### Returning a value from a method

Now it's time to discover the *power* feature of using a method. This usually means sending data to the method, which you just learned to do. Then we have the method return a value. Previously, we used a `void` type method. I have mentioned before that this is a keyword for *nothing*, which means that the function isn't returning anything.

Let's learn about `return` type functions now. We won't use `void` anymore. Instead of that, we will write the type of data that we want our method to return. Don't worry if this sounds complicated; it isn't. I remember that, years ago, I had some issues getting my head around it. In practice, this is a very simple concept.

Let's take a look at the following example. I have highlighted two key areas that we will speak about next.

```c
int AddTwoNumbers (int firstNumber, int secondNumber) {
    int result = firstNumber + secondNumber;
    return result;
}
```
As you can see, this method is very similar to the `AddAndPrintTwoNumbers` method that we spoke of previously. The two main differences are highlighted.

A return type function will always begin with a description of the type of data that it's returning. In this case, we will be returning the sum of two numbers, so our type is `int` (an integer). In simple words, the `AddTwoNumbers` function is returning a number.

**Returning the value**

Once you have decided what type of data will be returned by a method, you must tell the function what value will be returned. The syntax is very straightforward. We use the `return` keyword, as highlighted in blue, followed by the value we are returning.

**Example**

You just learned how to write a return type method. Time to put it to use! Let's write a new script and call it `LearningReusableMethodsWithReturn`:

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

public class LearningReusableMethodsWithReturn : MonoBehaviour {
    public int number1 = 2;
    public int number2 = 3;

    void Start () {
        int sumResult = AddTwoNumbers(number1, number2);
        DisplayResult(sumResult);
    }

    int AddTwoNumbers (int firstNumber, int secondNumber) {
        int result = firstNumber + secondNumber;
        return result;
    }

    void DisplayResult (int total) {
        Debug.Log("The grand total is: " + total);
    }
}
```
Getting into the Details of Methods

What do we have here? You probably understand most of this code with no issues, but it's good practice to go through it line by line. Lines 7 and 8 contain declarations of the number1 and number2 integer variables. Lines 22 to 27 are exactly the same as we used in the last example. They have the declaration of a method that takes two parameters—firstNumber and secondNumber—and it returns a value of the int type.

Lines 30 to 34 contain the declaration of a method that simply prints the given int value on the Unity console. Now is the most important part you need to remember. Take a look at line 14:

```csharp
int sumResult = AddTwoNumbers(number1, number2);
```

The left-hand side of this line is a simple declaration of an int variable called sumResult. Simple! What I want to talk about is the right-hand side—the assignment of this variable. As you can see, what we are doing here is calling the AddTwoNumbers method instead of simply giving the value to be stored in sumResult. It might look a bit awkward. You would expect a value to be passed instead of another method call.

Let me explain how it works. The AddTwoNumbers method is a return type method. It does return an int value in every place where you call it—instantly. In even simpler words, AddTwoNumbers() is an integer, and a number value.

This concept might be a bit difficult to get your head around. If you still don't get it, don't worry. All you need to remember right now is the fact that, whenever a program calls a method that returns something, it is calling the method and inserting the value that the method returns into the place where it made the call.

Remember I told you that, when you call a method, it's just a substitute for the code block that will be executed. It's like taking all of the code in the method's code block and placing it right where the method was called.

Summary

In this chapter, you learned more details about methods. We will start using methods everywhere in this book. Feel free to come back to this chapter if you feel lost.

In the next chapter, we will introduce a little more complex ideas of handling, lists, arrays, and dictionaries.
Where to buy this book

You can buy Learning C# by Developing Games with Unity 5.x Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.

Get more information Learning C# by Developing Games with Unity 5.x
Second Edition