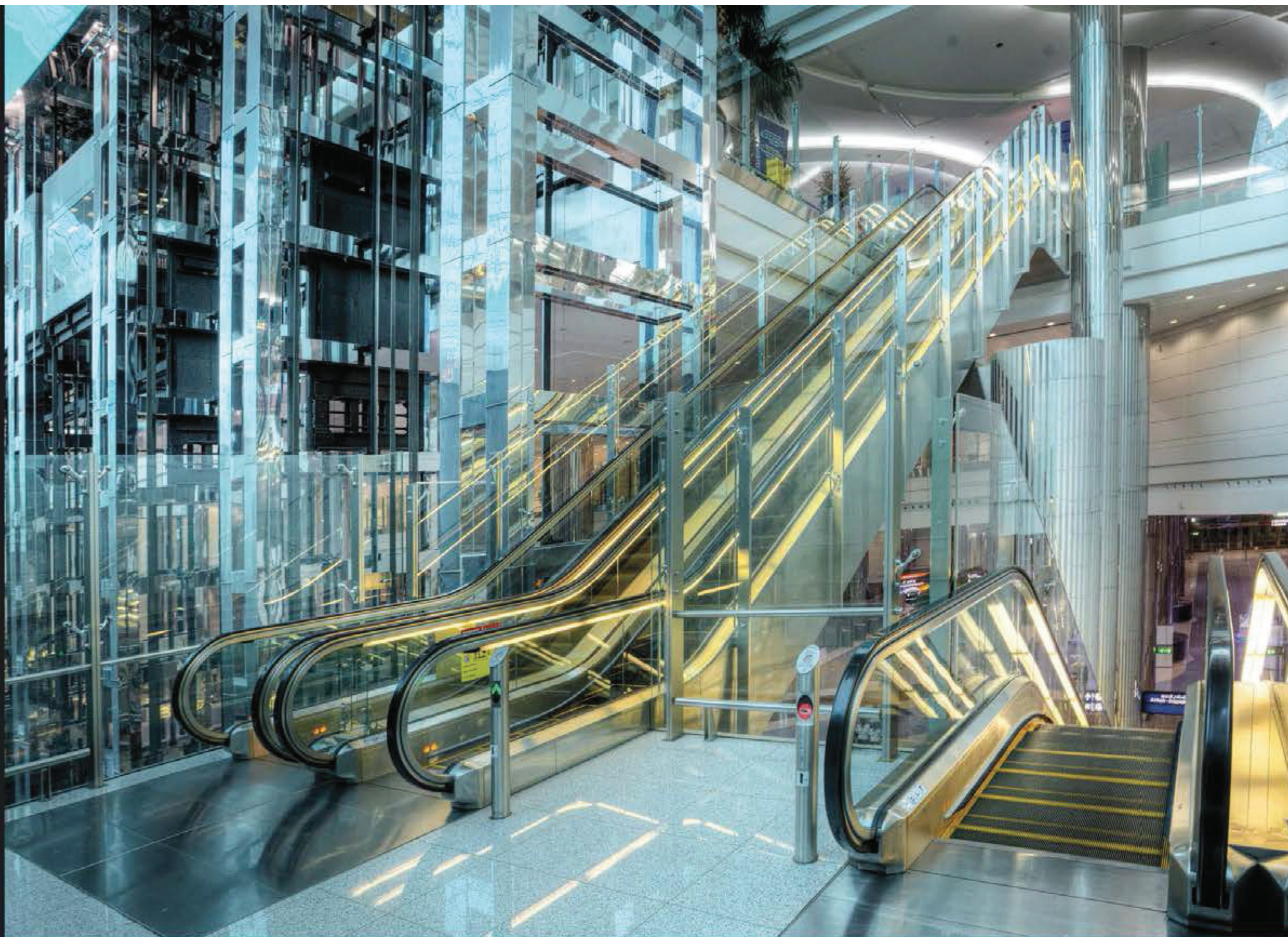


Learn by doing: less theory, more results

Learning C# by Developing Games with Unity 3D

Learn the fundamentals of C# to create scripts for your GameObjects

Beginner's Guide

Terry Norton

[PACKT] PUBLISHING

Learning C# by Developing Games with Unity 3D Beginner's Guide

Learn the fundamentals of C# to create scripts for your GameObjects

Terry Norton

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Learning C# by Developing Games with Unity 3D Beginner's Guide

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2013

Production Reference: 1190913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-84969-658-6

www.packtpub.com

Cover Image by Artie Ng (artherng@yahoo.com.au)

Credits

Author
Terry Norton

Reviewers
Gaurav Garg
Kristian Hedeholm

Acquisition Editor
James Jones

Lead Technical Editor
Dayan Hyames

Technical Editors
Ruchita Bhansali
Dylan Fernandes
Dipika Gaonkar
Monica John
Proshonjit Mitra

Project Coordinator
Apeksha Chitnis

Proofreader
Ameesha Green

Indexers
Rekha Nair
Tejal Soni

Graphics
Ronak Dhruv

Production Coordinator
Aditi Gajjar

Cover Work
Aditi Gajjar

About the Author

Terry Norton was born and raised in California. During the Vietnam era, he served six and half years in the US Air Force. While in the military, he was trained in electronics for electronic counter-measures. Upon discharge, he earned his Electrical Engineering degree, and later working for Joslyn Defense Systems in Vermont, designing and developing test equipment for the US Navy.

When personal computers came on the scene, he took an interest in building computers, but never quite delved deep into the programming side. It wasn't until 2004 that programming peaked his interest. He began writing articles for OS/2 Magazine to teach C++ programming. Unfortunately, damaging his left hand in a snowblower accident in 2005 ended his writing for a couple years.

IBM abandoned OS/2, so Terry bought his first Apple computer in early 2006. He tried a few times to learn Objective-C, but work and family always seemed to sidetrack his efforts. It wasn't until about 2010 when he discovered Unity and the need to write scripts, that he finally made some progress into the programming world. He began writing an online tutorial for UnityScript titled *UnityScript for Noobs*. It was a basic tutorial for beginners made available just before Unite 2011.

Since then, Terry has been learning C# for writing scripts for Unity. Packt Publishing noticed *UnityScript for Noobs* and asked if he would be interested in writing a book about learning UnityScript. He declined. He felt that C# was a better language, and his heart just wasn't into UnityScript any longer. Two weeks later, Packt offered him the opportunity to write a book about learning C# for Unity. He jumped on it.

I want to thank my daughter Emily Norton, the artist in the family, for helping me with the graphic's design.

About the Reviewers

Gaurav Garg was born in Delhi. He is a Computer Applications graduate from Indira Gandhi University and has passed his higher secondary from the CBSE Board. During his under- graduate studies, he started his career as an indie game programmer, but didn't gain success because of a lack of or say, no experience. After this, he learnt that passion is not the only thing for geting success; experience matters a lot. Then he joined Isis Design Service as a game programmer, where he published a few iOS titles and one web-based game. He worked there for a year and a half. Then, he moved to Jump Games, Pune, and worked on a few good game titles such as Realsteal and Dancing with the Stars. Now, he works for Mr Manvender Shukul in Lakshya Digital Pvt ltd. and has been there since the past year.

He hasn't reviewed a book before, but one of his articles was published in Game Coder Magazine. The article was on Unity3D. You can download the article from his personal website, `http://gauravgarg.com/`.

I would like to thanks my parents who taught me the value of hard work and an education.

I need to thanks my friends, particularly Manjith and Vibhash, who always took the time to listen, even when I was just complaining. They always are my best supporters and advisors.

Finally, I would like to thank Harshit who gives me this opportunity.

Kristian Hedeholm studied Computer Science at Aarhus University and now works as a game programmer at Serious Games Interactive in Copenhagen, Denmark. Since Kristian joined the game industry back in 2009, he has worked on a couple of released casual games. In addition to this, he is also the chairman of an association called Young Game Developers, which aims to spread information about game development among children and teenagers. In the future, Kristian will use his "computer mind" to develop artificial intelligence and dynamic difficulty adjustment systems for computer games.

When Kristian isn't developing games, teaching others to develop games, or playing games himself, he thinks about them a lot!

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface 1

Chapter 1: Discovering Your Hidden Scripting Skills 7

Prerequisite knowledge for using this book **8**
Dealing with scriptphobia **8**
Teaching behaviors to GameObjects 9
Choosing to use C# instead of UnityScript **10**
Reason 1 for choosing C# – vast amount of documentation on the Internet 10
Reason 2 for choosing C# – flexibility to use Unity scripts and regular C# code files 10
Reason 3 for choosing C# – coding rules are specific 11
Maneuvering around Unity's documentation **11**
Time for action – opening the Reference Manual documentation for the transform Component **11**
Time for action – opening the scripting reference documentation for the transform component **12**
Are we really supposed to know all that stuff? 13
What is all that information? 13
Working with C# script files **14**
Time for action – create a C# script file **14**
Introducing the MonoDevelop code editor 15
Syncing C# files between MonoDevelop and Unity 15
Time for action – opening LearningScript in MonoDevelop **15**
Watching for a possible "gotcha" when creating script files in Unity 16
Fixing sync if it isn't working properly 16
Summary **17**

Chapter 2: Introducing the Building Blocks for Unity Scripts 19

Using the term method instead of function **20**
Understanding what a variable does in a script **20**

Naming a variable 21
A variable name is just a substitute for a value 21
Time for action – creating a variable and seeing how it works 22
Time for action – changing the number 9 to a different number 23
Using a method in a script 24
What is a method? 24
Time for action – learning how a method works 24
What's in this script file? 25
Method names are substitutes too 25
Introducing the class 27
By using a little Unity magic, a script becomes a Component 28
A more technical look at the magic 28
Even more Unity magic 29
Components communicating using the Dot Syntax 29
What's with the dots? 30
Summary 30

Chapter 3: Getting into the Details of Variables 31
Writing C# statements properly 32
Understanding Component properties in Unity's Inspector 32
Variables become Component properties 33
Unity changes script and variable names slightly 33
Changing a property's value in the Inspector panel 33
Displaying public variables in the Inspector panel 34
Time for action – making a variable private 34
Naming your variables properly 35
Begin variable names with lowercase 36
Using multi-word variable names 36
Declaring a variable and its type 37
The most common built-in variable types 38
Time for action – assigning values while declaring the variable 38
Where you declare a variable is important 39
Variable scope – determining where a variable can be used 40
Summary 42

Chapter 4: Getting into the Details of Methods 43
Ending a method definition using curly braces 44
Using methods in a script 44
Naming methods properly 44
Begin method names with an uppercase letter 45
Using multi-word names for a method 45
Parentheses are part of the method name 45

Defining a method properly **45**
The minimum requirements for defining a method 46
Understanding parentheses – why are they there? 47
Time for action – adding code between the parentheses **47**
Specifying a method's parameters 48
How many parameters can a method have? 49
Calling a method **49**
Using arguments in the parentheses of a method 49
Returning a value from a method **50**
Time for action – returning a value from AddTwoNumbers() **51**
Calling a method is a logic detour 54
Using Unity's Update and Start methods **54**
The Start method is called one time 55
The Update method is called over and over and over… 55
Summary **56**
Chapter 5: Making Decisions in Code 57
Testing conditions with an if statement **58**
Testing if conditions are true or false 58
Time for action – create a couple of if statements **58**
Using the NOT operator to change the condition 60
Checking many conditions in an if statement 60
Time for action – create if statements with more than one condition to check **60**
Using an if-else statement to execute alternate code **63**
Time for action – add "else" to the if statement **63**
Making decisions based on user input **65**
Storing data in an array, a List, or a Dictionary **66**
Storing items in an array 66
Storing items in a List 68
Time for action – create a List of pony names **68**
Storing items in a Dictionary 73
Time for action – create a dictionary of pony names and keys **73**
Using a Collection Initializer to add items to a List or Dictionary 75
Time for action – adding ponies using a Collection Initializer **75**
Looping though lists to make decisions **77**
Using the foreach loop 77
Time for action – using foreach loops to retrieve data **77**
Using the for loop 81
Time for action – selecting a pony from a List using a for loop **81**
Using the while loop 84
Time for action – finding data and breakout of the while loop **84**
Summary **87**

Chapter 6: Using Dot Syntax for Object Communication 89
Using Dot Syntax is like addressing a letter 90
Simplifying the dots in Dot Syntax 90
Using access modifiers for variables and methods 91
Working with objects is a class act 91
Using Dot Syntax in a script 93
Accessing a Component's own variables and methods 93
Time for action – accessing a variable in the current Component 94
Accessing another Component on the current GameObject 97
Time for action – communicating with another Component on the Main Camera 97
Accessing other GameObjects and their Components 101
Time for action – creating two GameObjects and a new script 102
Accessing GameObjects using drag-and-drop versus writing code 108
Time for action – trying drag-and-drop to assign a GameObject 108
Summary 109
Chapter 7: Creating the Gameplay is Just a Part of the Game 111
Applying your new coding skills to a State Machine 112
Understanding the concepts of a State Machine 112
Benefits of by using a State Machine 113
Following the State Machine logic flow 114
Delegating game control to a State 114
Switching to another State when called to do so 115
Keeping track of the active State 116
Creating Components objects and C# objects 117
Unity creates Components behind the scenes 117
Instantiate a class to create an object 117
Time for action – creating a script and a class 118
Time for action – instantiating the BeginState class 121
Specifying a file's location with a namespace declaration 122
Locating code files with a using statement 123
Introducing the C# interface 123
The State Machine and the interface guarantee 124
Time for action – implementing an interface 124
Summary 128
Chapter 8: Developing the State Machine 129
Creating four State classes 130
Time for action – modyifing BeginState and add three more States 130
Setting up the StateManager controller 132
Studying an example of inheritance 134
Enter the IStateBase interface again 136

Time for action – modify StateManager **137**
Adding another State 142
Time for action – modifying PlayState to add another State **142**
Adding OnGUI to the StateManager class 143
Time for action – adding OnGUI() to StateManager **143**
Changing the active State and controlling the Scene **144**
Time for action – adding GameObjects and a button to the Scene **144**
Pausing the game Scene 145
Time for action – adding code to pause the game Scene **146**
Time for action – creating a timer in BeginState **147**
Changing Scenes **151**
Time for action – setting up another Scene **152**
Changing Scenes destroys the existing GameObjects 153
Keeping GameManager between scenes 153
Time for action – adding the Awake method to StateManager **154**
Changing the Scenes 155
Time for action – adding the code to change the Scenes **156**
Verifying the code of your classes 157
Summary **161**
Chapter 9: Start Building a Game and Get the Basic Structure Running 163
Easing into Unity's scripting documentation **164**
Reading the Unity Reference Manual first 165
Finding code examples in the Scripting Reference as needed 165
Setup the State Machine and add a Player GameObject **165**
Time for action – setting up nine States and three Scenes **167**
Calling the Restart method of the StateManager 169
Add a Player GameObject 170
Placing and using the Player Collider 171
Placing and using the Sphere Collider 171
Time for action - adding a Player GameObject **172**
Storing game data in its own script 172
Time for action – creating a GameData script **173**
Displaying the splash screens 174
Controlling the Player GameObject **179**
Time for action – rotating Player in SetupState **180**
Adding the Player Color option 183
Time for action – changing the color using GUI buttons **184**
Adding the Lives option for Player 187
Time for action – setting the Lives for Player **187**
Summary **191**

Chapter 10: Moving Around, Collisions, and Keeping Score 193
Visualizing the completed game 194
Switching to the first play State and playable scene 194
Loading Scene1 using code 195
Adding cameras for different viewing options 196
Time for action – setting up two additional cameras in the scene 196
Attaching scripts to the new cameras 199
Time for actioning – attach the LookAtPlayer camera script 199
Time for action – attaching the FollowingPlayer camera script 200
Moving the Player using Rigidbody physics 201
Time for action – adding a Rigidbody to the Player 202
Keeping score during the game 204
Initializing the scoring system 205
Keeping score in the Scene1 play State 207
Losing the game in Scene1 207
Winning the level in Scene1 208
Determining how to win or lose 210
Time for action – creating a good and bad prefab 210
Scoring for the win 210
Losing when Player crashes 211
Shooting projectiles at the orbs 212
Time for action – creating the EnergyPulse prefab 212
Shooting a single-shot EnergyPulse 214
Shooting rapid-fire EnergyPulses 214
The EnergyPulse is fired 215
Controlling EnergyPulse objects 216
Summary 219

Chapter 11: Summarizing Your New Coding Skills 221
Coding a Unity Project 222
Working with objects 222
Scratching the surface of C# programming 223
Looking at even more C# features 223
Looking at even more Unity features 224
Controlling the game with a State Machine 224
Using a State Machine is a design pattern choice 225
Using the State Machine at the GameObject level 225
Pulling all the little C# pieces together 226
Learning more after this book 226
Visit my favorite website for C# 227
Visit my favorite websites for Unity coding: 227
Summary 227

Appendix A: Initial State Machine files 229
- BeginState 230
- SetupState 230
- PlayStateScene1_1: (1 of 2 available States in Scene1) 231
- PlayStateScene1_2: (2 of 2 available States in Scene1) 232
- WonStateScene1 233
- LostStateScene1 234
- PlayStateScene2 235
- WonStateScene2 236
- LostStateScene2 236
- StateManager 237
- IStateBase 239

Appendix B: Completed code files for Chapters 9 and 10 241
- BeginState 241
- SetupState 242
- PlayStateScene1_1: (1 of 2 available States in Scene1) 244
- PlayStateScene1_2: (2 of 2 available States in Scene1) 246
- WonStateScene1 247
- LostStateScene1 248
- PlayStateScene2 249
- WonStateScene2 251
- LostStateScene2 252
- StateManager 253
- PlayerControl 254
- GameData 256
- LookAtPlayer 257
- FollowingPlayer 258
- EnergyPulsePower 258
- IStateBase 259

Appendix C: Pop Quiz Answers 261

Index 267

Preface

Unity has become one of the most popular game engines for developers, from the amateur hobbyist to the professional working in a large studio. Unity used to be considered a 3D tool, but with the release of Unity 4.3, it now has dedicated 2D tools. This will expand Unity's use even more.

Developers love its object-oriented drag-and-drop user interface which makes creating a game or interactive product so easy. Despite the visual ease of working in Unity, there is a need to understand some basic programming to be able to write scripts for GameObjects. For game developers that have any programming knowledge, learning how to write scripts is quite easy. For the the artist coming to Unity, creating the visual aspects of a game is a breeze, but writing scripts may appear to be a giant roadblock.

This book is for those with no concept of programming. I introduce the building blocks, that is, basic concepts of programming using everyday examples you are familiar with. Also, my approach to teaching is not what you will find in the typical programming book. In the end, you will learn the basics of C#, but I will spoon-feed you the details as they are needed.

I will take you through the steps needed to create a simple game, with the focus not being the game itself but on how the many separate sections of code come together to make a working game. I will also introduce the concept of a State Machine to organize code into simple, game controlling blocks. At the end, you will be saying "Wow! I can't believe how easy that was!"

What this book covers

Chapter 1, *Discovering Your Hidden Scripting Skills,* explains that the very first thing you need to do is overcome your perceived fear of writing scripts. You'll see that writing scripts is very similar to many of your daily routines. We also have a first look at Unity's scripting documentation. Finally, we see how to create a C# script file in Unity.

Chapter 2, *Introducing the Building Blocks for Unity Scripts,* explains that there are two primary building blocks for writing code, variables and methods. This chapter introduces the concepts of a variable and a method. With these two building blocks, we look into the concept of a "class," a container of variables and methods used to create Unity Components. Finally, communication between GameObjects is discussed by introducing Dot Syntax.

Chapter 3, *Getting into the Details of Variables,* explains using variables in detail. We see how they're used for storing data, and how the magic works to turn variables into Component properties which appear in the Unity Inspector panel.

Chapter 4, *Getting into the Details of Methods,* explains how methods perform the actions that take place on GameObjects. We see how to create and use methods in detail. We also look into two of Unity's most often used methods, the Start() method and the Update() method.

Chapter 5, *Making Decisions in Code,* explains that during gameplay, decisions have to be made about many things, just like you do in your daily life. We look at many of the ways choices are made and some of the common reasons for which decisions are required.

Chapter 6, *Using Dot Syntax for Object Communication,* shows us what Dot Syntax actually is, a simple address format to retrieve information or send information to other Components.

Chapter 7, *Creating the Gameplay is Just a Part of the Game,* shows that developing the gameplay is fun, but there are other parts needed to make a fully functional game. We look into some of the possible parts needed and how to organize all the parts by introducing the use of a State Machine.

Chapter 8, *Developing the State Machine,* creates a simple State Machine to show how it works, and see the simplicity it brings for controlling a game. We show how to change Scenes for a multi-level game and how to deal with GameObjects when changing to another scene.

Chapter 9, *Start Building a Game and Get the Basic Structure Running,* teaches us how to access and use Unity's Scripting Reference and the Reference Manual for the features we want. Then we begin creating a multi-level game using the state machine and three scenes. A Player GameObject is added and we learn how to control it.

Chapter 10, *Moving Around, Collisions, and Keeping Score,* shows how to move the Player around using Unity's physics system, and have cameras follow the Player's movements. We develop a GUI scoring system, start shooting projectiles at enemy objects, and see how to win or lose the game. Ultimately, we see how all the separate pieces of code come together and work together.

Chapter 11, *Summarizing Your New Coding Skills,* reviews the main points you learned about programming with C# and working with objects. I tell you about some of the C# and Unity features you may want to learn now that you understand the basics of C#. I will highlight the benefits of incorporating a state machine into your Unity projects. Finally, I present my favorite sources for further learning.

Appendix A, *Initial State machine files*, shows the initial code for the classes needed for changing States in our game. These State Machine classes are the starting point for organizing and adding game code.

Appendix B, *Completed code files for Chapters 9 and 10*, shows all the class and script files used for playing our completed game.

What you need for this book

You need the free version of Unity located at `http://unity3d.com/unity/download/`. The MonoDevelop code editor is included in the Unity installation.

Your computer will need to meet the minimum requirements for Unity as specified at `http://unity3d.com/unity/system-requirements.html`.

Windows: XP SP2 or later; Mac OS X "Snow Leopard" 10.6 or later. Note that Unity was not tested on server versions of Windows and OS X.

Graphics card with DirectX 9 level (shader model 2.0) capabilities. Any card made since 2004 should work.

Who this book is for

If you don't know anything about programming in general, writing code, writing scripts, or have no idea where to even begin, then this book is perfect for you. If you want to make games and need to learn how to write C# scripts or code, then this book is ideal for you.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple-choice questions intended to help you test your own understanding.

Have a go hero – heading

These practical challenges give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the include directive."

A block of code is set as follows:

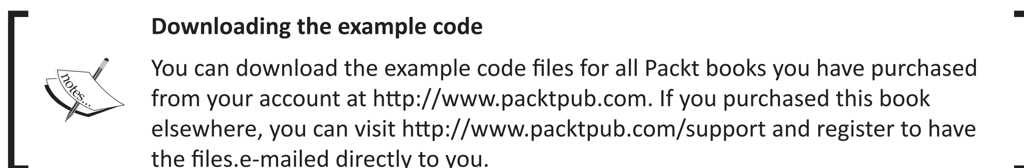
```
public BeginState (StateManager managerRef)
{
manager = managerRef;
if(Application.loadedLevelName != "Scene0")
Application.LoadLevel("Scene0");
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
if(instanceRef == null)
{
instanceRef = this;
DontDestroyOnLoad(gameObject);
}
```

Any command-line input or output is written as follows:

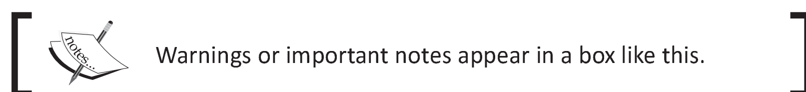
```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
    /etc/asterisk/cdr_mysql.conf
```



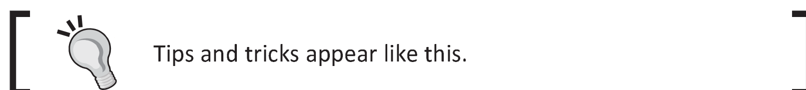
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. 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.

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. 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.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Discovering Your Hidden Scripting Skills

Computer programming is viewed by the average person as requiring long periods of training to learn skills that are totally foreign, and darn near impossible to understand. The word geek is often used to describe a person that can write computer code. The perception is that learning to write code takes great technical skill that is just so hard to learn. This perception is totally unwarranted. You already have the skills needed but don't realize it. Together we will crush this false perception you may have of yourself by refocusing, one step at a time, the knowledge you already possess to write Unity scripts.

In this chapter we shall:

- Deal with preconceived fears and concepts about scripts
- See why we should use C# instead of UnityScript
- Introduce Unity's documentation for scripting
- Learn how Unity and the MonoDevelop editor work together

Let's begin our journey by eliminating any anxiety about writing scripts for Unity, and become familiar with our scripting environment.

Prerequisite knowledge for using this book

Great news if you are a scripting beginner! This book is for those with absolutely no knowledge of programming. It is devoted to teaching the basics of C# with Unity.

However, some knowledge of Unity's operation is required. I will only be covering the parts of the Unity interface that are related to writing C# code. I am assuming that you know your way around Unity's interface, how to work with GameObjects in your **Scene**, and how to locate **Components** and view their **Properties** in the **Inspector**.

Dealing with scriptphobia

You've got Unity up and running, studied the interface, added some GameObjects to the Scene. Now you're ready to have those GameObjects move around, listen, speak, pick up other objects, shoot the bad guys, or anything else you can dream of. So you click on Play, and nothing happens. Well darn it all anyway.

You just learned a big lesson, all those fantastic, highly detailed GameObjects are dumber than a hammer. They don't know anything, and they sure don't know how to do anything.

So you proceed to read the Unity forums, study some scripting tutorials, maybe even copy and paste some scripts to get some action going when you press Play. That's great, but then you realize you don't understand anything in the scripts you've copied. Sure, you probably recognize the words, but you fail to understand what those words do or mean in a script. It feels like gibberish.

You look at the code, your palms get sweaty, and you think to yourself, "Geez, I'll never be able to write scripts!" Perhaps you have **scriptphobia**: the fear of not being able to write instructions (I made that up). Is that what you have?

The fear that you cannot write down instructions in a coherent manner? You may believe you have this affliction, but you don't. You only think you do.

The basics of writing code are quite simple. In fact, you do things every day that are just like the steps executed in a script. For example, do you know how to interact with other people? How to operate a computer? Do you fret so much about making a baloney sandwich that you have to go to an online forum and ask how to do it?

Of course you don't. In fact, you know these things as "every day routines", or maybe as habits. Think for a moment, do you have to consciously think about these routines you do every day? Probably not. After you do them over and over, they become automatic.

The point is, you do things everyday following sequences of steps. Who created these steps you follow? More than likely you did, which means you've been scripting your whole life. You just never had to write down the steps, for your daily routines, on a piece of paper before doing them. You could write the steps down if you really wanted to, but it takes too much time and there's no need. But you do, in fact, know how to. Well, guess what? To write scripts, you only have to make one small change, start writing down the steps. Not for yourself but for the world you're creating in Unity.

So you see, you are already familiar with the concept of dealing with scripts. Most beginners to Unity easily learn their way around the Unity interface, how to add assets, and work in the Scene and Hierarchy windows. Their primary fear, and roadblock, is their false belief that scripting is too hard to learn.

Relax! You now have this book. I am going to get really basic in the beginning chapters. Call them baby-steps if you want, but you will see that scripting for Unity is similar to doing things you already do everyday. I'm sure you will have many "Ah-Ha" moments as you learn and overcome your unjustified fears and beliefs.

Teaching behaviors to GameObjects

You have Unity because you want to make a game or something interactive. You've filled your game full of dumb GameObjects. What you have to do now is be their teacher. You have to teach them everything they need to know to live in this make-believe world. This the part where you have to write down the instructions so that your GameObjects can be smarter.

Here's a quote from the Unity Manual:

> *The* **behavior** *of GameObjects is controlled by the* **Components** *that are attached to them... Unity allows you to create your own* **Components** *using scripts.*

Notice that word, behavior. It reminds me of a parent teaching a child proper behavior. This is exactly what we are going to do when we write scripts for our GameObjects, we're teaching them the behaviors we want them to have. The best part is, Unity has provided a big list of all the behaviors we can give to our GameObjects. This list of behaviors is documented in the Scripting Reference.

This means we can pick and chose, from this list of behaviors anything we want a GameObject to do. Unity has done all the hard work of programming all these behaviors for you. All we need to do is use a little code to tie into these behaviors. Did you catch that? Unity has already created the behaviors, all we have to do is supply a little bit of C# code to apply these behaviors to our GameObjects. Now really, how difficult can it be since Unity has already done most of the programming?

Choosing to use C# instead of UnityScript

So why choose C# to create this code? This maybe after-the-fact information for you if you've already acquired this book and chosen to use C#, but these are valuable points to know anyway:

Reason 1 for choosing C# – vast amount of documentation on the Internet

Have a look at the following bullet list, it will help you understand the reason for choosing C#:

- C# is a well known and a heavily used programming language developed by Microsoft for creating Windows application and web-based applications. If you ever need to know anything about C#, simply do a search on the Internet.
- UnityScript is just a scripting language designed specifically for Unity. It's similar to JavaScript, yet it isn't. You may be able to search for JavaScript solutions on the web, but the code may or may not work within the confines of Unity without modification, if at all.
- Why start off learning a limited scripting language, specific only to Unity, when you can use C#, a true programming language, and find information everywhere?
- Who knows, once you see how easy C# is, maybe you might decide to develop for Windows or the Web some day. You'll already have the basics of C#.
- Once you learn C#, you'll pretty much know UnityScript, too.

Reason 2 for choosing C# – flexibility to use Unity scripts and regular C# code files

- Any C# files you have in your Unity Project folder, that are not Unity scripts, are accessible without the need of attach them to GameObjects.
- The State Machine project we will create for this book makes use of C# code files that are not attached to any GameObject.
- I'm not saying you can't create a State Machine by using UnityScript. It's just so much easier with C#. Every UnityScript file has to be attached to a GameObject to work and be accessible to other scripts. C# overcomes this necessity.

Reason 3 for choosing C# – coding rules are specific

- C# is known as a strictly-typed language. What does this means to you?
- As you write code, Unity will catch coding errors immediately. Learning a subject is always easier when the rules are specific, and not some fuzzy "you can if you want to" kind of logic.
- UnityScript is not a strictly-typed language. You have the potential to write code that is not valid, but Unity won't catch the errors until you press Play.
- Finding mistakes as you write the code is so much easier than having to deal with them when a user has found them when they're playing the game.
- Please be aware, it is easy to force UnityScript to be strictly-typed, but if you're going to do that, then you may as well be using C# anyway, which brings us back to Reason 1.

Maneuvering around Unity's documentation

When we begin writing scripts, we will be looking at Unity's documentation quite often, so it's beneficial to know how to access the information we need. For an overview of a topic we'll use the **Reference Manual**. For specific coding details and examples we'll use the **Scripting Reference**.

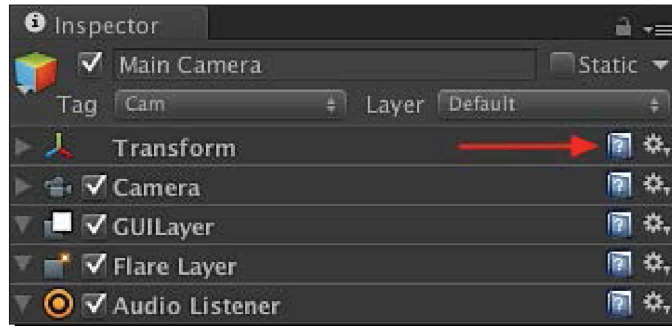
When you look at the code examples in the **Scripting Reference**, they probably won't make sense to you, which is expected at this point. In the beginning chapters, as I teach you the basics of programming, it will be necessary for me to use a few things in the **Scripting Reference** such as displaying some output to Unity's **Console**. For now, just copy the code I use because you will be learning the detail of it later.

Time for action – opening the Reference Manual documentation for the transform Component

To get a feel for accessing Unity's documentation from within Unity, we'll use the **Main Camera** to demonstrate. Every GameObject in a Scene has a **Transform** Component, so we'll look at the documentation for **Transform** in the **Reference Manual** and the **Scripting Reference**. Getting to the information is pretty easy. Click on the tiny book icon with the question mark.

1. In the **Hierarchy** tab, select the **Main Camera**.

2. Click on the book icon for the **Transform**.



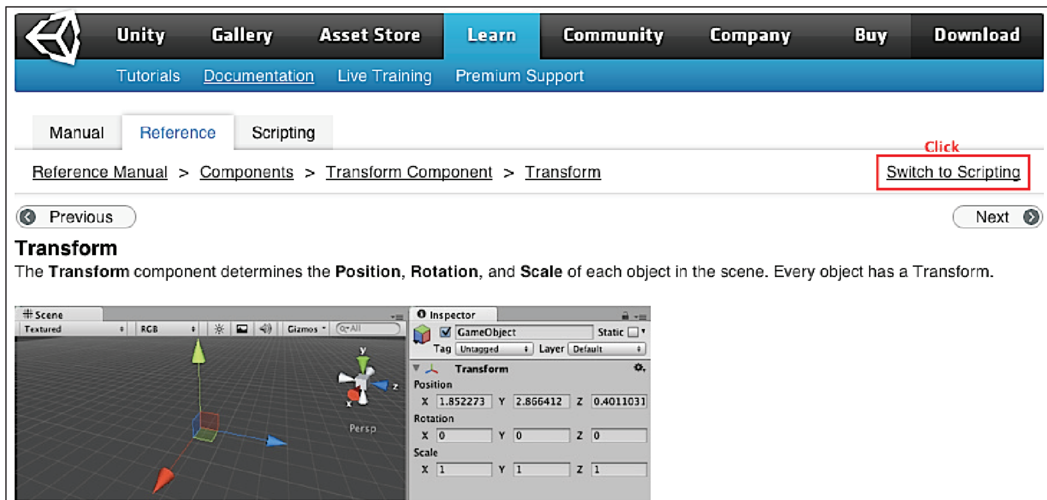
What just happened?

The web browser opened the **Reference Manual** showing information about Transform.

Time for action – opening the scripting reference documentation for the transform component

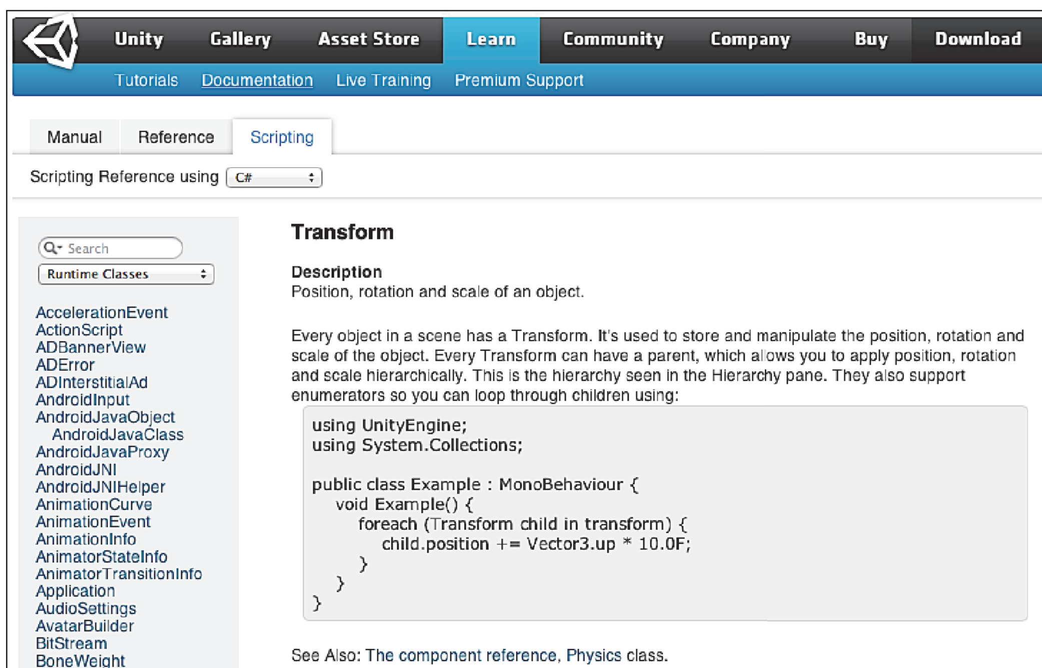
From the **Reference Manual**, we'll now open the **Scripting Reference** documentation for the **Transform Component**.

1. Click the link **Switch to Scripting** in the upper right-hand side of the browser window as shown in the following screenshot:



What just happened?

The **Transform** page in the **Scripting Reference** opens in the web browser as shown in the following screenshot:



Are we really supposed to know all that stuff?

Actually, no. The whole reason for why the Scripting Reference exist is so we can look for information as we need it. Which will actually happen us to remember the code we do over and over, just like our other daily routines and habits.

What is all that information?

The previous screenshot shows a description and some sample code which probably doesn't mean much right now. Fear not! You'll eventually be able to look at that and say, "Hey, I know what that means!"

Working with C# script files

Until you learn some basic programming concepts, it's too early to study how scripts work, but we still need to know how to create one.

There are several ways to create a script file using Unity:

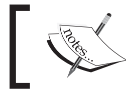
- In the menu navigate to **Assets** | **Create** | **C# Script**

  Or

- In the **Project** tab navigate to **Create** | **C# Script**

  Or

- In the **Project** tab right-click , from the pop-up menu navigate to **Create** | **C# Script**

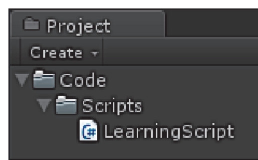


From now on, when I tell you to create a C# script, please use which ever method you prefer.

Time for action – create a C# script file

As our Unity project progresses, we will have several folders to organize and store all of our C# files.

1. Create a new Unity project and name it as `State Machine`.
2. Right-click on in the **Project** tab and create a folder named `Code`.
3. Right-click on the `Code` folder and a create a folder named `Scripts`.
4. In the `Scripts` folder, create a `C# Script`.
5. Immediately rename `NewBehaviourScript` to `LearningScript`.



What just happened?

We created one of the `Code` subfolders, named `Scripts`, that we will be using to organize our C# files. This folder will contain all of our Unity script files. Later we will create other C# file folders.

We also used Unity to create a C# script file named `LearningScript.cs`.

Introducing the MonoDevelop code editor

Unity uses an external editor to edit its C# scripts. Even though Unity can create a basic starter C# script for us, we still have to edit the script using the **MonoDevelop** code editor that's included with Unity.

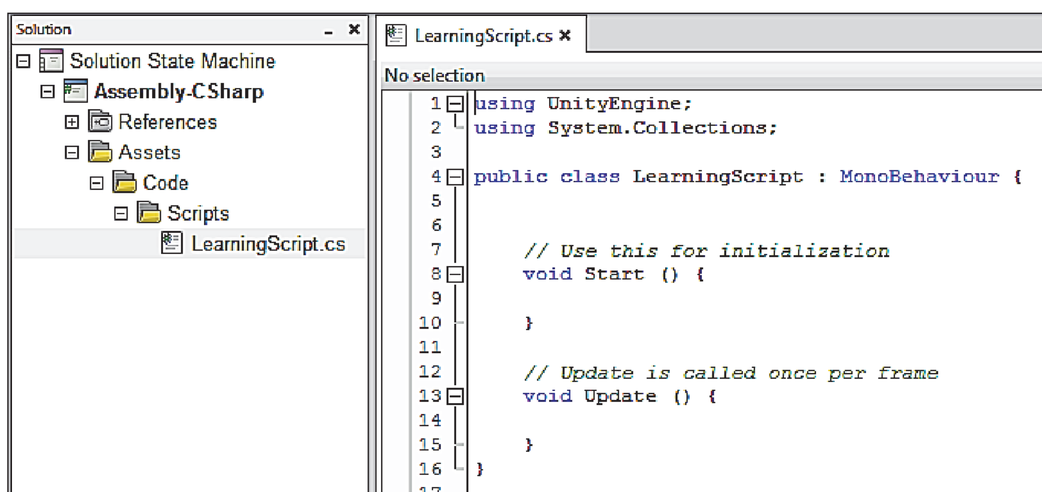
Syncing C# files between MonoDevelop and Unity

Since Unity and MonoDevelop are separate applications, Unity will keep MonoDevelop and Unity synchronized with each other. This means that if you add, delete, or change a script file in one application, the other application will see the changes automatically.

Time for action – opening LearningScript in MonoDevelop

Unity will synchronize with MonoDevelop the first time you tell Unity to open a file for editing. The simplest way to do this is just double-click on `LearningScript` in the `Scripts` folder.

1. In Unity's Project tab, double-click on `LearningScript`:



What just happened?

MonoDevelop started with `LearningScript` open, ready to edit.

Watching for a possible "gotcha" when creating script files in Unity

Notice line 4 in the previous screenshot:

```
public class LearningScript : MonoBehaviour
```

The class name `LearningScript` is the same as the file name `LearningScript.cs`. This is a requirement. You probably don't know what a class is yet, that's ok. Just remember that the file name and the class name must be the same.

When you create a C# script file in Unity, the filename, in the **Project** tab, is in Edit mode, ready to be renamed. Please rename it right then and there. If you rename the script later, the filename and the class name won't match. The filename would change, but line 4 would be this:

```
public class NewBehaviourScript : MonoBehaviour
```

This can easily be fixed in MonoDevelop by changing `NewBehaviourScript` on line 4 to the same name as the filename, but it's much simpler to do the renaming in Unity immediately.

Fixing sync if it isn't working properly

So what happens when Murphy's Law strikes and syncing just doesn't seem to be working correctly? Should the two apps somehow get out-of-sync as you switch back-and-forth between the them, for whatever reason, do this:

- Right-click on Unity's **Project** window and select **Sync MonoDevelop Project**. MonoDevelop will re-sync with Unity.

Pop quiz – dealing with scripts

Q1. As a beginner, what's the biggest obstacle to be overcome to be able to write C# code?

Q2. The Scripting Reference supplies example code and a short description of what the code does. What do you use to get full detailed descriptions of Unity's Components and features?

Q3. The Scripting Reference is a large document. How much it should you know before attempting to write any scripts?

Q4. When creating a script file in Unity, when is the best time to name the script file?

Summary

This chapter tried to put you at ease about writing scripts for Unity. You do have the ability to write down instructions which is all a script is, a sequence of instructions. We saw how simple it is to create a new script file. You probably create files on your computer all the time. We saw how to easily bring up Unity's documentation. Finally we had a look at the MonoDevelop editor. None of this was complicated. In fact, you probably use apps all the time that do similar things. Bottom line, there's nothing to fear here.

Alright, let's start off *Chapter 2, Introducing the Building Blocks for Unity Scripts by having an* introductory look at the building blocks of programming we'll be using: variables, methods, Dot Syntax, and the class. Don't let these terms scare you. The concepts behind each one of these are similar to things you do often, perhaps every day.

2

Introducing the Building Blocks for Unity Scripts

A programming language such as C# can appear to be very complicated at first but in reality, there are two parts that form its foundation. These parts are variables and methods. Therefore, understanding these critical parts is a prerequisite for learning any of the other features of C#. Being as critical as they are, they are very simple concepts to understand. Using these variable and method foundation pieces, we'll be introduced to the C# building blocks used to create Unity scripts.

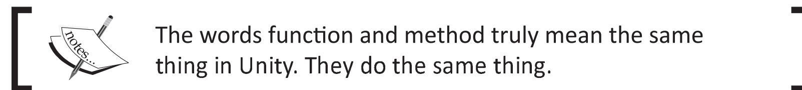
For those people who get sweaty palms just thinking of the word script, wipe your hands and relax. In this chapter, I'm going to use terms that are already familiar to you to introduce the building blocks of programming. The following are the concepts introduced in this chapter:

- Using variables in a script
- Using methods in a script
- The class which is a container for variables and methods
- Turning a script into a Component
- Components communicating using the Dot Syntax

Let's have a look at these primary concepts.

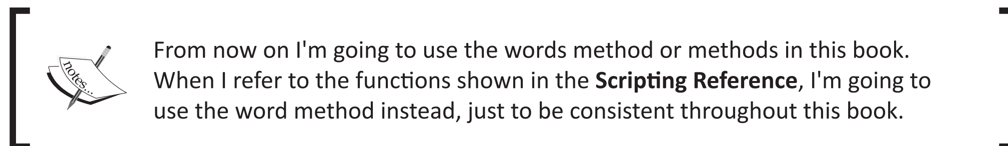
Using the term method instead of function

You are constantly going to see the words **function** and **method** used everywhere as you learn Unity.



The words function and method truly mean the same thing in Unity. They do the same thing.

Since you are studying C#, and C# is an **Object-Oriented Programming** (**OOP**) language, I will use the word "method" throughout this book, just to be consistent with C# guidelines. It makes sense to learn the correct terminology for C#. Also, UnityScript and Boo are OOP languages. The authors of the **Scripting Reference** probably should have used the word method instead of function in all documentation.



From now on I'm going to use the words method or methods in this book. When I refer to the functions shown in the **Scripting Reference**, I'm going to use the word method instead, just to be consistent throughout this book.

Understanding what a variable does in a script

What is a variable? Technically, it's a tiny section of your computer's memory that will hold any information you put there. While a game runs, it keeps track of where the information is stored, the value kept there, and the type of the value. However, for this chapter, all you need to know is how a variable works in a script. It's very simple.

What's usually in a mailbox, besides air? Well, usually there's nothing but occasionally there is something in it. Sometimes there's money (a paycheck), bills, a picture from aunt Mabel, a spider, and so on. The point is what's in a mailbox can vary. Therefore, let's call each mailbox a variable instead.

Naming a variable

Using the picture of the country mailboxes, if I asked you to see what is in the mailbox, the first thing you'd ask is which one? If I said in the Smith mailbox, or the brown mailbox, or the round mailbox, you'd know exactly which mailbox to open to retrieve what is inside. Similarly, in scripts, you have to name your variables with a unique name. Then I can ask you what's in the variable named `myNumber`, or whatever cool name you might use.

A variable name is just a substitute for a value

As you write a script and make a variable, you are simply creating a placeholder or a substitute for the actual information you want to use. Look at the following simple math equation: 2 + 9 = 11

Simple enough. Now try the following equation: 11 + `myNumber` = ???

There is no answer to this yet. You can't add a number and a word. Going back to the mailbox analogy, write the number 9 on a piece of paper. Put it in the mailbox named `myNumber`. Now you can solve the equation. What's the value in `myNumber`? The value is 9. So now the equation looks normal: 11 + 9 = 20

The `myNumber` variable is nothing more than a named placeholder to store some data (information). So anywhere you would like the number 9 to appear in your script, just write `myNumber`, and the number 9 will be substituted.

Although this example might seem silly at first, variables can store all kinds of data that is much more complex than a simple number. This is just a simple example to show you how a variable works.

Time for action – creating a variable and seeing how it works

Let's see how this actually works in our script. Don't be concerned about the details of how to write this, just make sure your script is the same as the script shown in the next screenshot.

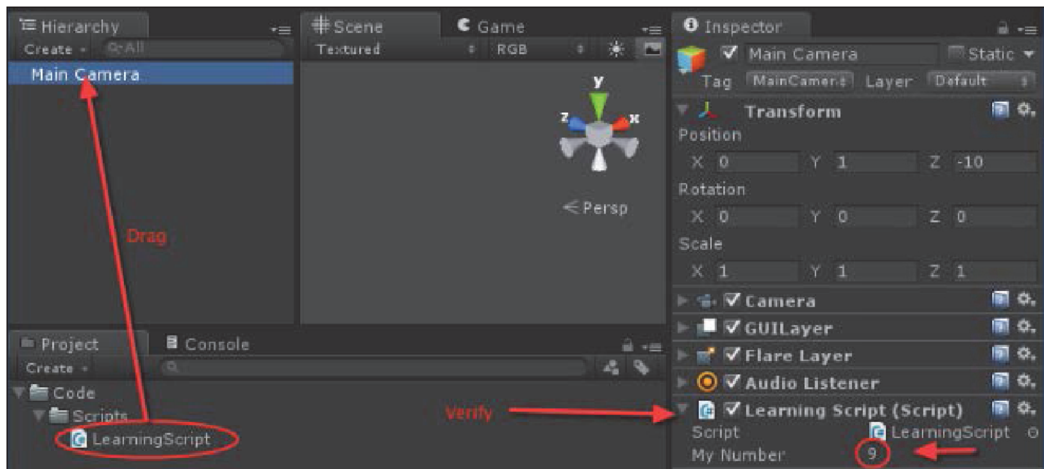
1. In the Unity **Project** panel, double-click on `LearningScript`.
2. In MonoDevelop, write the lines 6, 11, and 13 from the next screenshot.
3. Save the file.

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class LearningScript : MonoBehaviour
5  {
6      public int myNumber = 9;
7
8      // Use this for initialization
9      void Start ()
10     {
11         Debug.Log(2 + 9);
12
13         Debug.Log(11 + myNumber);
14     }
15
16     // Update is called once per frame
17     void Update ()
18     {
19
20     }
21 }
```

To make this script work, it has to be attached to a GameObject. Currently, in our **State Machine** project we only have one GameObject, the **Main Camera**. This will do nicely since this script doesn't affect the **Main Camera** in any way. The script simply runs by virtue of it being attached to a GameObject.

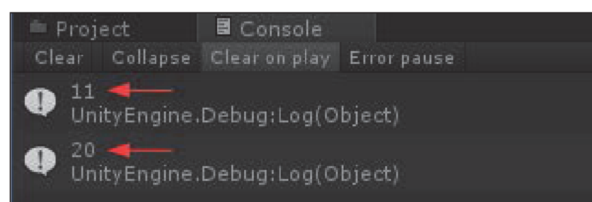
1. Drag `LearningScript` onto the **Main Camera**.
2. Select **Main Camera** so that it appears in the **Inspector** panel.
3. Verify whether `LearningScript` is attached.
4. Open the Unity **Console** panel to view the output of the script.
5. Click on **Play**.

The preceding steps are shown in the following screenshot:



What just happened?

In the following **Console** panel is the result of our equations. As you can see, the equation on line 13 worked by substituting the number 9 for the `myNumber` variable:



Time for action – changing the number 9 to a different number

Since `myNumber` is a variable, the value it stores can vary. If we change what is stored in it, the answer to the equation will change too. Follow the ensuing steps:

1. Stop the game and change **9** to **19**.
2. Notice that when you restart the game, the answer will be **30**.

What just happened?

You learned that a variable works by simple process of substitution. There's nothing more to it than that.

We didn't get into the details of the wording used to create `myNumber`, or the types of variables you can create, but that wasn't the intent. This was just to show you how a variable works. It just holds data so you can use that data elsewhere in your script. We'll get into the finer details of variables in *Chapter 3*, *Variables in Detail*.

Have a go hero – changing the value of myNumber

In the **Inspector** panel, try changing the value of `myNumber` to some other value, even a negative value. Notice the change in answer in the **Console**.

Using a method in a script

Methods are where the action is and where the tasks are performed. Great, that's really nice to know but what is a method?

What is a method?

When we write a script, we are making lines of code that the computer is going to execute, one line at a time. As we write our code, there will be things we want our game to execute more than once. For example, we can write a code that adds two numbers. Suppose our game needs to add the two numbers together a hundred different times during the game. So you say, *"Wow, I have to write the same code a hundred times that adds two numbers together. There has to be a better way."*

Let a method take away your typing pain. You just have to write the code to add two numbers once, and then give this chunk of code a name, such as `AddTwoNumbers()`. Now, every time our game needs to add two numbers, don't write the code over and over, just call the `AddTwoNumbers()` method.

Time for action – learning how a method works

We're going to edit `LearningScript` again. In the following screenshot, there are a few lines of code that look strange. We are not going to get into the details of what they mean in this chapter. We will discuss that in *Chapter 4*, *Getting into the Details of Methods*. Right now, I am just showing you a method's basic structure and how it works:

1. In MonoDevelop, select `LearningScript` for editing.
2. Edit the file so that it looks exactly like the following screenshot.

3. Save the file.

```
 1  using UnityEngine;
 2  using System.Collections;
 3
 4  public class LearningScript : MonoBehaviour
 5  {
 6      public int number1 = 2;
 7      public int number2 = 9;
 8
 9      void Start ()
10      {
11
12      }
13
14      void Update ()
15      {
16          if(Input.GetKeyUp(KeyCode.Return))
17              AddTwoNumbers(); <- Calling the method
18      }
19
20      void AddTwoNumbers()
21      {
22          Debug.Log(number1 + number2);
23      }                                        The Method
24  }
```

What's in this script file?

In the previous screenshot, lines 6 and 7 will look familiar to you; they are variables just as you learned in the previous section. There are two of them this time. These variables store the numbers that are going to be added.

Line 16 may look very strange to you. Don't concern yourself right now with how this works. Just know that it's a line of code that lets the script know when the *Return*/*Enter* key is pressed. Press the *Return*/*Enter* key when you want to add the two numbers together.

Line 17 is where the `AddTwoNumbers()` method gets called into action. In fact, that's exactly how to describe it. This line of code calls the method.

Lines 20, 21, 22, and 23 make up the `AddTwoNumbers()` method. Don't be concerned about the code details yet. I just want you to understand how calling a method works.

Method names are substitutes too

You learned that a variable is a substitute for the value it actually contains. Well, a method is no different.

Take a look at line 20 from the previous screenshot:

```
void AddTwoNumbers ()
```

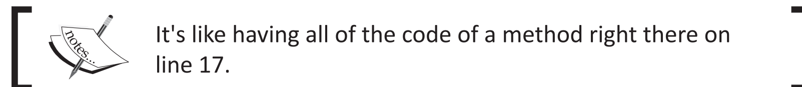
The `AddTwoNumbers()` is the name of the method. Like a variable, `AddTwoNumbers()` is nothing more than a named placeholder in the memory, but this time it stores some lines of code instead. So anywhere we would like to use the code of this method in our script, just write `AddTwoNumbers()`, and the code will be substituted.

Line 21 has an opening curly-brace and line 23 has a closing curly-brace. Everything between the two curly-braces is the code that is executed when this method is called in our script.

Look at line 17 from the previous screenshot:

```
AddTwoNumbers();
```

The method name `AddTwoNumbers()` is called. This means that the code between the curly-braces is executed.



It's like having all of the code of a method right there on line 17.

Of course, this `AddTwoNumbers()` method only has one line of code to execute, but a method could have many lines of code.

Line 22 is the action part of this method, the part between the curly-braces. This line of code is adding the two variables together and displaying the answer to the Unity **Console**. Then, follow the ensuing steps:

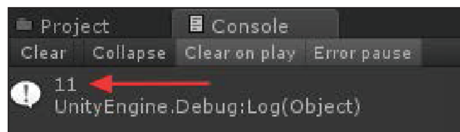
1. Go back to Unity and have the **Console** panel showing.
2. Now click on **Play**.

What just happened?

Oh no! Nothing happened!

Actually, as you sit there looking at the blank **Console** panel, the script is running perfectly, just as we programmed it. Line 16 in the script is waiting for you to press the *Return*/*Enter* key. Press it now.

And there you go! The following screenshot shows you the result of adding two variables together that contain the numbers 2 and 9:



Line 16 waited for you to press the *Return*/*Enter* key. When you do this, line 17 executes which calls the `AddTwoNumbers()` method. This allows the code block of the method, line 23, to add the the values stored in the variables `number1` and `number2`.

Have a go hero – changing the output of the method

While Unity is in the **Play** mode, select the **Main Camera** so its Components show in the **Inspector**. In the **Inspector** panel, locate **Learning Script** and its two variables. Change the values, currently **2** and **9**, to different values. Make sure to click your mouse in the **Game** panel so it has focus, then press the *Return*/*Enter* key again. You will see the result of the new addition in the **Console**.

You just learned how a method works to allow a specific block of code to to be called to perform a task.

We didn't get into any of the wording details of methods here, this was just to show you fundamentally how they work. We'll get into the finer details of methods in *Chapter 4*, *Getting into the Details of Methods*.

Introducing the class

The **class** plays a major role in Unity. In fact, what Unity does with a class a little piece of magic when Unity creates Components.

You just learned about variables and methods. These two items are the building blocks used to build Unity scripts. The term script is used everywhere in discussions and documents. Look it up in the dictionary and it can be generally described as written text. Sure enough, that's what we have. However, since we aren't just writing a screenplay or passing a note to someone, we need to learn the actual terms used in programming.

Unity calls the code it creates a C# script. However, people like me have to teach you some basic programming skills and tell you that a script is really a class.

In the previous section about methods, we created a class (script) called `LearningScript`. It contained a couple of variables and a method. The main concept or idea of a class is that it's a container of data, stored in variables, and methods that process that data in some fashion. Because I don't have to constantly write class (script), I will be using the word script most of the time. However, I will also be using class when getting more specific with C#. Just remember that a script is a class that is attached to a GameObject.

In *Chapter 7*, *Creating the Gameplay is Just a Part of the Game*, we will be creating some classes for a State Machine. These classes will not be attached to any GameObjects, so I won't be calling them scripts.

By using a little Unity magic, a script becomes a Component

While working in Unity, we wear the following two hats:

- A Game-Creator hat
- A Scripting (programmer) hat

When we first wear our Game-Creator hat, we will be developing our Scene, selecting GameObjects, and viewing Components; just about anything except writing our scripts.

When we put our Scripting hat on, our terminology changes as follows:

- We're writing code in scripts using MonoDevelop
- We're working with variables and methods

The magic happens when you put your Game-Creator hat back on and attach your script to a GameObject. Wave the magic wand — ZAP — the script file is now called a Component, and the public variables of the script are now the properties you can see and change in the **Inspector** panel.

A more technical look at the magic

A script is like a blueprint or a written description. In other words, it's just a single file in a folder on our hard drive. We can see it right there in the **Projects** panel. It can't do anything just sitting there. When we tell Unity to attach it to a GameObject, we haven't created another copy of the file, all we've done is tell Unity we want the behaviors described in our script to be a Component of the GameObject.

When we click on the **Play** button, Unity loads the GameObject into the computer's memory. Since the script is attached to a GameObject, Unity also has to make a place in the computer's memory to store a Component as part of the GameObject. The Component has the capabilities specified in the script (blueprint) we created.

Even more Unity magic

There's some more magic you need to be aware of. The scripts inherit from `MonoBehaviour`.

For beginners to Unity, studying C# inheritance isn't a subject you need to learn in any great detail, but you do need to know that each Unity script uses inheritance. We see the code in every script that will be attached to a GameObject. In `LearningScript`, the code is on line 4:

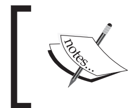
```
public class LearningScript : MonoBehaviour
```

The colon and the last word of that code means that the `LearningScript` class is inheriting behaviors from the `MonoBehaviour` class. This simply means that the MonoBehaviour class is making few of its variables and methods available to the LearningScript class. It's no coincidence that the variables and methods inherited look just like some of the code we saw in the **Unity Scripting Reference**.

The following are the two inherited behaviors in the `LearningScript`:

Line 9:: `void Start ()`

Line 14: `void Update ()`



> The magic is that you don't have to call these methods, Unity calls them automatically. So the code you place in these methods gets executed automatically.

Have a go hero – finding Start and Update in the Scripting Reference

Try a search on the **Scripting Reference** for **Start** and **Update** to learn when each method is called by Unity and how often.

Also search for `MonoBehaviour`. This will show you that since our script inherits from `MonoBehaviour`, we are able to use the `Start()` and `Update()` methods.

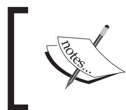
Components communicating using the Dot Syntax

Our script has variables to hold data, and our script has methods to allow tasks to be performed. I now want to introduce the concept of communicating with other GameObjects and the Components they contain. Communication between one GameObject's Components and another GameObject's Components using Dot Syntax is a vital part of scripting. It's what makes interaction possible. We need to communicate with other Components or GameObjects to be able to use the variables and methods in other Components.

What's with the dots?

When you look at the code written by others, you'll see words with periods separating them. What the heck is that? It looks complicated, doesn't it. The following is an example from the Unity documentation:

```
transform.position.x
```



Don't concern yourself with what the preceding code means as that comes later, I just want you to see the dots.

That's called the Dot Syntax. The following is another example. It's the fictitious address of my house: USA.Vermont.Essex.22MyStreet

Looks funny, doesn't it? That's because I used the syntax (grammar) of C# instead of the post office. However, I'll bet if you look closely, you can easily figure out how to find my house. We'll get into much more Dot Syntax detail in *Chapter 6*, *Using Dot Syntax for Object Communication.*

Pop quiz – knowing the C# building blocks

Q1. What is the purpose of a variable in a script?

Q2. What is the purpose of a method in a script?

Q3. How does a script become a Component?

Q4. What is the purpose of Dot Syntax?

Summary

This chapter introduced you to the basic concepts of variables, methods, and Dot Syntax. These building blocks are used to create scripts and classes. Understanding how these building blocks work is critical so you don't feel you're not getting it.

We discovered that a variable name is a substitute for the value it stores; a method name is a substitute for a block of code; when a script or class is attached to a GameObject, it becomes a Component. The Dot Syntax is just like an address to locate GameObjects and Components.

With these concepts under your belt, we can proceed to learn the details of the sentence structure, the grammar, and the syntax used to work with variables, methods, and the Dot Syntax. In the next chapter we will learn about the details of using variables.

3
Getting into the Details of Variables

Initially, computer programming appears difficult to beginners due to the fact how words are used in code. It's not the actual words that cause the problem because, for the most part, many of the words are the same words that we use in our everyday life. C# is not a foreign language. The main problem is that the words simply don't read like the typical sentences we are all used to. You know how to say the words and you know how to spell the words. What you don't know is where and why you need to put them in that crazy looking grammar, that is, the syntax that makes up a C #statement.

In this chapter, we will learn some of the basic rules for writing a C# statement. We will also be introduced to many of the words that C# uses and the proper placement of these words in the C# statements when we create our variables.

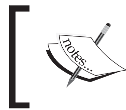
In this chapter we will cover the following topics:

- Writing C# statements properly
- Using C# syntax to write variable statements
- The GameObject Component's properties
- Using public variables for the Unity Inspector panel
- Naming a variable properly
- Declaring a variable for the type of data it will store

Ok, let's learn some programming grammar, otherwise known as C# syntax.

Writing C# statements properly

When you do normal writing, it's in the form of a sentence with a period used to end the sentence. When you write a line of code, it's called a statement with a semi-colon used to end the statement.



The reason a statement ends with a semi-colon is so that Unity knows when the statement ends. A period can't be used because they are used in the Dot Syntax.

The code for a C# statement does not have to be on a single line as shown in the following example:

```
public int number1 = 2;
```

The statement can be on several lines. Whitespace and carriage returns are ignored, so if you really want to, you can write it as follows:

```
public
int
number1
=
2;
```

But I recommend you to not write your code like this because it's terrible reading code formatted like the preceding code. However, there will be times that you'll have to write long statements that will be longer than one line. Unity won't care. It just needs to see the semi-colon at the end.

Understanding Component properties in Unity's Inspector

GameObjects have some Components that make them behave in a certain way. For instance, select **Main Camera** and look at the **Inspector** panel. One of the Components is the **Camera**. Without that Component, it will cease being a camera. It would still be a GameObject in your scene, just no longer a functioning camera.

Variables become Component properties

Any Component of any GameObject is just a script that defines a class, whether you wrote the script or the Unity's programmer did. We just aren't supposed to edit the scripts that Unity wrote. This means that all the properties we see in **Inspector** are just variables of some type. They simply store data that will be used by some methods.

Unity changes script and variable names slightly

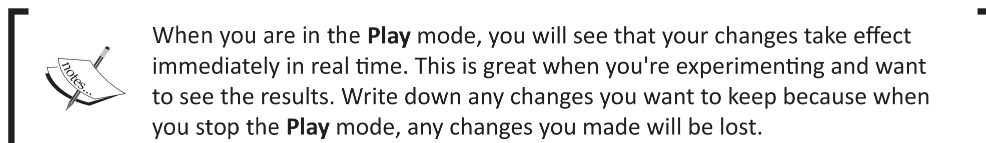
When we add our script to a GameObject, the name of our script shows up in the **Inspector** panel as a Component. Unity makes a couple of small changes. You might have noticed that when we added `LearningScript` to **Main Camera**, Unity actually showed it in the **Inspector** panel as **Learning Script**. Unity added a space to separate the words of the name. Unity does this modification to the variable names, too. Notice that the variable `number1` is shown as **Number 1**, and `number2` as **Number 2**. Unity capitalizes the first letter as well. These displayed changes improve readability in **Inspector**.

Changing a property's value in the Inspector panel

There are two situations when you can modify a property value:

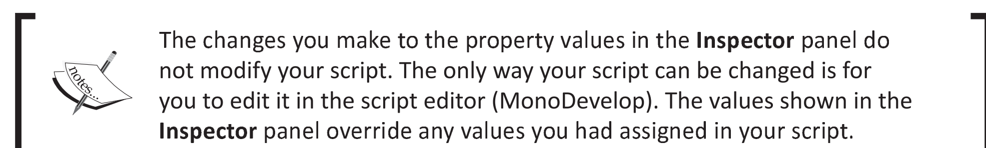
- During the **Play** mode
- During the development mode (not in the **Play** mode)

When you are in the **Play** mode, you will see that your changes take effect immediately in real time. This is great when you're experimenting and want to see the results.



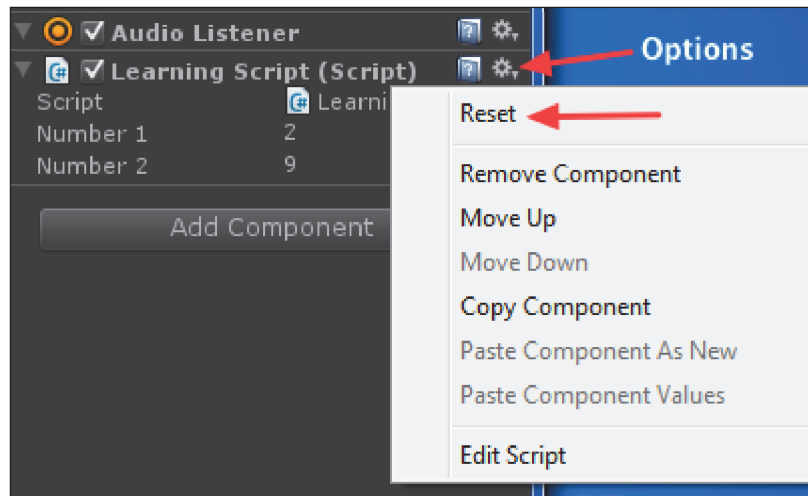
When you are in the **Play** mode, you will see that your changes take effect immediately in real time. This is great when you're experimenting and want to see the results. Write down any changes you want to keep because when you stop the **Play** mode, any changes you made will be lost.

When you are in the development mode, changes you make to the property values will be saved by Unity. This means that if you quit Unity and restart it again, the changes will be retained. Of course you won't see the effect of your change until you click on **Play**.



The changes you make to the property values in the **Inspector** panel do not modify your script. The only way your script can be changed is for you to edit it in the script editor (MonoDevelop). The values shown in the **Inspector** panel override any values you had assigned in your script.

If you desire to undo the changes you've made in the **Inspector** panel, you can reset the values to the default values assigned in your script. Click on the Cog icon (the gear) on the far right of the Component script, and then select **Reset** as shown in the following screenshot:



Displaying public variables in the Inspector panel

I'm sure you're wondering what the word `public` means at the beginning of a variable statement:

```
public int number1 = 2;
```

It means that the variable will be visible and accessible. It will be visible as a property in the **Inspector** panel so that you can manipulate the value stored in the variable. It also means it can be accessed from other scripts using the Dot Syntax. You'll learn more about the Dot Syntax in *Chapter 6, Using Dot Syntax for Object Communication*.

Time for action – making a variable private

Not all variables need to be `public`. If there's no need for a variable to be changed in the **Inspector** panel nor be accessed from other scripts, it doesn't make sense to clutter the **Inspector** panel with needless properties. In `LearningScript`, perform the following steps:

1. Change line 6 to the following:

```
private int number1 = 2;
```

2. Change line 7 to the following:

```
int number2 = 9;
```

3. Save the file.
4. In Unity, select **Main Camera**.

What just happened?

You will notice in the **Inspector** panel that both properties, **Number 1** and **Number 2** are gone.

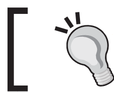
- Line 6: `private int number1 = 2;`

  The preceding line explicitly states that the `number1` variable is to be `private`, therefore the variable is no longer a property in the **Inspector** panel. It is now a private variable to store data.

- Line 7: `int number2 = 9;`

  The `number2` variable is no longer visible as a property either, but you didn't specify it as `private`.

If you don't explicitly state whether a variable will be `public` or `private`, by default, the variable will implicitly be `private`.



It is good coding practice to explicitly state whether a variable will be `public` or `private`.

So now when you click on **Play**, the script works exactly as it did before. You just can't manipulate the values manually in the **Inspector** panel anymore.

Naming your variables properly

Always use meaningful names for storing your variables. If you don't do that, six months down the line, you will be sad. I'm going to exaggerate here a little bit to make a point. I will name a variable as shown in the following code:

```
public bool theBearMakesBigPottyInTheWoods = true;
```

That's a descriptive name. In other words, you know what it means by just reading the variable, and so ten years from now when you look at that name, you'll know exactly what it means. Now suppose instead of `theBearMakesBigPottyInTheWoods`, I had named this variable as shown in the following code:

```
public bool potty = true;
```

Sure, you know what `potty` is, but would you know that it referred to a bear making a big potty in the woods? I know right now you'll understand it because you just wrote it, but six months down the line, after writing hundreds of other scripts for all sorts of different projects, you'll look at that and wonder what `potty` meant. You'll have to read several lines of code you wrote to try to figure it out.

You may look at the code and wonder who in their right mind would write such a terrible code. So take the time to write a descriptive code that even a stranger can look at and know what you mean. Believe me, in six months or probably less, you will be that stranger.

Begin variable names with lowercase

You should begin a variable name with lowercase because it helps to distinguish between a class name and a variable name in your code. The Component names (class names) begin with a capital letter. For example, it's easy to know that `Transform` is a class, and `transform` is a variable.

There are of course exceptions to this general rule, and every programmer has a preferred way to use lowercase, uppercase, and perhaps an underscore to begin a variable name. At the end, you will have to decide upon a naming convention you like. If you read the Unity forums, there are some heated debates on naming variables. In this book, I will show you my preferred way, but you can use whatever is more comfortable for you.

Using multi-word variable names

Let's use the same example again as follows:

```
public bool theBearMakesBigPottyInTheWoods = true;
```

You can see that the variable name is actually eight words squished together. Since variable names can be only one word, begin the first word with a lowercase, and then just capitalize the first letter of each additional word. It greatly helps to create descriptive names and still being able to read it. There's a word for this called **camelCasing**.

Have a go hero – viewing multi-word variables in the Inspector panel

I already mentioned that for `public` variables, Unity's **Inspector** will separate each word and capitalize the first word. Go ahead, add the previous statement to `LearningScript` and see what Unity does with it in the **Inspector** panel.

Declaring a variable and its type

Every variable we want to use in a script must be declared in a statement. What does that mean? Well, before Unity can use a variable; we have to tell Unity about it first. Ok then, what are we supposed to tell Unity about the variable?

There are only three absolute requirements for declaring a variable and they are as follows:

- We have to specify the type of data a variable can store
- We have to provide a name for the variable
- We have to end the declaration statement with a semi-colon

The following is the syntax we use for declaring a variable:

```
typeOfData nameOfTheVariable;
```

Let's use one of the `LearningScript` variables as an example; the following is how to declare a variable with the bare minimum requirements:

```
int number1;
```

The following is what we have:

- **Requirement #1** is the type of data that `number1` can store, which in this case is an `int`, meaning an integer
- **Requirement #2** is a name which is `number1`
- **Requirement #3** is the semi-colon at the end

The second requirement of naming a variable has already been discussed. The third requirement of ending a statement with a semi-colon has been discussed. The first requirement of specifying the type of data will be covered next.

The following is what we know about this bare minimum declaration as far as Unity is concerned:

- There's no `public` modifier which means it's `private` by default
- It won't appear in the **Inspector** panel, or be accessible from other scripts
- The value stored in `number1` defaults to zero

The most common built-in variable types

This section only shows the most common built-in types of data that C# provides for us and that variables can store.

Just these basic types are presented here so that you understand the concept of a variable being able to store only the type of the data you specify. The custom types of data that you will create later will be discussed in *Chapter 7*, *Creating the Gameplay is Just a Part of the Game* in the discussion of Dot Syntax.

The following chart shows the most common built-in types of data you will use in Unity:

Type	Contents of the variable
int	A simple integer, such as the number 3
float	A number with a decimal, such as the number 3.14
string	Characters in double quotes, such as, “Watch me go now”
bool	A boolean, either **true** or **false**

There are few more built-in types of data that aren't shown in the preceding chart. However, once you understand the most common types, you'll have no problem looking up the other built-in types if you ever need to use them.

We know the minimum requirements to declare a variable. However, we can add more information to a declaration to save our time and coding. We've already seen some examples in `LearningScript` of assigning values when the variable is being declared and now we'll see few more examples.

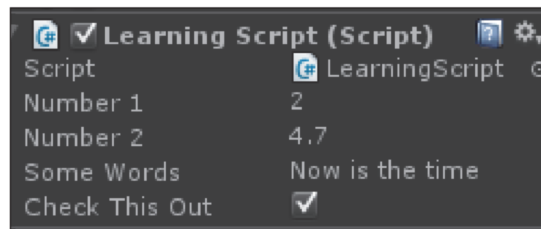
Time for action – assigning values while declaring the variable

Add some more variables to `LearningScript` using the types shown in the previous chart. While declaring the variables, assign them values as shown in the following screenshot. See how they are presented in the **Inspector** panel. These are all `public` variables so they will appear in the **Inspector** panel:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class LearningScript : MonoBehaviour
5  {
6      public int number1 = 2;
7      public float number2 = 4.7f;
8      public string someWords = "Now is the time";
9      public bool checkThisOut = true;
10
11     void Start ()
12     {
13
14     }
15
16     void Update ()
17     {
18
19     }
```

What just happened?

The following screenshot shows what Unity presents in the **Inspector** panel:

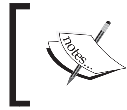


The variables are displayed in the **Inspector** panel with the values already set.

Where you declare a variable is important

You will be declaring and using variables in many places in a script. The variables that I have shown you so far are called **member variables**. They are members of the `LearningScript` class, not declared within any method. These member variables are the only variables that have the option of being displayed in the **Inspector** panel or being accessed by other scripts.

So where in the class should the member variables be declared? This is another subject that can lead to heated discussions. Personally, I declare them at the top of a class file before any methods are declared so that I see them all in one place. Other people like to declare variables close to the point of first use in a method.



Declaring your member variables at the beginning of a class may give you a mental clue that these member variables can be used everywhere in the script.

We will also be creating variables in methods. These variables are called as **local variables** and are never displayed in the Unity's **Inspector** panel, nor can they be accessed by other scripts. This brings us to another programming concept called **variable scope**.

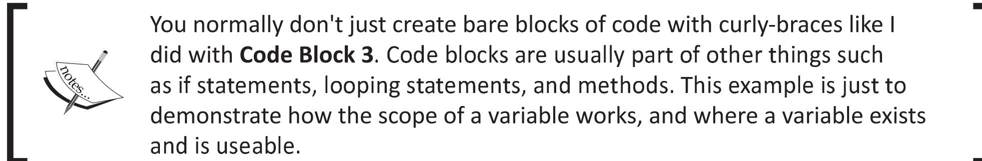
Variable scope – determining where a variable can be used

Variable scope is a fancy way of saying "Where in the script does a variable exist". The following screenshot explains you the scope of variables:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class LearningScript : MonoBehaviour
5  {                                                   Code Block 1
6      string block1 = "Block 1 text";
7
8      void Start ()
9      {                                               Code Block 2
10         Debug.Log(block1);
11         string block2 = "Block 2 text";
12         Debug.Log(block2);
13         {                                           Code Block 3
14             Debug.Log(block1);
15             Debug.Log(block2);
16             string block3 = "Block 3 text";
17             Debug.Log(block3);
18         }
19     }
20 }
```

You might have noticed that the rectangular blocks start and end with curly-braces. Just like the `AddTwoNumbers()` method in *Chapter 2*, *Introducing the Building blocks for Unity Scripts*, the code between the opening curly-brace and a closing curly-brace is called a **code block**. Absolutely anywhere in a code that you have an opening curly-brace, there will be a closing curly-brace to match. All the code between the two braces is a code block.

Notice that the code blocks can be nested inside other code blocks.



You normally don't just create bare blocks of code with curly-braces like I did with **Code Block 3**. Code blocks are usually part of other things such as if statements, looping statements, and methods. This example is just to demonstrate how the scope of a variable works, and where a variable exists and is useable.

The following is what you have:

Line 16: `string block3 = "Block 3 text";`

The preceding line declares a local `string` variable named `block3`. This variable exists in the code block that is labeled **Code Block 3**. If you try to use the variable `block3` outside of **Code Block 3**, such as in **Code Block 2** or **Code Block 1**, Unity will give you an error message saying that variable `block3` doesn't exist.

The scope of the variable `block3` is the code block defined by the curly-braces of lines 13 and 18.

Now let's look at the `block1` variable:

Line 6: `string block1 = "Block 1 text";`

The preceding line declares a `string` type member variable named `block1`. This variable exists in the code block that is labeled **Code Block 1**. This code block begins on line 5 and ends on line 20. This means the variable `block1` can be used everywhere, including **Code Block 2 and Code Block 3** because, they are also within **Code Block 1**. The `block1` variable is used in **Code Block 2** on line 10, and in **Code Block 3** on line 14.

The scope of the `block1` variable is the code block defined by the curly-braces between lines 5 and 20.

Pop quiz – knowing how to declare a variable

Q1. What is the proper way to name a variable?

Q2. How do you make a variable appear in the Unity's **Inspector** panel?

Q3. Can all variables in a script show in the **Inspector** panel?

Q4. Can a variable store any type of data?

Summary

We first covered how to write a C# statement, especially the semi-colon to terminate a statement. All the Component properties shown in the **Inspector** panel are member variables in the Component's class. Member variables can be shown in the **Inspector** panel, or accessed by other scripts when the variable is declared as `public`. The type of data a variable can store is specified when it's declared. Finally, we learned that variable scope determines where it is allowed to be used.

Now that we've learned about variables, we're ready to learn the details of C# methods that will use the variables we create – which is the topic of the next chapter.

4

Getting into the Details of Methods

In the previous chapter, you were introduced to a variable's scope—where 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 second chapter you saw that a method is a code block that can execute by just calling the method's name. It's time to see the importance of code blocks and the variables used in them. A method defines a code block which begins and ends with curly braces.

In this chapter we will cover the features of methods:

- Ending method definitions with curly braces
- Using methods in a script
- Naming methods properly
- Defining a method
- Calling a method
- Returning a value from a method
- Using Unity's `Update()` and `Start()` methods

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

Ending a method definition using curly braces

At the beginning of *Chapter 3*, *Getting into the Details of Variables* you learned about C# statements and the requirement to end them with a semicolon. A method definition has a different requirement.

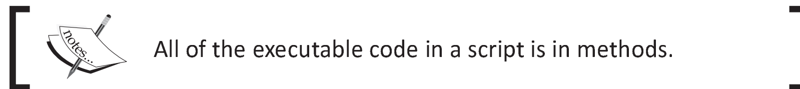
A method definition ends with a code block between a pair of curly braces. *DO NOT* end a method definition with a semicolon.

If you do accidentally place a semicolon at the end, MonoDevelop will gladly remind you with an error message that you're not supposed to use a semicolon at the end of a method definition.

Using methods in a script

There are two reasons for using methods in a script:

- To provide behavior to a GameObject
- To create reusable sections of code



All of the executable code in a script is in methods.

The first purpose of a method is to work with the member variables of the class. The member variables store data that's 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 we make the 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 into a code block and give it a name so you can call it when needed.

Naming methods properly

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

Since methods make GameObject do something useful, you should give your method a name that sounds like "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 in six months you'll look at that and say "Wiggle? Wiggle what?" It only takes a moment to be a little more precise and write `WiggleMyButt`. Now when you see that method name, you'll know exactly what it's going to do.

Begin method names with an uppercase letter

Why? We do this to make it easier to tell the difference between what is a class or method, and what is a variable. Also, Microsoft suggests 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 multi-word names for a method

Using this example again:

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

You can see the name is actually three words squished together. Since method names can only one word, the first word begins uppercase, then just capitalize the first letter of each additional word. For example, PascalCasing.

Parentheses are part of the method name

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

Defining a method properly

Just like for 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. Which is correct? In C#, it doesn't make any difference. Use which ever term helps you learn easier. 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, a method will return to the place where the method was called
- The name of the method should be followed by a pair of parentheses
- A pair of curly braces should be present for containing the code block:

```
returnDataType   NameOfTheMethod ( )
{
}
```

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

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

Here's what we have:

- Our first requirement is the type of data 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 keyword `void` is used. This informs Unity that nothing is being returned from the method.
- Second requirement is the method name which is `Start()`.
- Last requirement is the curly braces, which contains the code that defines what the method is going to do.

This example fulfills the bare minimum requirements to be 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, but it's still 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 is done being 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 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.

Understanding parentheses – why are they there?

One thing for sure is that it makes 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 get called multiple times. That's one of the reasons why a method created in the first place, so we don't have to write the same code over and over. Remember the AddTwoNumbers () method back in Chapter 2. It was very simple method used to explain the concept of a method and how to call it. Now it's time to take the next step and learn the usefulness of the parentheses.

Time for action – adding code between the parentheses

We're going to modify `LearningScript` to send some information to the `AddTwoNumbers()` method to make it much more useful.

Why would we need to send information to a method?

A script may need to add two numbers several times, but they probably won't always be the same two numbers. We could possibly have 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.

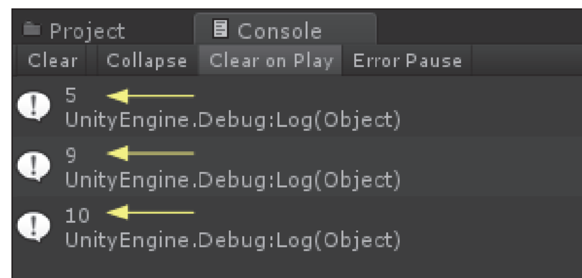
```
1  using UnityEngine;
2  using System.Collections;
3
4  public class LearningScript : MonoBehaviour
5  {
6      int number1 = 2;
7      int number2 = 3;
8      int number3 = 7;
9
10     void Start ()
11     {
12         AddTwoNumbers(number1, number2);
13         AddTwoNumbers(number1, number3);
14         AddTwoNumbers(number2, number3);
15     }
16
17     void Update ()
18     {
19
20     }
21
22     void AddTwoNumbers (int firstNumber, int secondNumber)
23     {
24         int result = firstNumber + secondNumber;
25         Debug.Log(result);
26     }
27 }
28
```

Using the preceding screenshot, perform the following steps:

1. Open `LearningScript` in MonoDevelop to modify it.
2. Add lines 6, 7, and 8 to declare three integer variables.
3. Add lines 22 to 26 to define the `AddTwoNumbers()` method with parameters.
4. Add lines 12, 13, and 14 to call the `AddTwoNumbers()` three times.
5. Save the file.
6. Click on **Play** in Unity.

What just happened?

As this script executes, the `AddTwoNumbers()` method is called three times on lines 12, 13, and 14. The method's code block adds two numbers and displays the result in the Unity **Console** (see the yellow arrows in the following screenshot):



Those parentheses are like a *cubbyhole*. When we call `AddTwoNumbers()` a couple of numbers are stuffed into the cubbyhole. When the code block executes, it takes those two numbers held in the cubbyhole and uses them on line 24.

There's a special name for that information between the parentheses of a method definition, such as line 22—the code is called the 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 the method is allowed to use. That's it, it's very simple.

In the earlier screenshot, on line 22 the red arrows pointed to the type of the declared variables `firstNumber` and `secondNumber`. The type is `int`, or `integer`. Now notice the red arrow pointing to the variables `number1`, `number2`, and `number3`. They are also of the type `int`. These variables have to be of type `int` since they store the numbers that will be added in the method, which the parameters specify will be of type `int`.

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, an integer, that the method will use. You set some limits on what's allowed.

Ok, 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's storing a value that will be used in the code block on line 24. What do we call things that store data? That's right, variables. Variables are used everywhere.

Remember, a variable is just a substitute name for the value it actually stores.

As you can see on line 22 of the code block, those variables are being added together.

How many parameters can a method have?

We can have as many as you need to make the method work properly. Whether we write our own custom methods, or you 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.

Calling a method

In the earlier screenshot, look at lines 12 and 22. Do you notice anything different? They sure don't look the same, do they? The variable names, which the blue arrows point to, are different.

If you are looking at that code and saying "What the heck?" then don't feel bad. When I was first learning the concept of calling methods, I had one heck of time understanding how the code worked. It is, in fact, very simple, but I fought with this for days before the lights came on. I consulted all the programming books I had, written by all the experts, and not a single one had the decency to explain how the code worked. All those book authors just assumed I'd "get it" because after all, they were experts.

I had to figure it out myself with trial and error testing. After many days, I finally had my "Ah-Ha" moment.

Using arguments in the parentheses of a method

Arguments?? Who dreams up these words? We all know what an argument is. Every one of us has been involved in an argument at some time. Well, someone decided this would be a good word to mean something in programming. Sure enough, look it up in a dictionary and you'll probably see something like this: "A value or address passed to a procedure or function at the time of call."

Yup, that explains it totally, right? Ok, let's really learn what arguments are, and what they do in code. In the previous screenshot, look at line 12:

```
AddTwoNumbers(number1, number2);
```

Between the parentheses are the variables, `number1` and `number2`. Those two variables are called the arguments that are being passed to the method. In simple terms, the values stored in these two variables, 2 and 3, are placed in the cubbyhole.

On line 22, the method defines that it takes in two parameters called `firstNumber` and `secondNumber`. This means, of course, that somewhere in this process these parameters will have to have values assigned them.

Here's the secret I finally discovered on my own. Behind the scenes, where you can't see, the values 2 and 3, that are in the cubbyhole, are now assigned to the variables `firstNumber` and `secondNumber`.

You don't see this code, but if you could see it, what happens with arguments and parameters looks just like this:

```
firstNumber = number1;
secondNumber = number2;
```

- Since the argument `number1` contained the value 2, now the parameter `firstNumber` contains the value 2
- Since the argument `number2` contained the value 3, now the parameter `secondNumber` contains the value 3

Now the code block is executed and the value 5 is displayed in the Unity **Console**.

As you can now see, the names of the arguments and the names of the parameters don't need to be the same. They're just names of variables used in different places in your code. They're just substitutes for the actual values each contain, and it's the value that's getting transferred from the method call to the method code block.

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 have the method return a value back. Previously, all you had the `AddTwoNumbers()` method do was take the result of adding two numbers and display it to Unity's Console.

Now, instead of displaying the result directly, you're going to modify `AddTwoNumbers()` to return the result of the addition back to the place the method was called.

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 the code in the method's code block and placing it right there where the method was called.

The next screenshot is still very simple, but it shows how this substitution works and how returning a value from a method works.

Time for action – returning a value from AddTwoNumbers()

Modify `LearningScript` to call the `AddTwoNumbers()` method twice and get a grand total. Also create another method whose sole purpose is to display the grand total result.

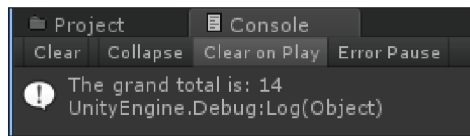
1. Open `LearningScript` in MonoDevelop to modify it.
2. On line 12, declare the answer variable (this statement is on 3 lines).
3. On lines 19 to 23, redefine the `AddTwoNumbers()` method with a return type.
4. On lines 25 to 28, define the `DisplayResult()` method.
5. Save the file.
6. Click on Play in Unity.

```
1   using UnityEngine;
2   using System.Collections;
3
4   public class LearningScript : MonoBehaviour
5   {
6       int number1 = 2;
7       int number2 = 3;
8       int number3 = 7;
9
10      void Start ()
11      {                       Notice this single statement is on 3 lines.
12          int answer =
13          AddTwoNumbers(number1, number2) +
14          AddTwoNumbers(number1, number3);
15
16          DisplayResult(answer);
17      }
18
19      int AddTwoNumbers (int firstNumber, int secondNumber)
20      {
21          int result = firstNumber + secondNumber;
22          return result;
23      }
24
25      void DisplayResult(int total)
26      {
27          Debug.Log("The grand total is: " + total);
28      }
29
30  }
```

What just happened?

As you can see in the following screenshot, the result is **14**. However, the main concept to learn from this example is this:

- When a method returns a value, it's a type of data just like a variable would store
- In fact, the value returned from a method could easily be stored in a variable



Analysis of the code is as follows:

- The code on line 10 and its description is as follows:

  ```
  void Start()
  ```

  Unity calls the `Start()` method once only.

- The code on lines 12 to 14 and its description is as follows: (Note: I have put this single statement on three lines for a better screenshot.)

  ```
  int answer =
  AddTwoNumbers(number1, number2) +
  AddTwoNumbers(number1, number3);
  ```

  All this line does is add two numbers and store the result in a variable named "answer".

  First there is a call to `AddTwoNumbers(number1, number2)` on line 19.

  The arguments `number1` and `number2` send the integers 2 and 3 to the method parameters on line 19.

- The code on line 19 and its description is as follows:

  ```
  int AddTwoNumbers(int firstNumber, int secondNumber);
  ```

  The integers 2 and 3 are assigned to the parameter variables `firstNumber` and `secondNumber`.

- The code on line 21 and its description is as follows:

  ```
  int result = firstNumber + secondNumber;
  ```

  The numbers 2 and 3 are added and stored in the declared variable result.

- The code on line 22 and its description is as follows:

  ```
  return result;
  ```

  The integer 5, stored in the variable result, is returned back to line 12, where the method was called.

- Back to the code on line 12 with its description:

  Where you see `AddTwoNumbers(number1, number2)`, now sits the integer 5. The substitution has taken place.

  Now, line 12 continues its execution with another call to:
  `AddTwoNumbers(number1, number3)` on line 19

  The only difference is that the arguments have changed.

  The arguments `number1` and `number3` send the integers 2 and 7 to the method parameters on line 19.

- Back to the code on line 19 again with its explanation:

  The integers 2 and 7 are assigned to the parameter variables `firstNumber` and `secondNumber`.

- The code on line 21 and its description:

  `2` and `7` are added and stored in result.

- The code on line 22 with its description:

  The integer 9, stored in result, is returned back to line 12, where the method was called.

- Back to the code on line 12 again with its description:

  Where you see `AddTwoNumbers(number1, number3)`, now sits the integer `9`. The substitution has taken place.

  Now line 12 continues its execution. There is a plus sign between the two method calls which means `5` and `9` are added together and the resultant integer `14` is now stored in the variable answer.

  The `Start()` method code block now continues execution on line 16.

- The code on line 16 and its description is as follows:

  ```
  DisplayResult(answer);
  ```

  This is calling the `DisplayResult()` method on line 25.

  It takes one argument. The argument used is the variable answer which stores a value of type `int`.

  The argument answer sends the integer 14 to the method parameter on line 25.

- The code on line 25 with its descripton:

  ```
  void DisplayResult(int total)
  ```

  The integer 14 is assigned to the parameter variable total.

- The code on line 27 and its description:

  ```
  Debug.Log("The grand total is: " + total);
  ```

  This output to the Unity **Console** includes a little peek into the next chapter.

  Some text is displayed as well as the value stored in the variable total.

  The Unity **Console** displays **The grand total is: 14**.

  The `Start()` method is done executing its code. Since there is no further code in `LearningScript` to execute, the script is done.

Have a go hero – add two more numbers together

Try modifying line 12 to add the numbers together that are stored in the variables `number2` and `number3`. You will have to include an additional call to `AddTwoNumbers()`. The result in the **Console** should be **The grand total is: 24**.

Calling a method is a logic detour

As you can see by following the code analysis, code is executed one step at a time. However, calling a method does send code execution on a detour. The method is then executed one line at a time until the end of the method is reached. If the method return type is void, then execution restarts from the point where the method was called. If the method returns a value, then the value returned is substituted at the place the method was called, then execution restarts from the point of substitution.

Using Unity's Update and Start methods

Every time you create a script in Unity, these two skeleton methods are included. That's because they are rather important. These are the most commonly used `MonoBehaviour` methods, see the next screenshot for others. I like to call these Unity's magic methods because you don't call these methods, Unity does. It's usually important that at least one `MonoBehaviour` method is included in a Unity script to cause the script to execute. I say usually because other methods in the script may be called from another script or class.

How do I know these two methods are called by Unity and that they are `MonoBehaviour` methods? Here, the Unity **Scripting Reference** is your friend.

Here's just a portion of the methods Unity can call in a script. This is from the **Scripting Reference**. Just search for `MonoBehavior`:

Overridable Functions
Update
LateUpdate
FixedUpdate
Awake
Start

Look at line 4 of **LearningScript**:

```
public class LearningScript : MonoBehaviour
```

This line says that `LearningScript` inherits from `MonoBehaviour`. Any script that inherits from `MonoBehaviour` will automatically call the methods `Update()` and `Start()` if they are in the script. Therefore, if you want, you can create a script in MonoDevelop instead of Unity, just have it inherit from `MonoBehavior` by adding: `: MonoBehaviour` after the class name.

Please notice the **colon** that needs to be included.

The Start method is called one time

Unity calls this method only one time. When the GameObject your script is attached to is first used in your scene, the `Start()` method is called. This method is primarily used to initialize, or setup, the member variables in your script. This allows everything in your script to be ready to go before `Update()` is called for the first time.

You've probably noticed that many of the examples I used in `LearningScript` are making use of `Start()`. These examples weren't initializing any code, I was just taking advantage of the fact that since `Start()` is only called once, displaying output to the Console would, therefore, only be displayed once, which just made it easier to see the output displayed.

The Update method is called over and over and over...

As you study the sample code in the **Scripting Reference**, you will notice that a vast majority of the code is in the `Update()` method. As your game runs, the Scene is displayed many times per second. This is called **Frames per Second**, or **FPS**. After each frame is displayed, the `Update()` method is called by Unity to run your code.

Since `Update()` is called every frame, it allows your game to detect input, such as mouse clicks and key presses, every frame. User input is one of the topics we are about to cover in the next chapter.

Pop quiz – understanding method operation

Q1. What are the minimum requirements for defining a method?

Q2. What is the purpose of the parentheses at the end of the method's name?

Q3. What does void mean in a method definition?

Q4. In a Unity script, how is the `Update()` called?

Summary

In this chapter, we learned that a method definition ends with a code block between two curly braces, not with a semicolon. The parentheses are part of a method's name. We also learned how to call a method into action, how to use data returned from a method, and that Unity calls some methods automatically, such as the `Start()` and `Update()` methods, when the script inherits from the `MonoBehaviour` class.

You now know the two major building blocks of scripting, variables and methods. From now on, everything else you do will just be making use of variables and methods. Now that you understand these two building blocks, you are ready for the next chapter that deals with making decisions in your code.

5
Making Decisions in Code

One of the primary duties of a computer is controlling what happens next when certain conditions are met. That's what computers do whether the code is controlling an application or a game. We write scripts to make GameObjects behave a certain way one moment, then the behavior should change when the conditions change. A script has to detect when the conditions change, then make the appropriate code execute based on the new conditions. This chapter looks at some examples of the ways that conditions can change, and the code to detect these changes. This in turn determines which code in the script is executed next.

In this chapter we will discuss:

- If statement decisions
- Checking for many conditions
- If-else statement decisions
- User's input condition changes
- Looping though data in an Array, List, or Dictionary

Let's begin...

Testing conditions with an if statement

If, if, if.   If I do this… if I do that… What happens if…

Certainly you've had to make decisions about all kinds of things in your life every day. We all do it all the time without actually giving the process of making a decision much thought, if any. As we make daily decisions, most of the time we just do the decision processing in our head. Unity doesn't have that human luxury, so we have to write it out so Unity can know the conditions that lead to certain choices. Having to write the logic is the strange part of writing code for beginners, simply because people usually make the vast majority of decision without writing anything down first. However, it is very simple to do.

An `if` statement is the most common way GameObjects make decisions. Data used to make these decisions is the information usually stored in some variables. For an if statement it's as easy as saying "If my condition is met, then execute my code block."

Testing if conditions are true or false

A sampling of conditions that can be true or false:

- The user pressed a button
- The temperature is cold
- The character died
- The bear made big potty in the woods

General questions like these are answered by humans, usually, with either a yes or no. For Unity, the answers will be either true or false. For example: "the bear made big potty in the woods" is either true, or false.

Time for action – create a couple of if statements

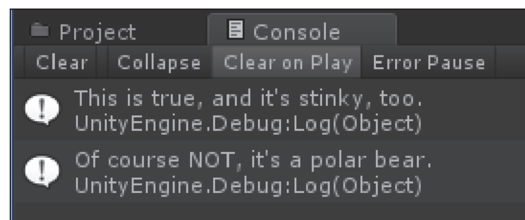
The **if** statements work by determining whether a condition inside a pair of parentheses is true or false.

1. Modify `LearningScript` as shown in the next screenshot.
2. Save the file.
3. In Unity, click on Play.

```
 1  using UnityEngine;
 2  using System.Collections;
 3
 4  public class LearningScript : MonoBehaviour
 5  {
 6      void Start ()
 7      {
 8          bool theBearMadeBigPottyInTheWoods = true;
 9
10          if(theBearMadeBigPottyInTheWoods)
11          {
12              Debug.Log("This is true, and it's stinky, too.");
13          }
14
15          theBearMadeBigPottyInTheWoods = false;
16
17          if(!theBearMadeBigPottyInTheWoods)
18          {
19              Debug.Log("Of course NOT, it's a polar bear.");
20          }
21      }
22  }
```

What just happened?

Here's the output in the Unity **Console**:



Code analysis:

- The code on line 8 is as follows:

  `bool theBearMadeBigPottyInTheWoods = true;`

  This Boolean variable is declared and assigned the value of `true`.

- The code on line 10 and its description:

  `if( theBearMadeBigPottyInTheWoods)`

  An `if` statement to test if the condition between the parenthesis is `true` or `false`.

  The variable `theBearMadeBigPottyInTheWoods` is storing a value `true`, therefore.The code block on lines 11 to 13 is executed, as shown in the **Console** screenshot.

Using the NOT operator to change the condition

Here's a little curveball to wrap your mind around, the NOT logical operator. It's written in code using an exclamation mark. This makes a true condition false, or a false condition true.

- The code on line 15 along with its description:

  ```
  theBearMadeBigPottyInTheWoods = false;
  ```

  Assigns the value `false` to `theBearMadeBigPottyInTheWoods`.

- The code on line 17 with its description is as follows:

  ```
  if( ! theBearMadeBigPottyInTheWoods)
  ```

  Another if statement, but this time `theBearMadeBigPottyInTheWoods` is false.

  However, there's a NOT logical operator in front of the variable. See the exclamation mark in the red circle shown in the previous screenshot.

  This means the if statement condition is NOT false, which is the same as saying true. Therefore the code block on lines 18 to 20 will be executed, as shown in the Console screenshot

  The code block on lines 18 to 20 will be executed, as shown in the Console screenshot

I can already hear your question, why not just check for true? As you will discover when writing if statements, you need to be able to make decisions based on whether a condition is true, or if the condition is false. You want the option to execute a code block for either of these two conditions. For example, you may want to execute some code based on whether a user didn't press a button at a particular time. If the user did not press the button, then execute the code block.

Checking many conditions in an if statement

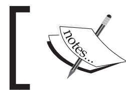
Sometimes you will want your if statements to check many conditions before any code block is executed. This is very easy to do. There are two more logical operators that you can use:

- AND: It is used by putting `&&` between the conditions being checked.
- OR: It is used by putting `||` between the conditions being checked.

Time for action – create if statements with more than one condition to check

1. Modify `LearningScript` as shown in the next screenshot.
2. Save the file.
3. In Unity, click on Play.

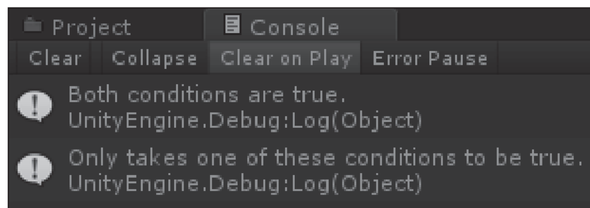
```
using UnityEngine;
using System.Collections;

public class LearningScript : MonoBehaviour
{
    void Start ()
    {
        bool theBearMadeBigPottyInTheWoods = true;
        int temperature = 40;

        if(temperature >= 35 && theBearMadeBigPottyInTheWoods)
        {
            Debug.Log("Both conditions are true.");
        }

        if(temperature >= 35 || theBearMadeBigPottyInTheWoods)
        {
            Debug.Log("Only takes one of these conditions to be true.");
        }
    }
}
```



Notice line 11 is using the AND operator, and line 16 is using the OR operator.

What just happened?

Here is the output you get in the Unity **Console**:



Code analysis:

- The code on line 8 and its description:

  `bool theBearMadeBigPottyInTheWoods = true;`

  A `bool` variable is declared and assigned the value of `true`.

- The code on line 9 with its description:

  `int temperature = 40;`

  An `int` variable is declared and assigned the value `40`.

- The code on line 11 with its description:

  ```
  if(temperature >= 35 &&  theBearMadeBigPottyInTheWoods)
  ```

  An `if` statement to test if both conditions are `true`.

  The first test is checking if the `temperature` is greater then, or equal to, `35`.

  The value stored in `temperature` is `40`, so this condition is true.

  The value stored in `theBearMadeBigPottyInTheWoods` is true. Therefore the first condition and the second condition are true, so the code block executes.

- The code on line 16 with its description:

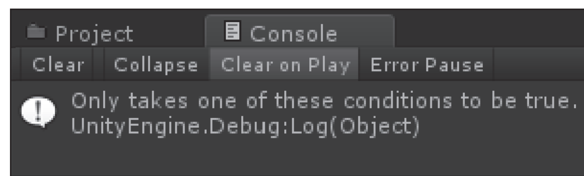
  ```
  if(temperature >= 35 || theBearMadeBigPottyInTheWoods)
  ```

  An `if` statement to test if either of the conditions are true.

  We already know that both the conditions are true, and either the first condition or the second condition needs to be true. Therefore the code block will execute.

Have a go hero – change the value assigned to temperature

Try changing `temperature` to a lower value such as `30`. Only one of the `if` statements will be true:



The following is the analysis of code:

- The code on line 11 and its description is as follows:

  ```
  if(temperature >= 35 && theBearMadeBigPottyInTheWoods)
  ```

  Only one of the conditions is now true, as 30 is not greater then, or equal to, 35.

  Therefore the first condition is false. Since both conditions have to be true, the code block does not execute.

- The code on line 16 and its description:

  ```
  if(temperature >= 35 || theBearMadeBigPottyInTheWoods)
  ```

  Only one of the conditions is now true.

  30 is not greater then or equal to 35, therefore the first condition is false.

  The second condition is true.

  Since only one of the two conditions has to be true, doesn't make any difference which one, the code block executes.

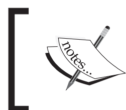
Have a go hero – change theBearMadeBigPottyInTheWoods to false

Now change `theBearMadeBigPottyInTheWoods` to false as well. Now you see that neither of the if statements will execute their code blocks.

Using an if-else statement to execute alternate code

So far, the `if` statements have needed certain conditions to be `true` for the code block to execute. There is an option that allows you to have an alternate code block execute when the `if` statement conditions are `false`.

- If my conditions are met, execute the following code block, else execute the alternate code block



This is very simple concept, just like a little kid saying: "If you give me an ice cream cone, I'll be nice, else I'm going to be naughty."

Time for action – add "else" to the if statement

if-else statements are just like regular **if statements** with the `else` option added.

1. Modify `LearningScript` as shown in the next screenshot.
2. Save the file.

3. In Unity, click on Play.

```
using UnityEngine;
using System.Collections;

public class LearningScript : MonoBehaviour
{
    void Start ()
    {
        bool theBearMadeBigPottyInTheWoods = false;

        if(theBearMadeBigPottyInTheWoods)
        {
            Debug.Log("This is executed because the condition is true");
        }
        else
        {
            Debug.Log("This is executed because the condition is false");
        }
    }
}
```

Line 14 shows how `else`, and its code block is simply added after the `if` code block.

What just happened?

The analysis of code is as follows:

- The code on line 8 and its description:

  `bool theBearMadeBigPottyInTheWoods = false;`

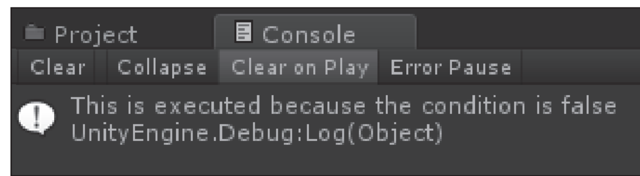
  The variable `theBearMadeBigPottyInTheWoods` is assigned the value of false.

- The code on line 10 and its description:

  `if( theBearMadeBigPottyInTheWoods)`

  Since the condition is false, the code block on lines 11 to 13 is not executed, and the script continues to line 14 of the if-else statement.

  Therefore, the code block on lines 15 to 17 is executed instead:



Pop quiz – understanding if statements

Q1. Humans can answer questions with a yes or no. What do C# if statements need as answers?

Q2. What logical operator can turn a `true` condition into `false`, or a `false` condition into `true`?

Q3. If two conditions have to be `true` to make an if statement's code execute, what logical operator would you use to join the conditions?

Q4. If only one of two conditions needed to be true to make an if statement's code execute, what logical operator would you use to join the two conditions?

Making decisions based on user input

Decisions always have to be made when the user provides input. In *Chapter 2*, *Introducing the Building Blocks for Unity Scripts,* we used an example where the user had to press the **Return/Enter** key to call the `AddTwoNumbers()` method:

```
if(Input.GetKeyUp(Keycode.Return))
AddTwoNumbers();
```

The `if` statement condition becomes true only when the `Return` key is released after being pressed down. Here's a partial screenshot of the `GetKeyUp()` method as shown in the **Scripting Reference**:

Input.GetKeyUp

static bool **GetKeyUp**(KeyCode **key**);

Description

Returns true during the frame the user releases the key identified by the key KeyCode enum parameter.

After the `Return` key is released, `AddTwoNumbers()` is executed.

Notice that the code, `AddTwoNumbers()`, isn't between two curly braces. When there is only one line of code to execute for an if or an else, you have the option to not use the curly braces.

Storing data in an array, a List, or a Dictionary

There are times that many items need to be stored in some type of list. Perhaps a selection of weapons that a character may use. An example used later in this book is a list of splashscreens for the State Machine project we will build.

There are basically two ways to access items in a list:

- **Direct retrieval**: The location of an item in the list is already known, so code is written to access it directly, or
- **Loop retrieval**: The location of an item in the list is not known, it's just in there somewhere, so code is written to loop through the list until the item desired is found.

First though, we need a list of items before we can select anything from the list. An example of collecting items into a list, then looping through the list, is shown in the **Scripting Reference** under the `GetComponents()` method:

```
public HingeJoint[] hingeJoints;
void Example() {
  hingeJoints = gameObject.GetComponents<HingeJoint>();
  ...
  }
```

All the `HingeJoints` in a GameObject are collected into an array (list). Once all the `HingeJoints` are in the array, it's up to us to decide which `HingeJoints` we want to work with in our code.

So bottom line, what are we talking about here? We know that a variable stores a single item. For instance, we could store a single weapon in a variable. That's great as long as we only have one weapon. Suppose we have the option of using many different weapons. To store these weapons we would need a separate variable for each one. A better option would be to store all the weapons in some sort of super-variable that can store many items, that way they're all stored in one place, not in a whole bunch of different variables.

That's what an array, a List, or a Dictionary is, a variable with the ability to store many items. Like a super-variable divided into many cubbyholes.

Storing items in an array

Looking at the `GetComponents()` example on the **Scripting Reference**, let's see how an array is created:

- As per the code on line 1: `public HingeJoint[] hingeJoints;`
    - `public` means this array will appear in the **Inspector**. Also the array is accessible from other scripts.
    - `HingeJoint[]` is the type of variable being created. It's going to be a `HingeJoint` type (`HingeJoint` is a class in the **Scripting Reference**).
    - The square brackets specify that the variable created is going to be an array, a variable with many cubby-holes to store several `HingeJoint` objects, and only `HingeJoint` objects.
    - `hingeJoint` is the name of the array being created.

  That was easy enough. It's just like creating any other variable. The only difference was the addition of the square brackets to specify that the type of variable being declared is actually going to be an array.

  Now that the array is created, the `GetComponent()` method retrieves all the `HingeJoints` on the GameObject and stores each of them into the array:

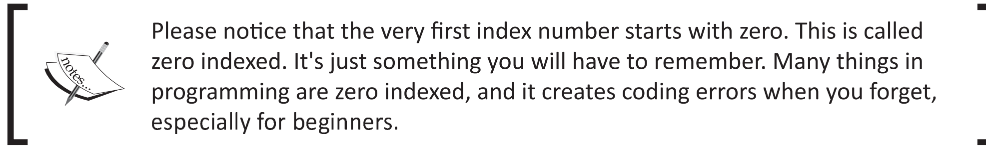
- As per the code on line 3: `hingeJoints = gameObject.GetComponents<HingeJoint>();`
    - `hingeJoints` is the array
    - GameObject is the variable that stores the GameObject this script is attached to
    - `GetComponents<HingeJoint>()` is the method used to find every `HingeJoint` object on this GameObject

As each `HingeJoint` object is found, it is stored into one of the cubbyholes of the array. These cubbyholes actually have a real name called an **element**. These elements actually have a specific location inside the `hingeJoint` array. Each element is given an index number. The first `HingeJoint` found would be stored in the element at index 0, the second one found is stored in the element at index 1. The third at index 2, and on and on until all the `HingeJoints` are found on the GameObject.

So if we knew exactly, which `HingeJoint` in the array we wanted to work with, perhaps the second `HingeJoint` which is stored in the element at index 1, we can simply retrieve it directly by saying it's stored in the variable:

```
hingeJoint[1]
```

Once again we use the square brackets because the variable is actually an array, and also to specify the index number.



Please notice that the very first index number starts with zero. This is called zero indexed. It's just something you will have to remember. Many things in programming are zero indexed, and it creates coding errors when you forget, especially for beginners.

That's all I want to say about using arrays to store objects because I want to discuss using a `List` instead. It's like an array with extra benefits.

Storing items in a List

Using a `List` instead of an array can be so much easier to work with in a script. Look at some forum sites related to C#, and Unity, and you'll discover that a great deal of programmers simply don't use an array unless they have to, they prefer to use a `List`.

Here are the basics of why a List is better, and easier, to use than an array:

- An array is a fixed size and unchangeable
- The size of a `List` is adjustable
- You can easily add to, and remove elements from a `List`
- To mimic adding a new element to an array, we would need to create a whole new array with the desired number of elements, then copy over the old elements

The first thing to understand is that a `List` has the ability to store any type of object, just like an array. Also, just like an array, we must specify, which type of object you want a particular `List` to store. This means that if you want a `List` of integers, of the `int` type, then you can create a `List` that will store only the `int` type. Want a `List` of pony names? Then create a `List` that will store only the `string` type.

Time for action – create a List of pony names

Create a `List` that stores the names of some ponies. Since they are names, use the `string` type.

1. Modify `LearningScript` as shown in the next screenshot.
2. Notice the change on line 2.
3. Save the file.

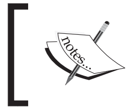
4. In Unity, click on Play.

```
1  using UnityEngine;
2  using System.Collections.Generic;
3
4  public class LearningScript : MonoBehaviour
5  {
6      void Start ()
7      {
8          List<string> myFavoritePonies = new List<string>();
9
10         myFavoritePonies.Add("Princess Cadence");
11         myFavoritePonies.Add("Fluttershy");
12         myFavoritePonies.Add("Nightmare Moon");
13
14         Debug.Log("This List has " + myFavoritePonies.Count + " ponies.");
15
16         Debug.Log("The pony's name at index 1 is " + myFavoritePonies[1]);
17         Debug.Log("The pony's name at index 2 is " + myFavoritePonies[2]);
18         Debug.Log("The pony's name at index 0 is " + myFavoritePonies[0]);
19     }
20 }
```

What just happened?

The following screenshot is the **Console** output. Notice the first output tells you there is a total of 3 elements in the `List`:

```
Project | Console
Clear | Collapse | Clear on Play | Error Pause
This List has 3 ponies.
UnityEngine.Debug:Log(Object)
The pony's name at index 1 is Fluttershy
UnityEngine.Debug:Log(Object)
The pony's name at index 2 is Nightmare Moon
UnityEngine.Debug:Log(Object)
The pony's name at index 0 is Princess Cadence
UnityEngine.Debug:Log(Object)
```



Please notice that your code is using dot syntax, which will be discussed in more detail in the next chapter. The main concepts I want you to focus on here are the features of a List.

The analysis of code is as follows:

- The code on line 2 is as follows:

  ```
  Using System.Collections.Generic;
  ```

  To be able to use a List, this tells Unity where to find the necessary C# code files for using a List.

  Change the using statement to using System.Collections.Generic;.

- The code on line 8 is as fololws:

  ```
  List<string> myFavoritePonies = new List<string>();
  ```

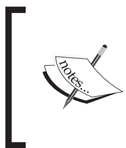
  This statement creates an empty List object.

  First thing to notice is that List<string> specifies that you are creating a List of type string.

  The name of the List is myFavoritePonies.

  Everything on the left side of the assignment operator (=) is creating a variable, declaring the type and the name.

  Everything on the right side is just like assigning a value to a variable, therefore new List<string() is a method called to create a new List object in computer memory, and give that memory location the name of myFavoritePonies.



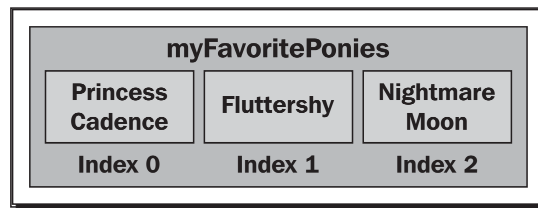
What is different here is that List is an object that itself can store data in elements. Imagine an egg carton as an object that can store the egg type. Creating objects will be discussed more in the next chapter about dot syntax.

- The code between lines 10 to 12:

  ```
  myFavoritePonies.Add("Princess Cadence");
  ```

  These three lines of code are adding `strings`, the pony names, to the `myFavoritePonies List`.

  Just like an array, each pony name `string` added is given an index number for the element that each pony name is stored in:



- The code on line 14 is as follows:

  ```
  Debug.Log("This List has " + myFavoritePonies.Count + " ponies");
  ```

  `myFavoritePonies.Count` retrieves the number of elements in the `List`.

- The code between lines 16 and 18:

  ```
  Debug.Log("The pony's name at index 1 is " + myFavoritePonies[1]);
  ```

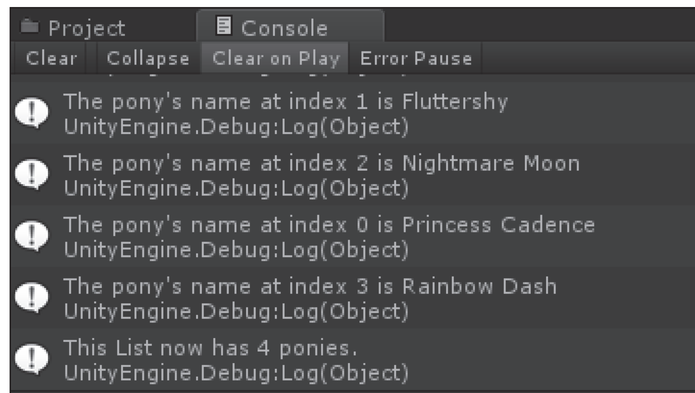
  Here you see the index number inside square brackets. Just like an array, this is how to directly retrieve the data in an element at a specific index.

  Like the array, the first element in a `List` is at index 0.

Have a go hero – add another pony to the List

Add another pony to the `List`, then display it's name. Also, in the **Console**, display the number of elements in the `List` after adding the fourth pony.

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

public class LearningScript : MonoBehaviour
{
    void Start ()
    {
        List<string> myFavoritePonies = new List<string>();

        myFavoritePonies.Add("Princess Cadence");
        myFavoritePonies.Add("Fluttershy");
        myFavoritePonies.Add("Nightmare Moon");

        Debug.Log("This List has " + myFavoritePonies.Count + " ponies.");

        Debug.Log("The pony's name at index 1 is " + myFavoritePonies[1]);
        Debug.Log("The pony's name at index 2 is " + myFavoritePonies[2]);
        Debug.Log("The pony's name at index 0 is " + myFavoritePonies[0]);

        myFavoritePonies.Add("Rainbow Dash");

        Debug.Log("The pony's name at index 3 is " + myFavoritePonies[3]);

        Debug.Log("This List now has " + myFavoritePonies.Count + " ponies.");
    }
}
```



Adding an element to the `List` shows the flexibility it has over an array. This is impossible to do using an array.

Storing items in a Dictionary

A dictionary has a Key/Value pair. The **Key** is just like an index in an array or list, it's associated with a particular value. The big benefit of a dictionary is that we can specify what the key is going to be. We have to specify the type and the name of the key that will be associated with the value stored.

A real world example you're familiar with is a collection of customers and their ID number. Just by knowing the customer's ID, you could retrieve the customer's information.

Time for action – create a dictionary of pony names and keys

Create a `Dictionary` using type `int` for the keys.

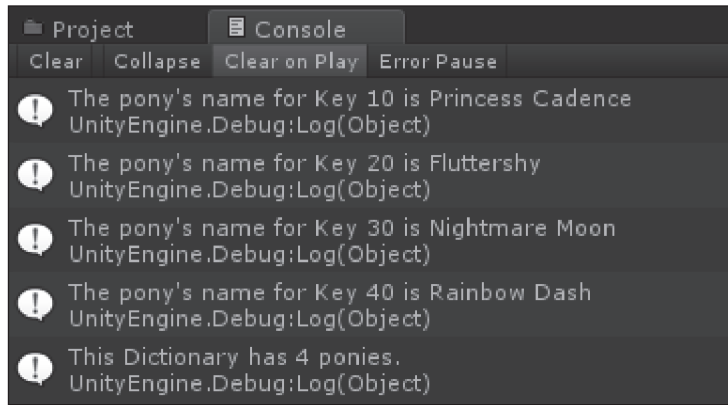
1. Modify `LearningScript` as shown in the next screenshot.
2. Save the file.
3. In Unity, click on Play.

```
1  using UnityEngine;
2  using System.Collections.Generic;
3
4  public class LearningScript : MonoBehaviour
5  {
6      void Start ()
7      {
8          Dictionary<int,string> myFavoritePonies = new Dictionary<int,string>();
9
10         myFavoritePonies.Add(10, "Princess Cadence");
11         myFavoritePonies.Add(20, "Fluttershy");
12         myFavoritePonies[30] = "Nightmare Moon";
13
14         Debug.Log("The pony's name for Key 10 is " + myFavoritePonies[10]);
15         Debug.Log("The pony's name for Key 20 is " + myFavoritePonies[20]);
16         Debug.Log("The pony's name for Key 30 is " + myFavoritePonies[30]);
17
18         myFavoritePonies[40] = "Rainbow Dash";
19
20         Debug.Log("The pony's name for Key 40 is " + myFavoritePonies[40]);
21
22         Debug.Log("This Dictionary has " + myFavoritePonies.Count + " ponies.");
23     }
24 }
```

2 ways to add to a Dictionary

What just happened?

Here is the output to Unity's **Console**.



The analysis of code is as follows:

- The code on line 8 is as follows:

  ```
  Dictionary<int, string> myFavoritePonies = new Dictionary<int,
    string>();
  ```

  Declaring a `Dictionary` is very similar to declaring a `List`.

  A `Dictionary` requires you to specify the type for the `Key`.

  This example used integers for the keys.

- The code on lines 10 and 11 is as follows:

  ```
  myFavoritePonies.Add(10, "Princess Cadence");
  myFavoritePonies.Add(20, "Fluttershy");
  ```

  Here you added two ponies using `Add`, just like you did for a `List`.

- The code on lines 12 and 18 with its description is as follows:

  ```
  myFavoritePonies[30] = "Nightmare Moon";
  myFavoritePonies[40] = "Rainbow Dash";
  ```

  Here you added ponies by assigning the pony name to a particular dictionary key.

Using a Collection Initializer to add items to a List or Dictionary

There is another way to add elements to a List or Dictionary. So far you have declared and created a new empty List and Dictionary, then added ponies to them on separate lines of code. You can add the ponies at the same time you declare the List or Dictionary with a Collection Initializer.

Time for action – adding ponies using a Collection Initializer

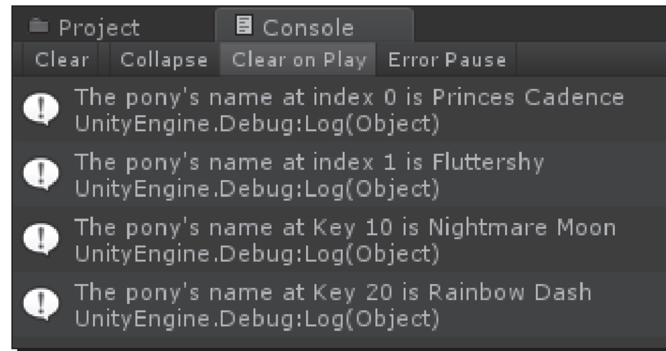
If we know the items to add ahead of time, we can add them when we create the `List` or `Dictionary`.

1. Modify `LearningScript` as shown in the next screenshot.
2. Save the file.
3. In Unity, click on Play.

```
1  using UnityEngine;
2  using System.Collections.Generic;
3
4  public class LearningScript : MonoBehaviour
5  {
6      void Start ()
7      {
8          List<string> myFavoritePonies = new List<string>()
9              {"Princes Cadence", "Fluttershy"};
10
11         Debug.Log("The pony's name at index 0 is " + myFavoritePonies[0]);
12         Debug.Log("The pony's name at index 1 is " + myFavoritePonies[1]);
13
14         Dictionary<int, string> ponyDictionary = new Dictionary<int, string>()
15         {{10,"Nightmare Moon"},{20,"Rainbow Dash"}};
16
17         Debug.Log("The pony's name at Key 10 is " + ponyDictionary[10]);
18         Debug.Log("The pony's name at Key 20 is " + ponyDictionary[20]);
19     }
20 }
```

What just happened?

Here's the **Console** output:



The analysis of the code is as follows:

- The code on lines 8 and 9 with its description:

  ```
  List<string> myFavoritePonies = new List<string>() {"Princess
    Cadence", Fluttershy"};
  ```

  This is actually a single statement. It's on two lines to make it fit the screenshot.

  Line 9 shows the **Collection Initializer** that's been added to the usual `List` declaration.

  Notice the pony names are between two curly braces. This is not a code block. This is another use of curly braces.

  This `List` Collection Initializer is the two curly braces and the strings, the pony names, that are between them.

  Notice there is a semicolon after the last curly brace. This ends the `List` declaration statement.

- The code between lines 14 and 15:

  ```
  Dictionary<int, string> ponyDictionary = new Dictionary<int,
    string>() {{10, "Nightmare Moon"}, {20, "Rainbow Dash"}};
  ```

  This is a single statement. It's on two lines to make it fit the screenshot.

  Line 15 shows the **Collection Initializer** that's been added to the usual `Dictionary` declaration.

  Each key and value pony name is between two curly braces, then all the key/value pair combinations being initialized are between two curly braces.

Pop quiz – understanding an array and a List

Q1. In an array or a List, what is an element?

Q2. In an array or a List, what is the index number of the first element?

Q3. Can a single array, or a single List, store different types of data?

Q4. How can you add more elements to an array to make room for more data?

Looping though lists to make decisions

These previous array, List, and Dictionary examples showed how to get data into them, and how they store data. It's now time to learn how to loop through the data to retrieve the needed data.

Here are some common ways to perform loops:

- `foreach` loop
- `for` loop
- `while` loop

Using the foreach loop

When working with Collections such as an array, a list or dictionary, the preferred way to cycle through the elements and retrieve data is to use the `foreach` loop.

Time for action – using foreach loops to retrieve data

We're going to create an array, a list and a dictionary, then loop through each one to retrieve the desired data from each one by using `foreach` loops.

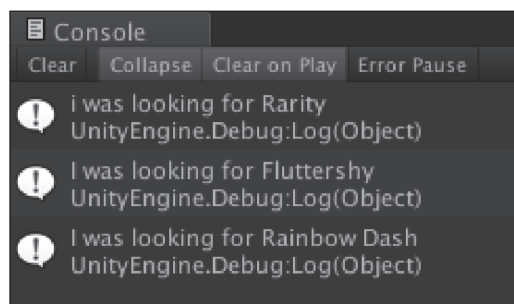
1. Modify `LearningScript` as shown in the next screenshot.
2. Save the file.

3. In Unity, click on Play.

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

public class LearningScript : MonoBehaviour
{

    void Start ()
    {
        string[] ponyArray = new string[]
            {"AppleJack", "Rarity"};

        foreach(string pony in ponyArray)
        {
            if(pony == "Rarity")
                Debug.Log("I was looking for " + pony);
        }

        List<string> ponyList = new List<string>()
            {"Princess Cadence", "Fluttershy"};

        foreach(string pony in ponyList)
        {
            if(pony == "Fluttershy")
                Debug.Log("I was looking for " + pony);
        }

        Dictionary<int, string> ponyDictionary = new Dictionary<int, string>()
            {{10, "Nightmare Moon"}, {20, "Rainbow Dash"}};

        foreach(KeyValuePair<int, string> pony in ponyDictionary)
        {
            if(pony.Key == 20)
                Debug.Log("I was looking for " + pony.Value);
        }
    }
}
```

What just happened?

As we looped through each list, we decided which data to display to the **Console**:



The analysis of the code is as follows:

For each list we created, we populated them using a Collection Initializer.

- The code between lines 9 and 10 with its description:

  ```
  string[] ponyArray = new string[] {"AppleJack", "Rarity"};
  ```

  A `string` array named `ponyArray` is declared and two strings are added.

- The code on line 12 with its description is as follows:

  ```
  foreach(string pony in ponyArray)
  ```

  A `foreach` loop is used to retrieve one element, a pony name string, stored in `ponyArray`.

  A variable is declared named `pony` to hold the retrieved pony name.

  Once a pony name is retrieved, the `foreach` code block, lines 13 to 16, is executed.

  This looping continues until each element in `ponyArray` has been retrieved and tested in the code block.

- The code on line 14 with its description is as follows:

  ```
  if(pony == "Rarity");
  ```

  If the retrieved string stored in `pony` is equal to `"Rarity"`, then line 15 executes.

- The code on line 15 with its description is as follows:

  ```
  Debug.Log("I was looking for " + pony);
  ```

  The string I was looking for plus the string value stored in pony is displayed in the **Console**.

- The code between lines 18 and 19 with its description:

  ```
  List<string> ponyList = new List<string>() {"Princess Cadence",
    "Fluttershy"};
  ```

  A `List` named `ponyList` is declared that will store the `string` type, and two strings are added.

- The code on line 21 with its description is as follows:

  ```
  foreach(string pony in ponyList)
  ```

  A `foreach` loop is used to retrieve one element, a pony name `string`, stored in `ponyList`.

  A variable is declared named `pony` to hold the retrieved pony name.

Once a pony name is retrieved, the `foreach` code block (that is, lines 22 to 25) is executed.

This looping continues until each element in `ponyList` has been retrieved and tested in the code block.

- The code on line 23 with its description is as follows:

  ```
  if(pony == "Fluttershy")
  ```

  If the retrieved string stored in `pony` is equal to `"Fluttershy"`, then line 24 executes.

- The code on line 24 with its description is as follows:

  ```
  Debug.Log("I was looking for " + pony);
  ```

  The string I was looking for plus the string value stored in `pony` is displayed in the **Console**.

- The code between lines 27 and 28 with its description:

  ```
  Dictionary<int, string> ponyDictionary = new Dictionary<int,
    string>() {{10, "Nightmare Moon"}, {20, "Rainbow Dash"}};
  ```

  A `Dictionary` named `ponyDictionary` is declared with key and value of type `<int, string>`, and two key/value pairs are added.

- The code on line 30 with its description is as follows:

  ```
  foreach(KeyValuePair<int, string> pony in ponyDictionary)
  ```

  A `foreach` loop is used to retrieve one `KeyValuePair`, a key and value, stored in `ponyDictionary`.

  A variable is declared named `pony` to hold the retrieved `KeyValuePair`.

  Once a key value and a pony name string are retrieved, the `foreach` code block (that is, lines 31 to 34) is executed.

  This looping continues until each `KeyValuePair` in `ponyDictionary` has been retrieved and tested in the code block.

- The code on lines32 with its description is as follows:

  ```
  if(pony.Key == 20)
  ```

  If the retrieved `Key` stored in pony is equal to `20`, then line 33 executes.

- The code on line 33 with its description is as follows:

  ```
  Debug.Log("I was looking for " + pony.Value);
  ```

  The string `I was looking for` plus the string value stored in `pony.Key` is displayed in the **Console**.

Using the for loop

The best description I've found for a `for` loop: "Allows a code block to be executed a specific number of times."

The syntax of a `for` loop:

```
for (initializer; condition; iterator)
  {
    code block
  }
```

Notice the three parts inside the parentheses are separated by semicolons, not commas.

Time for action – selecting a pony from a List using a for loop

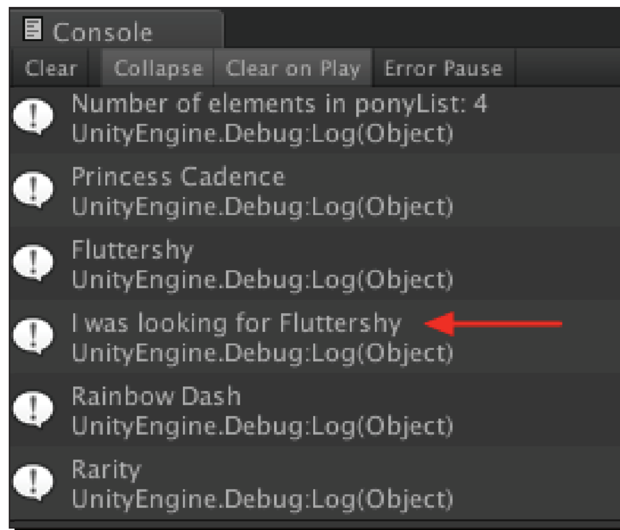
Let's add four pony names to a `List`. Retrieve and display the number of elements in the `List`. Then use a `for` loop to display each pony name, and select one of them:

1. Modify `LearningScript` as shown in the next screenshot.
2. Save the file.
3. In Unity, click on Play.

```
1  using UnityEngine;
2  using System.Collections.Generic;
3
4  public class LearningScript : MonoBehaviour
5  {
6      void Start ()
7      {
8          List<string> ponyList = new List<string>()
9              {"Princess Cadence", "Fluttershy", "Rainbow Dash", "Rarity"};
10
11         Debug.Log("Number of elements in ponyList: " + ponyList.Count);
12
13         for(int i = 0; i < ponyList.Count; i++)
14         {
15             Debug.Log(ponyList[i]);
16
17             if(ponyList[i] == "Fluttershy")
18                 Debug.Log("I was looking for " + ponyList[i]);
19         }
20     }
21 }
```

What just happened?

The following screenshot shows the number of elements in `ponyList`, the names of the ponies we added to `ponyList`, and the pony we were looking for:



The analysis of the code is as follows:

- The code between lines 8 and 9 with its description:

  ```
  List<string> ponyList = new List<string>() {"Princess
    Cadence", "Fluttershy", "Rainbow Dash", "Rarity"};
  ```

  A `List` named `ponyList` is declared that will store the `string` type.

  Four strings are added of the pony names.

- The code on line 11 with its description is as follows:

  ```
  Debug.Log("Number of elements in ponyList: " + ponyList.Count);
  ```

  The string `Number of elements in ponyList:` plus the number of elements used in `ponyList` is displayed in the **Console**.

  `ponyList.Count` is using dot syntax to access the `Count` property, a variable that stores the number of elements in a `List`.

  Four names were added to `ponyList`, so it has four elements to store the string names.

- The code on line 13 with its description is as follows:

  ```
  for(int i = 0; i < ponyList.Count; i++)
  ```

  The `for loop` is created.

  The initializer is simply a declared variable that's assigned a value.

  We declared a variable `i` of type `int`, and assigned it the a value of `0`. Why?

  The first index number in a `List` is `0`.

  The condition is checked for true before the code block, lines 14 to 19, is allowed to be executed.

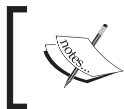
  When our `for` loop first begins, the variable `i` is equal to `0`, and `ponyList.Count` is equal to `4`, therefore `0` is less than `4`, which is `true`. Therefore the `for` loop code block is allowed to execute.

  The iterator, `i++`, now adds 1 to `i`, making `i` now equal to `1`.

  `i++` is the same thing as writing `i = i + 1`, which means that you are taking the value in `i` and adding 1, then assigning that to `i`.

  The loop repeats until the condition becomes `false`.

  After four times through the loop, `i` is now equal to 4, therefore the condition is now `false` because `i` is not less than `4`, so the loop is finished.



The letter "i" is typically used as the variable name in a for loop. It's tradition. If you happen to have nested for loops, then the variable names used will be the letters j, k, l, and so on, as needed.

- The code on lines 15 with its description is as follows:

  ```
  Debug.Log(ponyList[i]);
  ```

  The elements in `ponyList` are being access using the index number.

  As the `for` loop is executed for the first time, `i` is equal to `0`, therefore

  `ponyList[i]` is actually `ponyList[0]`, the element at index 0.

  The element at index 0 is storing `Princess Cadence`.

  After each iteration through the `for` loop, 1 is added to `i`, therefore the next trip through the for loop, `i` will be 1.

  `ponyList[i]` will actually be `ponyList[1]`, the next element at index 1.

  The result is all four ponies will be accessed and displayed in the **Console**.

- The code on lines 17 with its description is as follows:

  ```
  if(ponyList[i] == "Fluttershy")
  ```

  During each iteration through the code block, this if statement is checking to see if the name retrieved from `ponyList` is equal to `"Fluttershy"`.

  When it is, line 18 is executed.

- The code on lines 18 with its description is as follows:

  ```
  Debug.Log("I was looking for " + ponyList[i]);
  ```

  The string `I was looking for` plus the name `Fluttershy` is displayed in the **Console**.

Using the while loop

The `while` loop executes a code block until a specified expression evaluates to false.

A `while` loop is very similar to a for loop. It's like breaking the `for` loop into component parts:

```
The syntax of a while loop:
initializer
while (condition)
{
  code block
  iterator
}
```

Time for action – finding data and breakout of the while loop

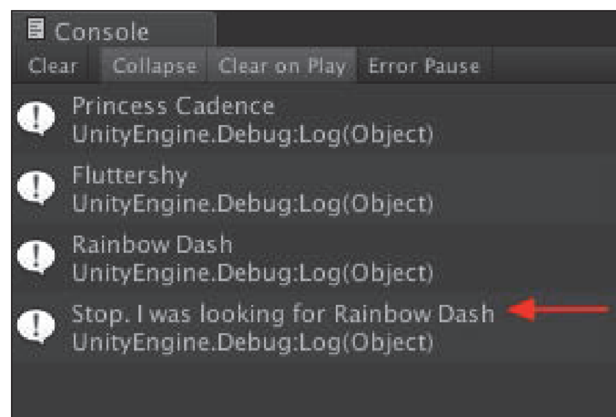
We're going to do something a little different in this loop. Once we find the pony we want, we'll breakout of the while loop. This is handy when looping through a large list of objects. When the desired data is found, there's no sense in continuing to loop through the rest of the list:

1. Modify `LearningScript` as shown in the next screenshot.
2. Save the file.
3. In Unity, click on Play.

```
1  using UnityEngine;
2  using System.Collections.Generic;
3
4  public class LearningScript : MonoBehaviour
5  {
6      void Start ()
7      {
8          List<string> ponyList = new List<string>()
9              {"Princess Cadence", "Fluttershy", "Rainbow Dash", "Rarity"};
10
11         int i = 0;
12         while(i < ponyList.Count)
13         {
14             Debug.Log(ponyList[i]);
15
16             if(ponyList[i] == "Rainbow Dash")
17             {
18                 Debug.Log("Stop. I was looking for " + ponyList[i]);
19                 break;
20             }
21             i++;
22         }
23     }
24 }
```

What just happened?

If we have been searching for `Fluttershy` instead of `Rainbow Dash`, and not included the `break` keyword on line 19, the output would have been exactly the same as the `for` loop example. In fact, the `break` keyword could have also have been used to breakout of the for loop.



I will skip explaining lines of code that are identical in the `for` loop example.

The analysis of the code is as follows:

- The code on line 11 with its description is as follows:

  ```
  int i = 0;
  ```

  The initializer is declared and assigned the value of 1.

- The code on line 12 with its description is as follows:

  ```
  while(i < ponyList.Count)
  ```

  The `while` loop is declared with the condition.

  Since `i` is `0`, it is less than `ponyList.Count`, which is `4`, the condition is true.

  The `while` loop code block (that is, lines 13 to 22), is executed.

- The code on line 16 with its description is as follows:

  ```
  if(ponyList[i] == "Rainbow Dash")
  ```

  During each iteration through the code block, this `if` statement is checking to see if the name retrieved from `ponyList` is equal to `Rainbow Dash`.

  When it is, the code block of lines 17 to 20 is executed.

  When it isn't, line 21 is the next line that is executed.

- The code on line 21 with its description is as follows:

  ```
  i++;
  ```

  The iterator `i` is incremented by 1 and the loop repeats back to line 12 to check the condition again.

  The loop repeats until `i` is equal to 4, making the condition false which exits the loop.

- The code on line 18 with its description is as follows:

  ```
  Debug.Log("Stop. I was looking for " + ponyList[i]);
  ```

  The string `Stop. I was looking for` plus the name `Rainbow Dash` is displayed in the **Console**.

- The code on line 19 with its description is as follows:

  ```
  break;
  ```

  `break` is a C# keyword that alters code flow.

  Code execution immediately leaves this `while` loop code block and continues to the first statement following the code block.

  There is no statement following the while loop, the script is finished.

Have a go hero – changing the pony name being searched

On line 16, change the pony name being searched and observe how it changes the number of pony names displayed in the **Console** before stopping.

Summary

There are unlimited ways to make decisions in code, however, we covered many of the common ways. The if statement is how the majority of decisions are made, including the if-else statements. Then we covered some of the sources that require making decisions, like user input, and using loops to evaluate data stored in arrays, lists and dictionaries. None of this is complicated. It's just a process of simple, logical steps.

Now that we've learned about the fundamentals of programming for writing scripts, it's time to dig into the world of objects. Since everything in Unity is an object, you need to know how to access the Components of an object, and how to communication between objects, by using dot syntax. You have seen some dot syntax used already in the examples we've coded. In next chapter, you will see how those dots work.

6
Using Dot Syntax for Object Communication

Scripts do many things by accessing the features built into Unity and third-party plugins. The Unity Scripting Reference is our link to the built-in Unity features. The thing is, exactly how do we invoke all of those Unity features?

So far all we've covered is basic C# programming. Granted, the example code we've seen has included some Dot Syntax, such as Debug.Log() to send output to Unity's Console. This is one of those Unity features. In the last chapter, we even saw some more Dot Syntax, pony.Key and pony.Value, which has nothing to do with Unity. These are just C# OOP (Object Oriented Programming) related features.

In both cases, there's some type of communication taking place to access methods and data to make things happen. To the beginner, those dots maybe odd looking and confusing, and they may ask, "What's the deal with all those darn dots between words?" Well, if you've been using the Internet and paid any attention at all, you've been using those dots, probably for years, and didn't pay much attention to them.

We see how to access the power of Dot Syntax as we cover the following sections:

- Dot Syntax being just an address
- Working with objects
- Using Dot Syntax in a script
- Accessing GameObjects using drag-and-drop versus writing code

So let's get on with it...

Using Dot Syntax is like addressing a letter

Ever seen something like this?

`www.unity3d.com`

That's right, a web address. Gee, I wonder why it's called a web **address**?

Simplifying the dots in Dot Syntax

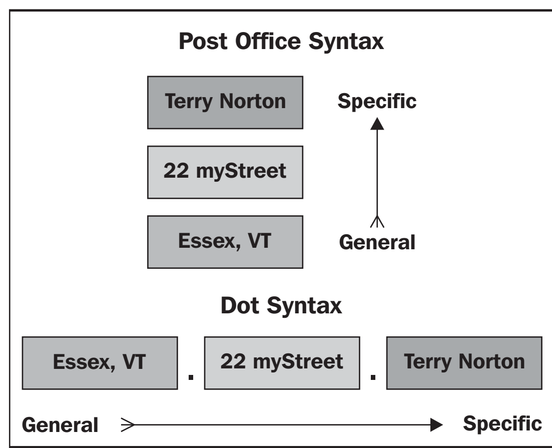
Here is a fictitious mailing address:

Terry Norton

22 myStreet

Essex, VT

You've understood how to read an address like this since you were a kid. Let's take a look at it again using a different format:



Looking at that, Dot Syntax isn't so confusing. It's just an address in a different format, in a way to locate things. Here's an example: imagine we met in Europe somewhere, and I ask you to get my sunglasses using only this information:

USA.Vermont.Essex.22 myStreet.2ndFloor.office.desk.center drawer.sunglasses

Would you have any problem locating them?

Using access modifiers for variables and methods

I could foresee one big issue trying to retrieve my sunglasses. My house isn't open to the public; it's a private residence, so the door is locked. This means you don't have access to the sunglasses.

The same rules of access apply to member variables and methods of a class or script.

In C#, when we create a member variable or method in a script, it is private by default. We can also explicitly specify that it's `private`.

Here, `private` means:

- A variable will not show in the **Inspector** as a Component property
- The variable or method will not be accessible from other scripts

We can specify a variable or method to be `public`.

Here, `public` means:

- A variable will show in the **Inspector** as a Component property
- The variable or method will be accessible from other scripts

Dot Syntax is the system used to locate and communicate with a variable or method in an object. To understand how to use Dot Syntax, we have to know the relationship between a class and its objects.

> A script always has access to its own member variable and methods whether they're `private` or `public`.

Working with objects is a class act

I'm throwing the word **object** around like you were born with the knowledge of what an object is. Actually you do know what it means. The coffee cup you may have in your hand is an object, a real one. That UFO flying around at night is an object; even if you can't identify it. In Unity, you may have a flying saucer in your Scene, but it's obviously not a real flying saucer, it's a virtual one. However, in the virtual world of gaming, most people would consider things they can see on the screen as objects.

If you can expand your mind just a little bit more, perhaps you can accept that not all objects in Unity have to be something you can see in a game Scene. In fact, the vast majority of objects in Unity are not visually in the Scene.

In a computer, an object is just a small section of your computer's memory that acts like a container. The container can have some data stored in variables and some methods to work with the data.

The best example I can show you is the object you've been using since you started this book.

In MonoDevelop, we've been working with the script called `LearningScript`. In Unity we use the general term **Script**, but it's actually a class, which means it's a definition of a type of container. Look at line 4 of the file:

```
public class LearningScript : MonoBehaviour
```

See that second word? That means that `LearningScript` is a class. In this class, we defined its member variables and methods. Any variable not declared in a method is a member variable of the class.

In *Chapter 2*, *Introducing the Building Blocks for Unity Scripts* I told you about the magic that happens when we attach the script (class) to a GameObject. Shazam!! The script becomes a Component object, a type of container for the GameObject that we defined as having some variables to store data and some methods to work that that data.

Besides the visual mesh in the Scene, can you visualize in your mind that a GameObject is just a bunch of different types of Component objects assembled together to construct that GameObject?

Each of those individual Components shown in the **Inspector** will become an object in our computer's memory when we click on the **Play** button.

Select any GameObject in the **Scene**, then look at the **Inspector**. For example, select the **Main Camera** GameObject. There are several Components on the **Main Camera** GameObject. Look at each of those defined Components. Every one of those Components started off as a class file in Unity, defining a type of container of variables and methods. We don't see or modify those Unity class files, but they're in Unity somewhere.

- The name of the class is also known as the object type of the object that will be created in memory from that class, when the **Play** button is clicked.
- Just like an `int`, or a `string` is a type of data, the name of a class is also a type of data.
- This means that when we declare a variable and specify the type of data it will store, it can just as easily store a reference to an object of the `LearningScript` type, as shown in the following line of code:

  ```
  LearningScript myVariable;
  ```

- Storing a reference to an object in a variable does not mean we are storing the actual object. It means we are storing the location in memory of that object. It's just a reference that points to the object in memory so that the computer knows where to access the object's data and methods.This means we can have several variables storing a reference to the same object, but there's still only one actual object in memory.

A script is just a file on your hard drive, and there's only ever one file. The class file simply defines a type of container of variables and methods that will become a Component object in the memory when you click on **Play**. You can attach the script to many GameObjects, but there's still only one file on your hard drive.

Attaching a **Script** to a GameObject is like placing a sticky-note on the GameObject. When we click on the **Play** button, Unity looks at our GameObject, sees the sticky-note which says, "This GameObject is supposed to have a Component of type `LearningScript`. Make some room in the computer's memory to hold this object of variables and methods as described in the `LearningScript` class file."

If we were to attach `LearningScript` to 1000 GameObjects, and click on **Play**, there will be 1000 separate sections created in your computer's memory that each stores an object of type `LearningScript`. Each one has its own set of variables and methods, as described by the `script` file. Each one of those 1000 sections of computer memory is a separate Component object of its respective GameObject.

Even though the object created from a class is called a Component by Unity; in more general C# terms, each object that gets created from a class is called an instance object. A Component object and an instance object are the same thing.

Using Dot Syntax in a script

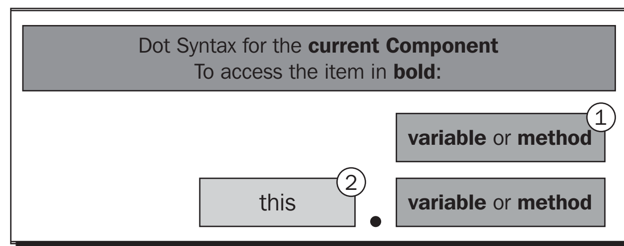
Now that you know that each Component object resides in computer memory, storing data in variables, it's time to use Dot Syntax to access those Component variables and methods.

Accessing a Component's own variables and methods

Dot Syntax can be used to access any pubic variable or method on any Component on any GameObject. Even though a Component always has access to its own variables and methods, we can still use Dot Syntax if we want.

In order to have access to a variable or method, we have to know its location. Let's start by looking in `LearningScript`.

Here's an overview of how to access a variable or method from within the current Component:



Time for action – accessing a variable in the current Component

Let's look at accessing a variable in `LearningScript` from inside `LearningScript`.

1. Modify `LearningScript` as shown in the following figure:

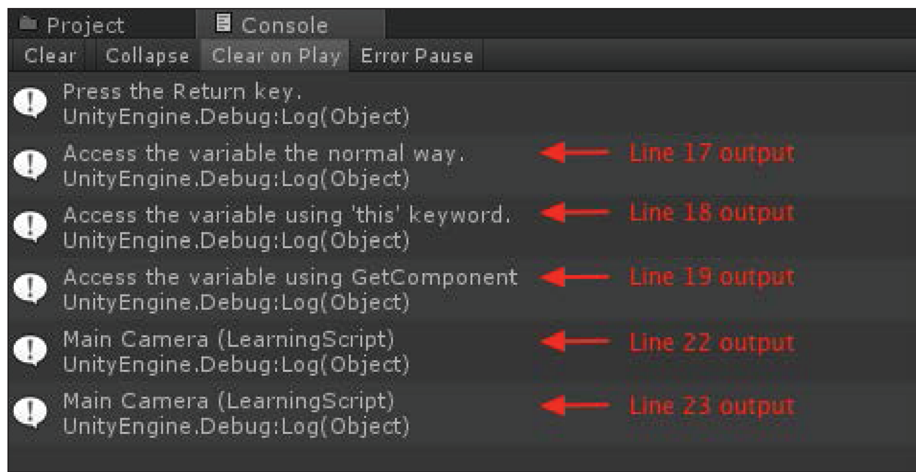
```
using UnityEngine;
using System.Collections.Generic;

public class LearningScript : MonoBehaviour
{
    string myString = "Access the variable ";

    void Start ()
    {
        Debug.Log("Press the Return key.");
    }

    void Update ()
    {
        if(Input.GetKeyDown(KeyCode.Return))
        {
            Debug.Log(myString + "the normal way.");
            Debug.Log(this.myString + "using 'this' keyword.");
            Debug.Log(GetComponent<LearningScript>().myString
                + "using GetComponent");

            Debug.Log(this);
            Debug.Log(GetComponent<LearningScript>());
        }
    }
}
```

this keyword

2. Save the file.
3. In Unity, click on **Play.**

What just happened?

Here are the outputs in the **Console**:



An analysis of the code shown in the previous code screenshot is as follows:

Line 6: `string myString = "Access the variable ";`

- `myString` is the variable that will be accessed
- Notice that it's `private` by default, yet it can still be accessed

Line 17: `Debug.Log(myString + "the normal way.");`

- This is how we have been accessing the value stored in a variable, by just using the variable name
- The `string` value in `myString`, `Accessing this variable`, is substituted for the variable name
- `myString` is being accessed without using Dot Syntax or `GetComponent()`, because a script always has access to its own variables and methods

Line 18: `Debug.Log(this.myString + "using 'this' keyword.");`

- `myString` is being accessed using Dot Syntax
- The `this` keyword refers to the current instance of the class, the current Component

Line 19: `Debug.Log(GetComponent<LearningScript>().myString + "using GetComponent.");`

- `myString` is being accessed using Dot Syntax again
- This time, the generic `GetComponent<T>()` method is retrieving the `LearningScript` Component

Line 22: `Debug.Log(this);`

- Using `this`, the Component is sent to the **Console** so we can see that `this` is the current `LearningScript` Component object

Line 23: `Debug.Log(GetComponent<LearningScript>());`

- Using `GetComponent<LearningScript>()`, the Component is sent to the **Console**, so we can see this also is the current `LearningScript` Component object
- The `this` keyword and `GetComponent<LearningScript>()` are both retrieving the same `LearningScript` Component object

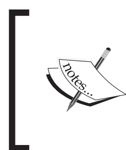
Whoa!! What's with line 18?

Notice item 1 in the graphic under the section, *Accessing a Component's own variables and methods*? This is the usual way we will access variables and methods in the current script; no Dot Syntax required. This is how we've been doing it from the beginning of this book. It's how we will probably continue to access them. However, we do have the option of accessing the variables and methods in the current Component object using Dot Syntax.

As you can see from the output of lines 17 and 18, the value stored in `myString` is substituted no matter how we access `myString`.

So if we really wanted to, we could use the `GetComponent()` method to retrieve the current Component object of the `LearningScript` class in memory, then use Dot Syntax to access `myString`. However, C# provides a shortcut to get the current Component object by using the `this` keyword.

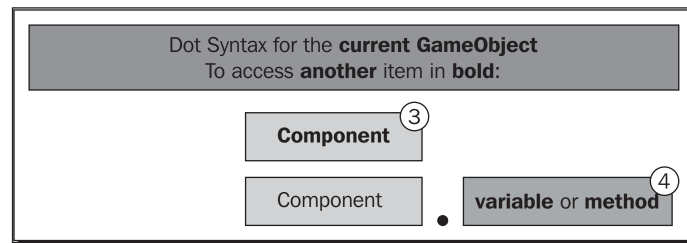
Item 2 in the graphic is the syntax used in line 18. In this example, the keyword `this` simply means the current instance object of the `LearningScript` class, the current Component.



Why do I even mention using `this` at this time? Later on when we get into the State Machine, we will be using `this`. I want you to be aware of what `this` is, a substitute for the current instance object of a class.

Accessing another Component on the current GameObject

Now we start to just touch on the real power of Dot Syntax, communicating with other objects to access variable data and methods. We will now communicate with another Component on the same GameObject, the **Main Camera**. Remember, `LearningScript` is attached to the **Main Camera** already. The following diagram will explain how this is done:



Time for action – communicating with another Component on the Main Camera

Let's create another script with a variable and a method, and attach it to the **Main Camera**, then have `LearningScript` communicate with it:

1. In Unity, create another **C# Script** and name it `TalkToMe`.
2. Make a `public string` variable named `hereItIs`.
3. Assign some text to `hereItIs`.
4. Make a `public` method named `MakeMeTalk()`.
5. Have `MakeMeTalk()` output some text to the **Console**.
6. Attach `MakeMeTalk()` to the **Main Camera**. Now the code should look something like this:

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class TalkToMe : MonoBehaviour
5  {
6      public string hereItIs = "This is the TalkToMe variable";
7
8      public void MakeMeTalk ()
9      {
10         Debug.Log("This is the TalkToMe method");
11     }
12 }
```

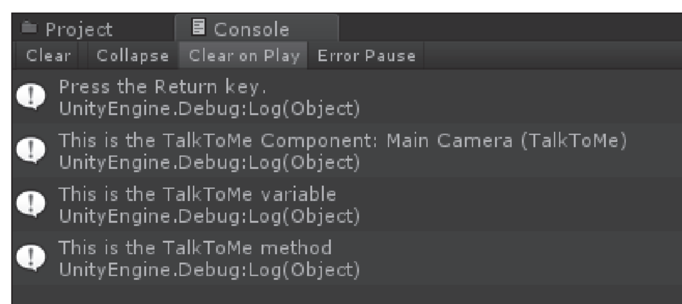
7. Modify `LearningScript` to retrieve the `TalkToMe` Component.
8. Modify `LearningScript` to retrieve the data in `hereItIs`.
9. Modify `LearningScript` to call the `MakeMeTalk()` method. Now the code snippet should look as follows:

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

public class LearningScript : MonoBehaviour
{
    TalkToMe otherComponent;

    void Start ()
    {
        otherComponent = GetComponent<TalkToMe>();

        Debug.Log("Press the Return key.");
    }

    void Update ()
    {
        if(Input.GetKeyDown(KeyCode.Return))
        {
            Debug.Log("This is the TalkToMe Component: " + otherComponent);
            Debug.Log(otherComponent.hereItIs);
            otherComponent.MakeMeTalk();
        }
    }
}
```

10. Save your scripts.
11. Click on **Play** in Unity.

What just happened?

Here's the output:



The `LearningScript` Component code retrieved a variable and called a method on the `TalkToMe` Component. Let's follow the code flow with these two Components.

An analysis of the code shown in the previous code screenshot is as follows:
On LearningScript:

Line 6: TalkToMe otherComponent;

- A variable otherComponent is declared to store a value of type TalkToMe
- A TalkToMe Component object will be created and stored in the variable otherComponent

Line 10: otherComponent = GetComponent<TalkToMe>();

- Remember, this is in the Start() method which Unity calls only once to initialize variables.
- The generic version of the GetComponent<T>() method is called to retrieve a reference to the TalkToMe Component object. This is item 3 on the previous graphic under the section, *Accessing another Component on the current GameObject*.
- The <T> part is the type of Component, the class name, that the method will return.
- This reference is stored in the variable otherComponent. Why?So that every time we need to use the TalkToMe Component with Dot Syntax, we can just use the reference stored in otherComponent instead of having to use GetComponent<TalkToMe>() each time.

Line 19: Debug.Log("This is the TalkToMe Component: " + otherComponent);

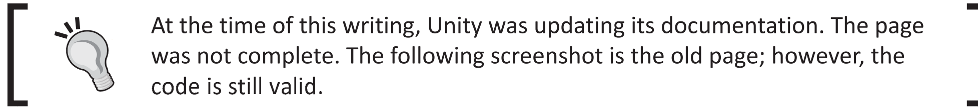
- This line of code sends the value stored in otherComponent to the Unity **Console** so we can see the reference that's pointing to the TalkToMe Component object

Line 20: Debug.Log(otherComponent.hereItIs);

- Dot Syntax is used to locate and retrieve the value stored in the variable hereItIs of the TalkToMe Component object. This is item 4 in the graphic under the section, *Accessing another Component on the current GameObject*.
- The hereItIs variable is declared on line 6 of the TalkToMe class.
- Notice that hereItIs is public so that it can be accessed from other scripts.
- If we didn't use otherComponent, we would have written the Dot Syntax expression like the following line of code:

  ```
  GetComponent<TalkToMe>().hereItIs
  ```

- The following is a screenshot of the **Scripting Reference** example:



At the time of this writing, Unity was updating its documentation. The page was not complete. The following screenshot is the old page; however, the code is still valid.

GameObject.GetComponent

function **GetComponent (type : Type) :** Component

Description
Returns the component of Type type if the game object has one attached, null if it doesn't. You can access both builtin components or scripts with this function.

GetComponent is the primary way of accessing other components. From javascript the type of a script is always the name of the script as seen in the project view. Example:

C#

```
using UnityEngine;
using System.Collections;

public class example : MonoBehaviour {
    void Start() {
        Transform curTransform;
        curTransform = gameObject.GetComponent<Transform>();
        curTransform = gameObject.transform;
    }
    void Update() {
        ScriptName other = gameObject.GetComponent<ScriptName>();
        other.DoSomething();
        other.someVariable = 5;
    }
}
```

function **GetComponent.<T> () : T**

Click the link for more info

Description
Generic version. See the Generic Functions page for more details.

Line 21: `otherComponent.MakeMeTalk();`

- Dot Syntax is used to locate and call the `MakeMeTalk()` method of the `TalkToMe` Component object
- Code flow now jumps over to the `TalkToMe` class. This is also item 4 in the graphic under the section, *Accessing another Component on the current GameObject*.

On `TalkToMe`:

Line 8: `public void MakeMeTalk()`

- The `MakeMeTalk()` method is `public` so that it can be called from other scripts
- Its code block simply sends a string of text to the Unity **Console**
- The code block ends and code flow returns to the `LearningScript` class

On `LearningScript`:

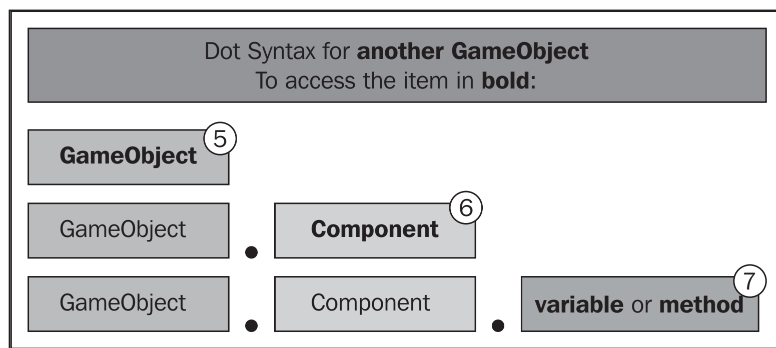
Line 22: }

- Code flow has now reached the end of the `if` statement which began on line 17, and is waiting to detect if we press the *Return* key again

Before you proceed further with the next section, remove the `TalkToMe` Component from the **Main Camera**. We are done with this script so there's no sense in having any of its Components hanging around.

Accessing other GameObjects and their Components

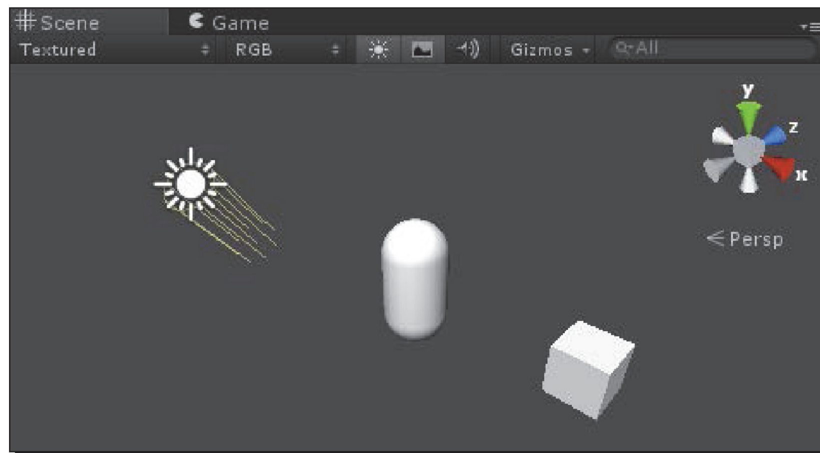
You just learned to access Components on the same GameObject. Now it's time to access other Gameobjects, and their Components using Dot Syntax.



Time for action – creating two GameObjects and a new script

I want you to create one script that will be attached to two GameObjects. The script will have two methods that will cause the GameObjects to rotate left and right. This will show you that from a single script file, two separate Component objects will be created in the memory. Each Component object is a separate instance object with no absolutely knowledge of the other.

1. In your **Scene**, create two GameObjects, `Capsule` and `Cube`.
2. Add a **Directional Light** to the **Scene** so you can easily see the GameObjects.
3. Here's my **Scene** as an example:



4. Create a new C# Script and name it Spinner.
5. Code the script as shown in the following screenshot:

```
 1  using UnityEngine;
 2  using System.Collections;
 3
 4  public class Spinner : MonoBehaviour
 5  {
 6      public void SpinLeft()
 7      {
 8          transform.Rotate(0,0,60 * Time.deltaTime);
 9      }
10
11      public void SpinRight()
12      {
13          transform.Rotate(0,0,-60 * Time.deltaTime);
14      }
15  }
```

This means: 1 per second
x-axis
y-axis
z-axis

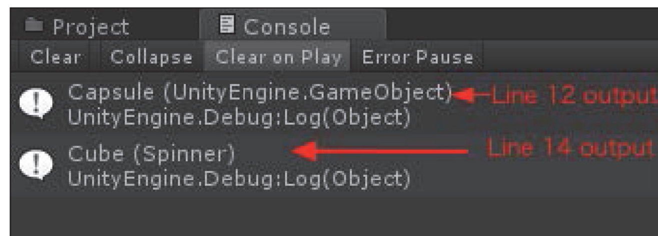
6. Attach the `Spinner` script to the **Capsule** and the **Cube** GameObjects.
7. Modify `LeaningScript` as shown in the following screenshot:

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

public class LearningScript : MonoBehaviour
{
    GameObject capsuleGO;
    Spinner cubeComp;

    void Start ()
    {
        capsuleGO = GameObject.Find("Capsule");
        Debug.Log(capsuleGO);
        cubeComp = GameObject.Find("Cube").GetComponent<Spinner>();
        Debug.Log(cubeComp);
    }

    void Update ()
    {
        if(Input.GetKey(KeyCode.LeftArrow))
        {
            capsuleGO.GetComponent<Spinner>().SpinLeft();
        }

        if(Input.GetKey(KeyCode.RightArrow))
        {
            capsuleGO.GetComponent<Spinner>().SpinRight();
        }

        if(Input.GetKey(KeyCode.UpArrow))
        {
            cubeComp.SpinLeft();
        }

        if(Input.GetKey(KeyCode.DownArrow))
        {
            cubeComp.SpinRight();
        }
    }
}
```

8. Save the file.
9. In Unity, click on **Play**.

What just happened?

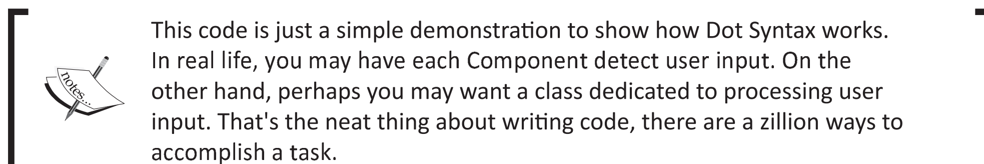
Here's the output to the **Console**:



Now press the left and right arrow keys to make the Capsule spin, and the up and down arrow keys to make the Cube spin.

You created one script named `Spinner`, then attached the script to two separate GameObjects. When you click on **Play**, two separate `Spinner` Component objects are created in the computer memory. This is an example of how the `Spinner` class is just a blueprint, a description, of what each Component object created will be.

To access each `Spinner` Component from the `LearningScript` Component, you need to know about each GameObject that the `Spinner` Component is attached to.



This code is just a simple demonstration to show how Dot Syntax works. In real life, you may have each Component detect user input. On the other hand, perhaps you may want a class dedicated to processing user input. That's the neat thing about writing code, there are a zillion ways to accomplish a task.

An analysis of the code shown in the previous code screenshot is as follows:

On `LearningScript`:

Line 6: `GameObject capsuleGO;`

- A variable of type `GameObject` is declared
- The value this will store is a reference to the **Capsule** in the **Scene**

Line 7: `Spinner cubeComp;`

- A variable of type `Spinner` is declared
- The value this will store is a reference to a `Spinner` Component object created from the `Spinner` class

Line 9: `void Start()`

- The `Start()` method is used to allow the two variables to be initialized
- Remember, this method is called only once

Line 11: `capsuleGO = GameObject.Find("Capsule");`

- The `Find()` method of the `GameObject` class locates a GameObject in our **Scene**
- The reference to the **Capsule** GameObject is assigned to the variable `capsuleGO`
- This is item 5 in the previous graphic and also on the previous code screenshot

Line 12: `Debug.Log(capsuleGO);`

- This line was added just to show that the **Capsule** GameObject is in fact referenced in the variable `capsuleGO`

Line 13: `cubeComp = GameObject.Find("Cube").GetComponent<Spinner>();`

- This line shows how to retrieve a Component on a GameObject
- This retrieved reference to the `Spinner` Component object is on the **Cube** GameObject
- This is item 6 on the previous graphic and also on the previous code screenshot

Line 14: `Debug.Log(cubeComp);`

- This line was added just to show that the `Spinner` Component is part of the **Cube** GameObject, and is in fact referenced in the variable `cubeComp`

Line 19: `if(Input.GetKey(KeyCode.LeftArrow)`

- This `if` statement checks to see if the user has pressed the left arrow key
- If pressed, Line 21 of the code block is executed

Line 21: `capsuleGO.GetComponent<Spinner>().SpinLeft();`

- This line shows using Dot Syntax to locate a method in a Component of another GameObject.
- The `CapsuleGO` variable substitutes the reference to the **Capsule** GameObject
- The `Spinner` Component object is located on the **Capsule** GameObject
- The `SpinLeft()` method is called in the `Spinner` Component of the **Capsule** GameObject
- Code flow now jumps to the `Spinner` Component object

Spinner (on the Capsule):

Line 6: `public void SpinLeft()`

- This is the `SpinLeft()` method called from the `LearningScript` object
- Line 8 in the code block is executed

Line 8: `transform.Rotate(0, 0, 60 * Time.deltaTime);`

- The `Rotate()` method on the `Transform` Component object is called which causes the Capsule to spin around the z-axis
- Notice though, that the variable named `transform` is used in the Dot Syntax statement instead of the `GetComponent<Transform>()` method
- Unity has several built-in Components, such as the `Transform` Component class
- Find the `GameObject` class in the **Scripting Reference** and notice that one of the variables is named `transform`
- Instead of having to use the `GetComponent()` method on a GameObject, Unity has provided a convenient variable already assigned the value of the `Transform` Component
- The following screenshot shows the `transform` variable described in the **Scripting Reference**:

GameObject
Inherits from Object

Base class for all entities in Unity scenes.

See Also: Component.

Variables

isStatic	Editor only API that specifies if a game object is static.
transform	The Transform attached to this GameObject. (null if there is none attached).
rigidbody	The Rigidbody attached to this GameObject (Read Only). (null if there is none attached).
camera	The Camera attached to this GameObject (Read Only). (null if there is none attached).
light	The Light attached to this GameObject (Read Only). (null if there is none attached).
animation	The Animation attached to this GameObject (Read Only). (null if there is none attached).
constantForce	The ConstantForce attached to this GameObject (Read Only). (null if there is none attached).

- The `Rotate()` method shows 3 arguments being sent to the method.
- In this example, the Capsule is rotating 60 degrees per second on the z-axis.
- Code flow now returns to the `LearningScript` object.

On LearningScript:

Line 24: if(Input.GetKey(KeyCode.RightArrow)

- This if statement checks if the user has pressed the right arrow key
- If pressed, line 26 of the code block is executed

Line 26: capsuleGO.GetComponent<Spinner>().SpinRight();

- This is almost an exact repeat of line 21, except the SpinRight() method is being called

Line 29: if(Input.GetKey(KeyCode.UpArrow)

- This if statement checks if the user has pressed the up arrow key
- If pressed, line 31 of the code block is executed

Line 31: cubeComp.SpinLeft();

- This is different than lines 21 and 26
- Refer back to line 13. The cubeComp variable already stores the reference to the **Cube** GameObject and the Spinner Component object, thereforeJust the variable cubeComp is needed in the Dot Syntax to call the SpinLeft() method on the **Cube** GameObject
- Code flow is similar to line 8, except that the Cube rotates now

Line 34: if(Input.GetKey(KeyCode.DownArrow)

- This if statement checks to see if the user has pressed the down arrow key
- If pressed, line 36 of the code block is executed, spinning the Cube right

Have a go hero – creating and using a new variable named capsuleComp

In LearningScript, lines 21 and 31 perform the same functionality of calling the SpinLeft() method on their Spinner Components. Yet the code on each line is very different. The difference is that cubeComp already stores a reference to the Cube's Spinner Component. There is no capsuleComp variable to store a reference to the Capsule's Spinner Component.

Try creating a capsuleComp variable and store a reference to the Capsule's Spinner Component. Then change lines 21 and 26 to use capsuleComp.

Accessing GameObjects using drag-and-drop versus writing code

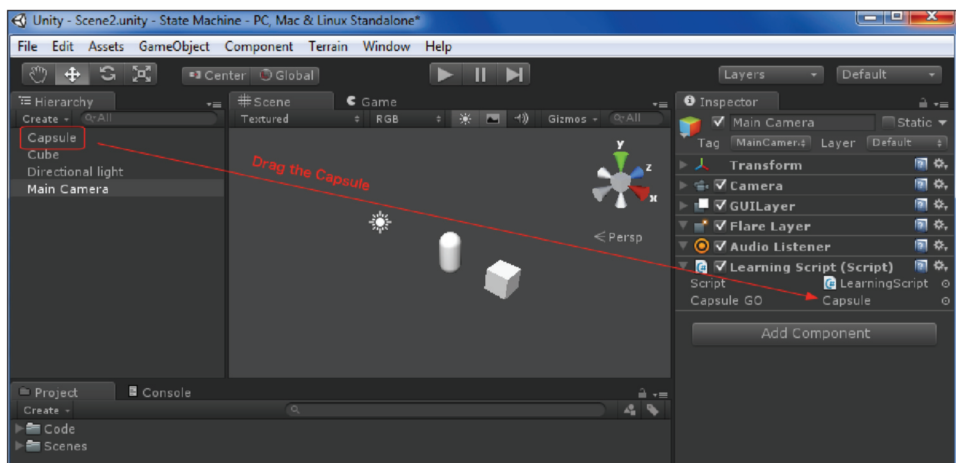
Unity has a rather neat feature that allows us to assign GameObjects to variables without writing the code. It definitely has its uses, however, if it's not really necessary, I recommend assigning GameObjects in code. Why?

> Six months from now, when you are the stranger looking at your own code, you may look at it and wonder why your code looks incomplete. It's your game though, so you can create it anyway you please. I'm just saying, don't go hog wild with drag-and-drop and then later wonder what it was you were trying to accomplish.

Time for action – trying drag-and-drop to assign a GameObject

Let's change a few lines of code in `LearningScript` to show how to assign the **Capsule** GameObject to the variable `capsuleGO` using drag-and-drop.

1. Either comment out line 11 using 2 forward slashes (`//`), or remove it.
2. On line 6, add the access modifier `public` like this: `public GameObject capsuleGO;`
3. Save the file.
4. In Unity, select the **Main Camera** GameObject.
5. Drag the **Capsule** to the **Capsule GO** field in the **Inspector**. The following screenshot shows how this is done.
6. Click on **Play**.



What just happened?

The **Capsule** GameObject is now assigned to the `capsuleGO` variable. We didn't have to write the code because Unity has done the assignment internally for us. Also, this doesn't change `LearningScript` in any way.

Pop quiz – understanding communication between objects

Q1. What is Dot Syntax, and what does it allow you to do?

Q2. When an object is assigned to a variable, what is actually stored in the variable?

Q3. Are there any limits to using Dot Syntax when trying to access variables and methods?

Q4. What is another way to assign GameObjects to variables besides writing code?

Summary

I hope you have discovered that Dot Syntax is actually a simple process for accessing other objects. It's this ability to communicate between objects that make OOP so powerful. Data is kept in objects, and methods are called on an object to get things done. Dot Syntax is just an address to easily access data and methods on objects.

All right, we've covered the very basics of C# scripting for Unity. Congratulations!

In the next chapter, I'm going to take you through a combination of Unity coding and general C# coding to actually apply your new knowledge. We will start looking at a State Machine to work with Unity. Yes, it's going to be a simple state machine to show the concepts. You've just barely learned C# scripting, so I'm going to ease you into some game creation which will help you see how to apply the concepts that you've just learned. Besides, bet you're darn sick and tired of constantly modifying, or completely changing `LearningScript`.