



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Mastering Unity Scripting

Learn advanced C# tips and techniques to make professional-grade games with Unity

Alan Thorn

[PACKT]
PUBLISHING

Mastering Unity Scripting

Learn advanced C# tips and techniques to make professional-grade games with Unity

Alan Thorn



BIRMINGHAM - MUMBAI

Mastering Unity Scripting

Copyright © 2015 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: January 2015

Production reference: 1230115

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

ISBN 978-1-78439-065-5

www.packtpub.com

Credits

Author

Alan Thorn

Project Coordinator

Kinjal Bari

Reviewers

Dylan Agis

John P. Doran

Alessandro Mochi

Ryan Watkins

Proofreaders

Samuel Redman Birch

Ameesha Green

Indexer

Rekha Nair

Commissioning Editor

Dipika Gaonkar

Production Coordinator

Shantanu N. Zagade

Acquisition Editor

Subho Gupta

Cover Work

Shantanu N. Zagade

Content Development Editors

Melita Lobo

Rikshith Shetty

Technical Editors

Shashank Desai

Pankaj Kadam

Copy Editors

Karuna Narayanan

Laxmi Subramanian

About the Author

Alan Thorn is a London-based game developer, freelance programmer, and author with over 13 years of industry experience. He founded Wax Lyrical Games in 2010, and is the creator of the award-winning game, *Baron Wittard: Nemesis of Ragnarok*. He is the author of 10 video-training courses and 11 books on game development, including *Unity 4 Fundamentals: Get Started at Making Games with Unity*, Focal Press, *UDK Game Development*, and *Pro Unity Game Development with C#, Apress*. He is also a visiting lecturer on the Game Design & Development Masters Program at the National Film and Television School.

Alan has worked as a freelancer on over 500 projects, including games, simulators, kiosks, serious games, and augmented reality software for game studios, museums, and theme parks worldwide. He is currently working on an upcoming adventure game, *Mega Bad Code*, for desktop computers and mobile devices. Alan enjoys graphics. He is fond of philosophy, yoga, and also likes to walk in the countryside. His e-mail ID is `directx_user_interfaces@hotmail.com`.

About the Reviewers

Dylan Agis is a programmer and game designer, currently doing freelance work on a few projects while also developing a few projects of his own. He has a strong background in C++ and C# as well as Unity, and loves to solve problems.

I would like to thank Packt Publishing for giving me the chance to review the book, and the author for making it an interesting read.

John P. Doran is a technical game designer who has been creating games for over 10 years. He has worked on an assortment of games in teams with members ranging from just himself to over 70 in student, mod, and professional projects.

He previously worked at LucasArts on *Star Wars: 1313* as a game design intern—the only junior designer on a team of seniors. He was also the lead instructor of DigiPen®-Ubisoft® Campus Game Programming Program, instructing graduate-level students in an intensive, advanced-level game programming curriculum.

John is currently a technical designer in DigiPen's Research & Development department. In addition to that, he also tutors and assists students on various subjects while giving lectures on game development, including C++, Unreal, Flash, Unity, and more.

He has been a technical reviewer for nine game development titles, and is the author of *Unity Game Development Blueprints*, *Getting Started with UDK*, *UDK Game Development [Video]*, and *Mastering UDK Game Development HOTSHOT*, all by Packt Publishing. He has also co-authored *UDK iOS Game Development Beginner's Guide*, Packt Publishing.

Alessandro Mochi has been playing video games since the Amstrad and NES era, tackling all the possible fields: PC, console, and mobile. Large or small video games are his love and passion. RPGs, strategy, action platformers... nothing can escape his grasp.

With a professional field degree in IT, a distinction in project management diploma, and fluent in Spanish, Italian, and English, he gained sound knowledge of many programs. New challenges are always welcome.

Currently a freelance designer and programmer, he helps young developers turn their concepts into reality. Always traveling all over the world, he is still easy to find on his portfolio at www.amochi-portfolio.com.

Ryan Watkins likes to party. He can be found on LinkedIn at www.linkedin.com/in/ryanswatkins/.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Unity C# Refresher	7
Why C#?	8
Creating script files	8
Instantiating scripts	12
Variables	13
Conditional statements	15
The if statement	15
The switch statement	18
Arrays	21
Loops	24
The foreach loop	24
The for loop	26
The while loop	27
Infinite loops	29
Functions	29
Events	32
Classes and object-oriented programming	34
Classes and inheritance	36
Classes and polymorphism	37
C# properties	42
Commenting	44
Variable visibility	47
The ? operator	48
SendMessage and BroadcastMessage	49
Summary	51

Chapter 2: Debugging	53
Compilation errors and the console	54
Debugging with Debug.Log – custom messages	57
Overriding the ToString method	59
Visual debugging	63
Error logging	66
Editor debugging	71
Using the profiler	74
Debugging with MonoDevelop – getting started	77
Debugging with MonoDevelop – the Watch window	82
Debugging with MonoDevelop – continue and stepping	85
Debugging with MonoDevelop – call stack	87
Debugging with MonoDevelop – the Immediate window	89
Debugging with MonoDevelop – conditional breakpoints	90
Debugging with MonoDevelop – tracepoints	91
Summary	94
Chapter 3: Singletons, Statics, GameObjects, and the World	95
The GameObject	96
Component interactions	99
GetComponent	99
Getting multiple components	101
Components and messages	103
GameObjects and the world	104
Finding GameObjects	105
Comparing objects	107
Getting the nearest object	108
Finding any object of a specified type	109
Clearing a path between GameObjects	110
Accessing object hierarchies	112
The world, time, and updates	113
Rule #1 – frames are precious	115
Rule #2 – motion must be relative to time	116
Immortal objects	117
Understanding singleton objects and statics	119
Summary	123
Chapter 4: Event-driven Programming	125
Events	126
Event management	130
Starting event management with interfaces	131
Creating an EventManager	133

Code folding in MonoDevelop with #region and #endregion	139
Using EventManager	140
Alternative with delegates	141
MonoBehaviour events	147
Mouse and tap events	147
Application focus and pausing	151
Summary	153
Chapter 5: Cameras, Rendering, and Scenes	155
Camera gizmos	156
Being seen	159
Detecting the object visibility	160
More on the object visibility	161
Frustum testing – renderers	162
Frustum testing – points	163
Frustum testing – occlusion	164
Camera vision – front and back	165
Orthographic cameras	167
Camera rendering and postprocessing	171
Camera shake	178
Cameras and animation	180
Follow cameras	180
Cameras and curves	183
Camera paths – iTween	186
Summary	189
Chapter 6: Working with Mono	191
Lists and collections	192
The List class	193
The Dictionary class	196
The Stack class	197
IEnumerable and IEnumerator	199
Iterating through enemies with IEnumerator	200
Strings and regular expressions	205
Null, empty strings, and white space	205
String comparison	206
String formatting	208
String looping	208
Creating strings	209
Searching strings	209
Regular expressions	210
Infinite arguments	211

Language Integrated Query	212
Linq and regular expressions	215
Working with Text Data Assets	216
Text Assets – static loading	216
Text Assets – loading from the local files	218
Text Assets – loading from the INI files	219
Text Assets – loading from the CSV files	221
Text Assets – loading from the Web	222
Summary	222
Chapter 7: Artificial Intelligence	223
Artificial Intelligence in games	224
Starting the project	225
Baking a navigation mesh	227
Starting an NPC agent	231
Finite State Machines in Mecanim	233
Finite State Machines in C# – getting started	240
Creating the Idle state	242
Creating the Patrol state	245
Creating the Chase state	249
Creating the Attack state	251
Creating the Seek-Health (or flee) state	253
Summary	256
Chapter 8: Customizing the Unity Editor	259
Batch renaming	259
C# attributes and reflection	265
Color blending	269
Property exposing	275
Localization	281
Summary	291
Chapter 9: Working with Textures, Models, and 2D	293
Skybox	294
Procedural meshes	300
Animating UVs – scrolling textures	308
Texture painting	310
Step 1 – creating a texture blending shader	312
Step 2 – creating a texture painting script	315
Step 3 – setting up texture painting	322
Summary	327

Chapter 10: Source Control and Other Tips	329
Git – source control	329
Step #1 – downloading	330
Step #2 – building a Unity project	332
Step #3 – configuring Unity for source control	333
Step #4 – creating a Git repository	335
Step #5 – ignoring files	336
Step #6 – creating the first commit	336
Step #7 – changing files	337
Step #8 – getting files from the repo	339
Step #9 – browsing the repo	342
Resources folder and external files	343
AssetBundles and external files	345
Persistent data and saved games	348
Summary	354
Index	355

Preface

Mastering Unity Scripting is a concise and dedicated exploration of some advanced, unconventional, and powerful ways to script games with C# in Unity. This makes the book very important right now because, although plenty of "beginner" literature and tutorials exist for Unity, comparatively little has been said of more advanced subjects in a dedicated and structured form. The book assumes you're already familiar with the Unity basics, such as asset importing, level designing, light-mapping, and basic scripting in either C# or JavaScript. From the very beginning, it looks at practical case studies and examples of how scripting can be applied creatively to achieve more complex ends, which include subjects such as Debugging, Artificial Intelligence, Customized Rendering, Editor Extending, Animation and Motion, and lots more. The central purpose is not to demonstrate abstract principles and tips at the level of theory, but to show how theory can be put into practice in real-world examples, helping you get the most from your programming knowledge to build solid games that don't just work but work optimally. To get the most out of this book, read each chapter in sequence, from start to finish, and when reading, use a general and abstract mindset. That is, see each chapter as being simply a particular example and demonstration of more general principles that persist across time and spaces; ones that you can remove from the specific context in which I've used them and redeploy elsewhere to serve your needs. In short, see the knowledge here not just as related to the specific examples and case studies I've chosen, but as being highly relevant for your own projects. So, let's get started.

What this book covers

Chapter 1, Unity C# Refresher, summarizes in very brief terms the basics of C# and scripting in Unity. It's not intended as a complete or comprehensive guide to the basics. Rather, it's intended as a refresher course for those who've previously studied the basics, but perhaps haven't scripted for a while and who'd appreciate a quick recap before getting started with the later chapters. If you're comfortable with the basics of scripting (such as classes, inheritance, properties, and polymorphism), then you can probably skip this chapter.

Chapter 2, Debugging, explores debugging in depth. Being able to write solid and effective code depends partially on your ability to find and successfully fix errors as and when they occur. This makes debugging a critical skill. This chapter will not only look at the basics, but will go deeper into debugging through the MonoDevelop interface, as well as establish a useful error-logging system.

Chapter 3, Singletons, Statics, GameObjects, and the World, explores a wide range of features for accessing, changing, and managing game objects. Specifically, we'll see the singleton design pattern for building global and persistent objects, as well as many other techniques for searching, listing, sorting, and arranging objects. Scripting in Unity relies on manipulating objects in a unified game world, or coordinate space to achieve believable results.

Chapter 4, Event-driven Programming, considers event-driven programming as an important route to re-conceiving the architecture of your game for optimization. By transferring heavy workloads from update and frequent events into an event-driven system, we'll free up lots of valuable processing time for achieving other tasks.

Chapter 5, Cameras, Rendering, and Scenes, dives deep into seeing how cameras work, not just superficially, but how we can dig into their architecture and customize their rendered output. We'll explore frustum testing, culling issues, line of sight, orthographic projection, depth and layers, postprocess effects, and more.

Chapter 6, Working with Mono, surveys the vast Mono library and some of its most useful classes, from dictionaries, lists, and stacks, to other features and concepts, such as strings, regular expressions and Linq. By the end of this chapter, you'll be better positioned to work with large quantities of data quickly and effectively.

Chapter 7, Artificial Intelligence, manages to apply pretty much everything covered previously in one single example project that considers Artificial Intelligence: creating a clever enemy that performs a range of behaviors, from wandering, chasing, patrolling, attacking, fleeing and searching for health-power ups. In creating this character, we'll cover line-of-sight issues, proximity detection, and pathfinding.

Chapter 8, Customizing the Unity Editor, focuses on the Unity Editor, which is feature filled in many respects, but sometimes you need or want it to do more. This chapter examines how to create editor classes for customizing the editor itself, to behave differently and work better. We'll create customized inspector properties, and even a fully functional localization system for switching your game seamlessly across multiple languages.

Chapter 9, Working with Textures, Models, and 2D, explores many things you can do with 2D elements, such as sprites, textures, and GUI elements. Even for 3D games, 2D elements play an important role, and here we'll look at a range of 2D problems and also explore effective and powerful solutions.

Chapter 10, Source Control and Other Tips, closes the book on a general note. It considers a wide range of miscellaneous tips and tricks (useful concepts and applications) that don't fit into any specific category but are critically important when taken as a whole. We'll see good coding practices, tips for writing clear code, data serialization, source and version control integration, and more.

What you need for this book

This book is a Unity-focused title, which means you only need a copy of Unity. Unity comes with everything you need to follow along with the book, including a code editor. Unity can be downloaded from <http://unity3d.com/>. Unity is a single application that supports two main licenses, free and pro. The free license restricts access to certain features, but nonetheless still gives you access to a powerful feature set. In general, most chapters and examples in this book are compliant with the free version, meaning that you can usually follow along with the free version. Some chapters and examples will, however, require the professional version.

Who this book is for

This is an advanced book intended for students, educators, and professionals familiar with Unity basics as well as the basics of scripting. Whether you've been using Unity for a short time, or are an experienced user, this book has something important and valuable to offer to help you improve your game development workflow.

Conventions

In this book, you will find a number of text styles 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: "Once created, a new script file will be generated inside the `Project` folder with a `.cs` file extension."

A block of code is set as follows:


```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyNewScript : MonoBehaviour
05 {
```


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

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

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

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "One way is to go to **Assets | Create | C# Script** from the application menu."

 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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in 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 at 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 from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/06550T_ColoredImages.pdf.

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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Unity C# Refresher

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

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

Why C#?

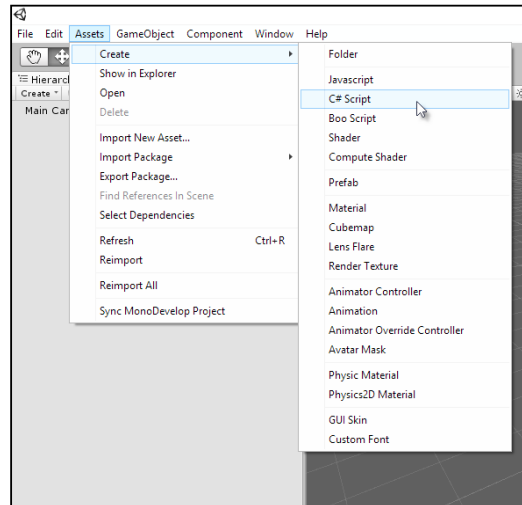
When it comes to Unity scripting, an early question when making a new game is which language to choose, because Unity offers a choice. The official choices are C# or JavaScript. However, there's a debate about whether JavaScript should more properly be named "JavaScript" or "UnityScript" due to the Unity-specific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

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

Creating script files

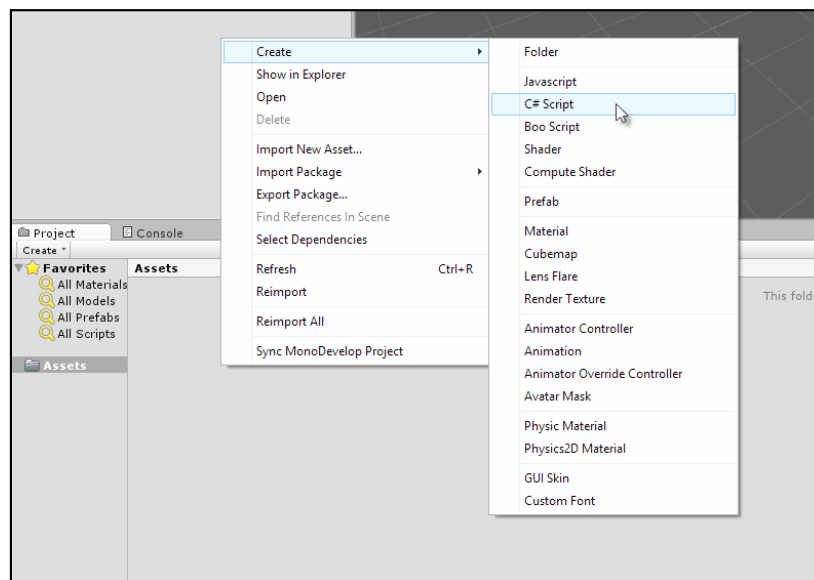
If you need to define a logic or behavior for your game, then you'll need to write a script. Scripting in Unity begins by creating a new script file, which is a standard text file added to the project. This file defines a program that lists all the instructions for Unity to follow. As mentioned, the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

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



Creating a script file via the application menu

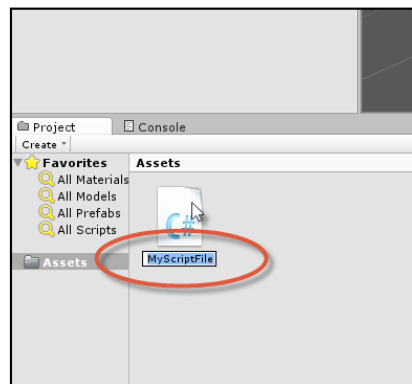
Another way is to right-click on the empty space anywhere within the **Project** panel and choose the **C# Script** option in the **Create** menu from the context menu, as shown in the following screenshot. This creates the asset in the currently open folder.



Creating a script file via the Project panel context menu

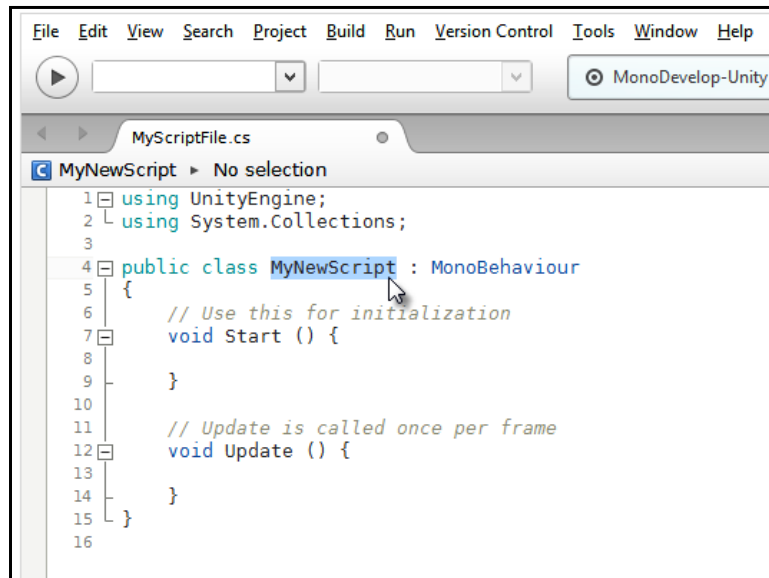
Once created, a new script file will be generated inside the `Project` folder with a `.cs` file extension (representing C Sharp). The filename is especially important and has serious implications on the validity of your script files because Unity uses the filename to determine the name of a C# class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

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



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

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



Renaming classes to match the renamed script files

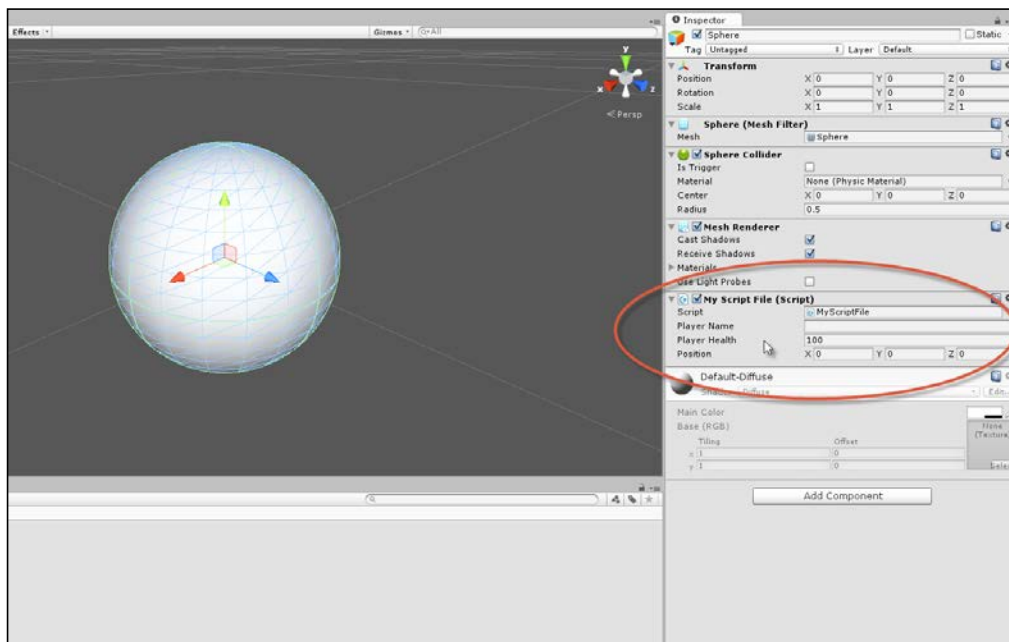
Compiling code

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



Instantiating scripts

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



Attaching scripts onto game objects as components

Variables are considered in more depth in the next section.



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

Variables

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

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

Variable data types

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



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

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

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

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

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

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

Variable scope

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

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

Conditional statements

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

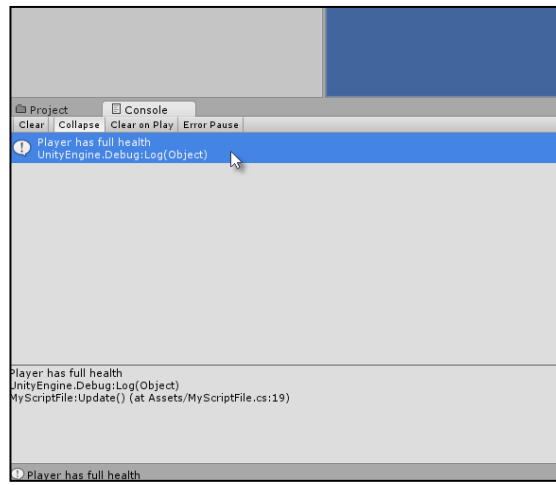
The if statement

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

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

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

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



The Unity Console is useful for printing and viewing debug messages

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

Unity Console



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

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

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



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

The if-else statement

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



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

The switch statement

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

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

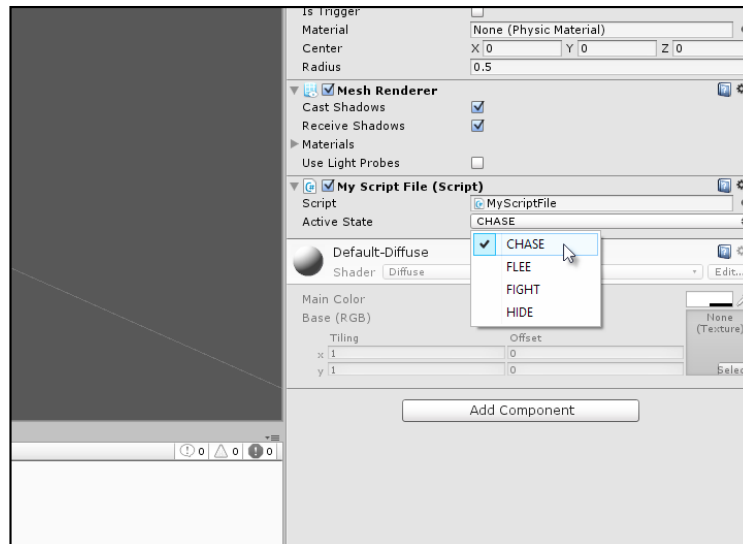
```
09    //The current state of enemy
10    public EnemyState ActiveState = EnemyState.CHASE;
11
12    // Use this for initialization
13    void Start () {
14    }
15
16    // Update is called once per frame
17    void Update ()
18    {
19        //Check the ActiveState variable
20        switch(ActiveState)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

Enumerations



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

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



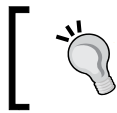
Enumerations offer you drop-down options for your variables from the Object Inspector

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

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

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

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



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

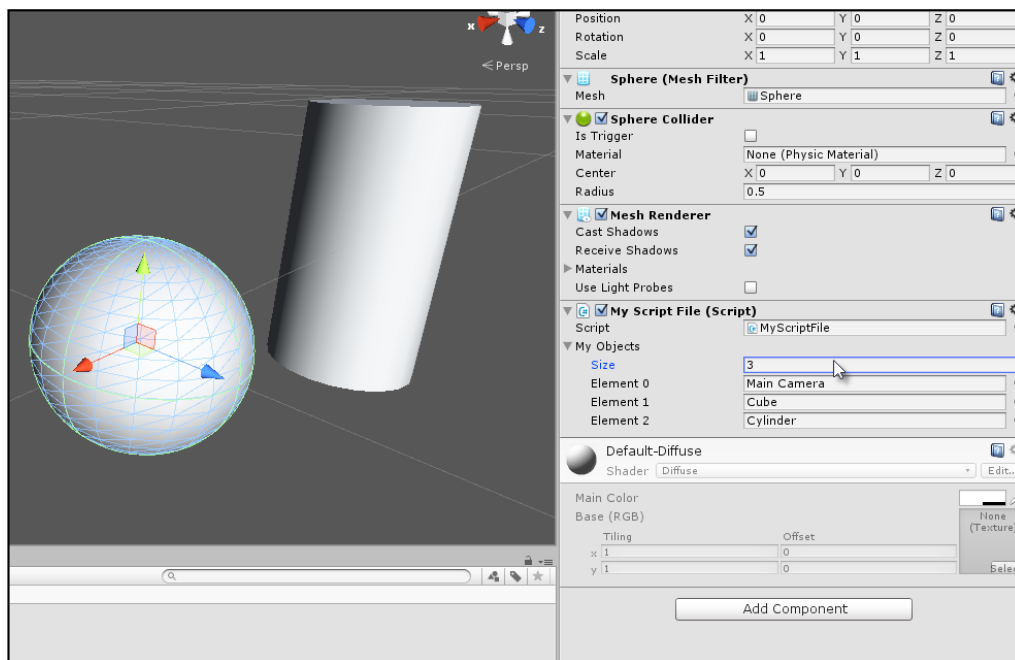
Arrays

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

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

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

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



Building arrays from the Unity Object Inspector

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

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

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

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