

# **Beginning C# Programming with Unity**

## **Visual Studio Edition**

**A.T. Chamillard**



Copyright © 2017 A.T. Chamillard. All rights reserved. No part of this book may be reproduced or retransmitted in any form or by any means without the written permission of the author or publisher.

Published by Burning Teddy, Colorado Springs, Colorado, U.S.A.

ISBN-13: 978-0-9985711-0-2

ISBN-10: 0-998-57110-5

This book is dedicated to my wife Chris. As always, she puts up with so many of my crazy ideas, even when I say “I think it would be fun to write another book ...” I love you, Chris.

# Contents

<b>Chapter 1. Introduction.....</b>	<b>1</b>
1.1. Hardware Organization.....	1
1.2. Computer Software .....	3
1.3. Writing and Running a C# Program .....	4
<b>Chapter 2. Starting to Code .....</b>	<b>7</b>
2.1. What Does a C# Program Look Like?.....	7
2.2. Using Other Namespaces and Classes.....	8
2.3. Namespace.....	9
2.4. Comments .....	9
2.5. Class.....	10
2.6. Identifiers.....	10
2.7. The Main Method .....	11
2.8. Variables and Constants.....	11
2.9. Console Output .....	12
2.10. A Word About Curly Braces .....	14
2.11. A Matter of Style .....	16
2.12. Solving Problems One Step at a Time.....	16
2.13. Sequence Control Structure .....	21
2.14. Testing Sequence Control Structures.....	22
2.15. Putting It All Together.....	22
2.16. Adding a Unity Script .....	24
<b>Chapter 3. Data Types, Variables, and Constants.....</b>	<b>30</b>
3.1. Variables .....	30
3.2. Value Types, Variables, and Constants .....	32
3.3. Integers.....	33
3.4. Floating Point Numbers .....	34
3.5. Decimals .....	35
3.6. Characters .....	36
3.7. Booleans.....	37
3.8. Operations.....	37
3.9. Choosing Between Variables and Constants .....	38
3.10. Giving Variables a Value.....	39
3.11. Type Conversions .....	40
3.12. What About Reference Types? .....	42
3.13. Putting It All Together in a Console App .....	42

3.14. Putting It All Together in a Unity Script.....	50
3.15. Common Mistakes.....	56
<b>Chapter 4. Classes and Objects.....</b>	<b>57</b>
4.1. Introduction to Classes and Objects .....	57
4.2. Your First C# Program Revisited.....	61
4.3. Calling Methods .....	66
4.4. Reference Types in Memory .....	68
4.5. The Circle Problem Revisited .....	70
4.6. The Circle Problem in Unity Revisited .....	76
4.7. Putting It All Together in a Console App.....	81
4.8. Putting It All Together in a Unity Script.....	87
4.9. Common Mistakes.....	93
<b>Chapter 5. Strings.....</b>	<b>95</b>
5.1. The String Class .....	95
5.2. The StringBuilder Class .....	97
5.3. Getting Input .....	98
5.4. Putting It All Together .....	101
5.5. Common Mistakes.....	109
<b>Chapter 6. Unity 2D Basics.....</b>	<b>110</b>
6.1. The Worst Game EVER.....	110
6.2. A Better Not A Game (NAG).....	111
6.3. Prefabs.....	115
6.4. Unity Circles Revisited .....	118
6.5. Putting It All Together .....	120
<b>Chapter 7. Selection.....</b>	<b>130</b>
7.1. Selection Control Structure .....	130
7.2. Testing Selection Control Structures.....	132
7.3. If Statements.....	133
7.4. Switch Statements .....	138
7.5. Timers.....	140
7.6. Putting It All Together .....	150
7.7. Common Mistakes.....	158
<b>Chapter 8. Unity Mice, Keyboards, and Gamepads.....</b>	<b>160</b>
8.1. The Input Manager .....	160
8.2. Mouse Input.....	162
8.3. Keyboard Input.....	169
8.4. Gamepad Input .....	170

8.5. Putting It All Together.....	172
<b>Chapter 9. Arrays and Collection Classes.....</b>	<b>182</b>
9.1. Declaring and Creating Array Objects.....	182
9.2. Accessing Elements of the Array.....	184
9.3. Arrays of Objects.....	185
9.4. Refactoring the Card Class .....	186
9.5. Multi-Dimensional Arrays.....	190
9.6. Collection Classes.....	190
9.7. Putting It All Together.....	192
9.8. Common Mistakes .....	210
<b>Chapter 10. Iteration: For and Foreach Loops.....</b>	<b>211</b>
10.1. Iteration Control Structure .....	211
10.2. For Loops.....	212
10.3. Foreach Loops.....	214
10.4. Choosing Between For and Foreach Loops .....	215
10.5. Nested Loops .....	215
10.6. Arrays and Loops - "Walking the Array" .....	217
10.7. Blowing Up Teddies, Take 1 .....	218
10.8. Blowing Up Teddies, Take 2 .....	223
10.9. Putting It All Together.....	226
10.10. Common Mistakes .....	235
<b>Chapter 11. Iteration: While Loops .....</b>	<b>237</b>
11.1. Iteration Control Structure Revisited.....	237
11.2. While Loops.....	238
11.3. Testing While Loops.....	240
11.4. Do-While Loops .....	241
11.5. Spawning Into a Collision-Free Location .....	243
11.6. Putting It All Together.....	246
11.7. Common Mistakes .....	253
<b>Chapter 12. Class Design and Implementation.....</b>	<b>255</b>
12.1. High-level Class Design .....	255
12.2. Implementing Classes in C#: Fields .....	258
12.3. Implementing Classes in C#: Properties .....	261
12.4. Implementing Classes in C#: Constructors and Other Methods.....	264
12.5. Putting It All Together.....	272
12.6. Common Mistakes .....	284

<b>Chapter 13. More Class Design and Methods.....</b>	<b>285</b>
13.1. A Brief Review.....	285
13.2. Deciding Which Methods to Use .....	285
13.3. Figuring Out Information Flow.....	286
13.4. Creating the Method Header .....	286
13.5. The Method Body.....	287
13.6. Parameters and How They Work .....	289
13.7. Passing Objects as Parameters .....	290
13.8. Putting It All Together .....	291
13.9. Common Mistakes.....	301
<b>Chapter 14. Unity Text IO.....</b>	<b>302</b>
14.1. A Big Idea .....	302
14.2. Text Output .....	302
14.3. Text Input .....	306
14.4. Putting It All Together: The Circle Problem in Unity Revisited.....	309
<b>Chapter 15. Unity Audio.....</b>	<b>315</b>
15.1. Adding Audio Content .....	315
15.2. Audio Sources and the Audio Listener.....	316
15.3. Playing Audio Clips from Scripts .....	317
<b>Chapter 16. Inheritance and Polymorphism.....</b>	<b>321</b>
16.1. Inheritance Concepts .....	321
16.2. Inheritance in C# .....	324
16.3. Polymorphism .....	328
16.4. The Standard Bank Account Example .....	330
16.5. Inheritance and Fields .....	334
16.6. Inheritance, Properties, and Methods .....	335
16.7. The Object Class .....	338
16.8. Polymorphism Revisited .....	339
16.9. The MonoBehaviour Class.....	341
16.10. Putting It All Together .....	342
16.11. Common Mistakes.....	357
<b>Chapter 17. Delegates and Event Handling .....</b>	<b>358</b>
17.1. What Are Delegates?.....	358
17.2. Customizing Behavior With Delegates .....	359
17.3. Events and Event Handling .....	363
17.4. Refactoring the Teddy Bear Destruction Game .....	369
17.5. UnityEvent and UnityAction.....	372

17.6. Refactoring the Teddy Bear Destruction Game Again.....	372
17.7. Menu Buttons.....	374
17.8. Adding a Menu System to the Fish Game .....	376
<b>Chapter 18. Exceptions, Equality, and Hash Codes .....</b>	<b>378</b>
18.1. Exceptions.....	378
18.2. Equality.....	386
18.3. Hash Codes .....	388
<b>Chapter 19. File IO .....</b>	<b>390</b>
19.1. Streams.....	390
19.2. Text Files .....	391
19.3. Files of Objects .....	395
19.4. PlayerPrefs Class .....	397
19.5. Game Configuration Storage .....	411
<b>Chapter 20. Putting It All Together .....</b>	<b>418</b>
20.1. Understand the Problem.....	418
20.2. Design a Solution (Menus).....	418
20.3. Write Test Cases (Menus).....	419
20.4. Write the Code (Menus) .....	421
20.5. Test the Code (Menus).....	441
20.6. Design a Solution (Basic Gameplay: Stupid Teddies).....	441
20.7. Write Test Cases (Basic Gameplay: Stupid Teddies).....	444
20.8. Write the Code and Test the Code (Basic Gameplay: Stupid Teddies).....	446
20.9. Design a Solution (Full Gameplay: AI Teddies) .....	467
20.10. Write Test Cases (Full Gameplay: AI Teddies).....	468
20.11. Write the Code and Test the Code (Full Gameplay: AI Teddies).....	469
20.12. Design a Solution (Sound).....	472
20.13. Write Test Cases (Sound) .....	472
20.14. Write the Code and Test the Code (Sound) .....	473
20.15. Design a Solution (XML Configuration Data) .....	478
20.16. Write Test Cases (XML Configuration Data).....	478
20.17. Write the Code and Test the Code (XML Configuration Data).....	479
20.18. Conclusion .....	481
<b>Appendix: Setting Up Your Development Environment.....</b>	<b>482</b>
A.1. Visual Studio Community 2019.....	482
A.2. Unity .....	487

## Preface

Welcome to the wonderful world of programming! In this book you'll learn the basics of programming using the C# programming language. While we admit we love to code (another word for program) just about anything, developing games is one of the coolest things of all. Most of the examples in this book are related to game development using Unity. So you'll learn how to program properly and you'll learn how to write C# scripts in Unity.

You'll notice we said "learn how to program properly" above, not "whack together games that seem to work." This is a book that focuses on the correct way to write game software (and software in general), so there's lots of discussion about our motivations for the particular design and coding decisions we make throughout the book. If you really just want to learn all the nuts and bolts of Unity, there are numerous books available for that, and you should buy one of those instead. If, however, you aspire to be a professional game programmer – whether as an indie game developer or in a large game company – then this book will give you a solid foundation for starting on that path.

It's important to note that the Unity focus in this book is on the actual C# scripting you do to build Unity games. The book doesn't cover the full gamut of how to build 2D and 3D Unity games from scratch (though we do build a complete game in Chapter 20); instead, we use just enough "Unity stuff" to build interesting pieces of game functionality so we can concentrate on learning C#. Don't worry, there are plenty of other books and resources for learning the rest of Unity!

### Who This Book Is For

This book assumes that you've never programmed before, so all the material starts at the most basic level. That means that anyone should be able to pick up the book and work their way through it without any prior knowledge. That's the good news.

The bad news is that programming is hard work, especially at first. If you truly want to learn how to program, you'll need to write programs yourself and struggle through some rough spots before some topics really click for you. Just as you can't learn how to ride a bicycle by reading about it – you have to actually do it, probably with some spills along the way – you can't learn to program just by reading about it. If you were hoping to read a book to learn how to program without doing any programming yourself, it's not going to happen. Coding games is incredibly fun, but it takes time and effort. This book will give you a good start, but you should only buy it if you're willing to work at it. If not, it's just a waste of your money that would be better spent on a new game <grin>.

If you're already an experienced programmer, you'd probably be better off buying a book that focuses on building games from scratch in Unity because you should already know the most important foundational programming concepts. Getting more sales is always great, of course, but we don't want you to waste your time and money.

### Code Details

There are lots of code examples sprinkled throughout the book. For a variety of reasons, you should download the code to accompany the book examples from [www.burningteddy.com](http://www.burningteddy.com). In some cases, we don't provide all the code in the body of the book text, so looking at the downloaded code is the only

way to see it. In addition, the code that's in the text may not look exactly the way it does in Visual Studio because of where the line breaks fall in the text. Finally, if you download the code you can actually run it and play around with it! Using the code in the text to get the general idea, then looking at and running the downloaded code to explore all the details, may be an effective technique for you to use.

## Typographical Conventions

### Menu Options

We indicate menu options using >, as in File > Open

### Constant Width

Indicates C# code. The code may also be colored as appropriate.

### *Italic*

Used to highlight a new term we haven't used before.

### Constant Width Italic

Indicates C# code you need to replace with an appropriate value (e.g., *variableName*).

## About the Author

The author retired from the U.S. Air Force after over 20 years, including 6 years spent teaching Computer Science at the U.S. Air Force Academy. He then joined the Computer Science faculty at the University of Colorado at Colorado Springs (UCCS) and immediately started building and teaching game development courses. He's now the Program Director for the Bachelor of Innovation™ in Game Design and Development at UCCS.

The author also started Peak Game Studios, an indie game development company, with his two sons in 2007. The company developed numerous games on work-for-hire contracts, released a commercial Windows game in 2010, and got another game almost all the way to completion. We've since shut that company down, but the author now has his own company (Burning Teddy) he uses to publish games and books. Except for one small Flash development, the company used C# for all development.

Because the author loves teaching as much as he loves game development, he offered the first Massive Open Online Course (MOOC) in UCCS history on Coursera in Fall 2013. He currently offers a variety of programming and game development courses on Coursera (<https://www.coursera.org/>) and Udemy (<https://www.udemy.com/>).

This is the author's fourth book; the previous books were Beginning C# Programming with MonoGame, Beginning C# Programming with XNA Game Studio (2 editions), and Introductory Problem Solving Using Ada 95 (3 editions).

You'll notice throughout the book that the author refers to himself as "we" instead of "I". That's not because he thinks he's royalty (as in "We are not amused")! It's just a convention that most authors use in their writing.

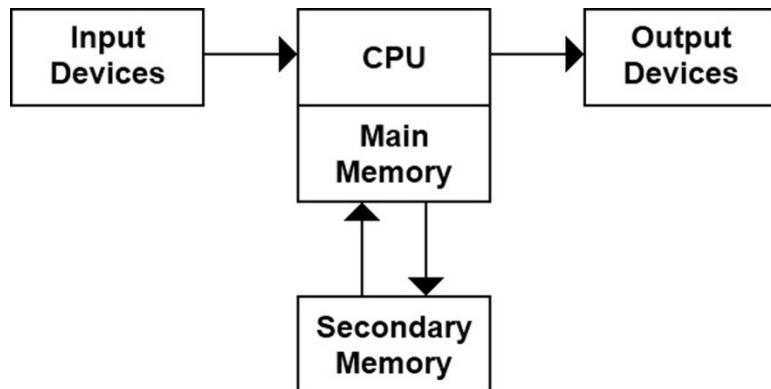
# Chapter 1. Introduction

C# is a programming language that's been around since 2001. It's one of the programming languages included in Microsoft .NET<sup>1</sup>, and it can be used to develop all kinds of programs, including games. Why did we pick C# as the language to teach in this book? For two important reasons. First, it's a robust, modern language that's relatively easy to learn as a first programming language. Second, and more importantly, C# is the preferred language to use when we write scripts in the Unity game engine. Because scripts are the things that actually implement the gameplay in our games, it's pretty important that you know how to write them as you develop your own games!

If we're going to write C# programs to run on our computer (or our professor's computer), it will sure help if we understand the basics of what a typical computer looks like. In this chapter, we'll look at basic computer organization, talk about how software works on that organization, and even get to writing and running our first C# program.

## 1.1. Hardware Organization

Before we start talking about how the hardware in a computer is organized, what is hardware in the first place? Well, the easiest way to think about it is that hardware will hurt if you drop it on your foot! In other words, hardware consists of the parts of the computer that we can see and touch. Most computers, and certainly almost all PCs, are composed of 4 different kinds of hardware – the Central Processing Unit (CPU), memory (main and secondary), input devices, and output devices<sup>2</sup>.



**Figure 1.1. Basic Hardware Organization**

The *Central Processing Unit*, commonly called the *CPU*, is really the “brains” of the computer. Computers are actually much dumber than you might think – the only thing your computer can do is execute instructions, one after another, until you turn it off! The trick to making computers do useful things is in the instructions we tell them to execute; those instructions are in the form of computer software, which we'll get to in the following section. In any case, the CPU is the part of the computer

<sup>1</sup> .NET was originally called the .NET Framework, then Microsoft added .NET Core, and now it's just called .NET

<sup>2</sup> There are regularly more cards and/or chips in a computer, particularly a gaming rig. For example, a graphics card has a Graphics Processing Unit (GPU) that can actually be used to do some calculations, and we regularly find a separate sound card as well. The hardware organization shown in Figure 1.1. is a simplification that gets at the core ideas of how software runs on a computer and how we interact with the software.

## 2 Chapter 1

that executes those instructions, including deciding what to do next, completing mathematical computations, and so on.

But where does the CPU get the instructions to execute? From the *main memory*. There are really two kinds of memory in the computer – main memory and secondary memory – but the CPU can only get its instructions from main memory. Main memory comes in two forms: Random Access Memory (RAM) and Read Only Memory (ROM). Read Only Memory is useful for storing some permanent information, such as instructions about what to do when the computer is turned on, but it can't be changed as the computer is running. Random Access Memory can be changed, and that's what we'll use when we write our programs. One other comment about RAM – it's called volatile memory, which means that whatever is stored in RAM will be erased when you turn off your computer. You may have also heard about cache memory, which is really just a special, extra-fast type of main memory that actually resides on the CPU chip rather than on separate RAM chips.

Wait a minute, you say, I can run lots of programs when I turn my computer on; they don't just disappear! That's because they're also stored in *secondary memory*, which is non-volatile. The most common form of secondary memory is your hard disk, but solid state drives fall into this category as well. The program instructions to be executed are stored in your secondary memory until you decide to run that program. At that point, the instructions are loaded into main memory so the CPU can get and execute them.

Okay, we know that memory can store instructions, but what else can it store? Just about anything! Ultimately, everything in the computer is stored in *binary* – that is, ones and zeros – but as long as we can encode whatever we're trying to store as ones and zeros, we can store it in memory. That means we can store numbers, letters, pictures, sound files, and all kinds of stuff in memory. We'll talk more about how we use memory as we get further into our programming.

So far, we've talked about the parts of the computer that execute instructions (the CPU) and that hold those instructions and other data (the memory). At this point, we have a computer that can run programs all day (and even all night) long. There's one problem, though – we can't provide any input to those programs, and we can't see the results either. To do that, we need to use input and output devices. Can you imagine how boring a game would be if we couldn't provide any input to interact with the game world and didn't get any output (like a graphical display) to find out what's happening in the game world? That would be horrible!

As you might expect, *input devices* are hardware devices we use to provide input to the computer. Some of the more obvious input devices are the keyboard and the mouse, but there are lots of other input devices as well: game controllers, touch pads, track balls, microphones, scanners, and so on. Basically, anything that we can use to provide input to the computer is an input device.

*Output devices*, on the other hand, are the devices that provide output to the user of the computer. The most obvious such device is your computer monitor. Any output from the computer can be provided to you through the monitor, so it's clearly an output device. What are some other examples? How about speakers, force-feedback gamepads and joysticks (which also serve as input devices), printers, and so on. To make our computer really useful, we need both input devices and output devices.

One last thing before we move on to software. How are all these hardware components hooked together? Using something called a bus. Think of the bus as a forum, where devices connected to the bus can post information and read the posts to communicate. All the hardware gets connected to the bus, then the

hardware components can “talk” to each other by posting messages on the bus. The key point is that there’s a centralized communication mechanism for all the hardware devices using the bus.

## 1.2. Computer Software

So we’ve talked about the parts of a computer and how they’re hooked together, but let’s face it – the computer is really just an expensive paperweight unless we’ve got software to run on it! Remember, all the computer can do is execute instructions. Computer software is the set of instructions the computer will run.

Now, we said earlier that everything gets stored in the computer as ones and zeros. Does that mean we have to write our software as sequences of ones and zeros? No, thank goodness! All we have to do is be able to convert our programs into instructions that the CPU can understand (these are often called *machine instructions* or *machine code*); we’ll talk about that process soon. In the meantime, you should know that the earliest programmers actually DID write their programs in binary.

Luckily, the art and science of programming has progressed far beyond having to write our programs in binary. In fact, we can now pick from a large number of programming languages; C#, C++, C, Java, Ada, Visual Basic, Ruby, and Python are just a few of these languages. We’ll talk specifically about how C# works in the rest of our discussion, but many of the concepts are the same no matter which language you use.

Before we talk about how we get a program to run on the computer, let’s talk for a minute about programming languages in general. One of the key things you need to remember as you start to program is that programming languages have both *syntax* and *semantics*. When we talk about a language’s syntax, we mean the rules that determine the order of the words we include and the punctuation we use. This is a lot like the grammar rules associated with more “normal” languages (like English). For example, English grammar rules tell us how we can build and punctuate a sentence. Similarly, C# tells us how we can build and punctuate the instructions (called statements) we create in that language.

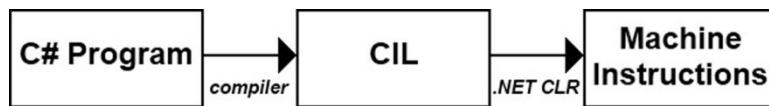
Semantics, on the other hand, relates to the meaning of the sentences (or instructions) we build. For example, the English sentence “The large blue fox jumped over the moon” is grammatically correct, but it doesn’t really make sense. Similarly, we can also create C# statements that are syntactically correct but don’t really make sense. The bottom line is that when we learn a new language, like English or C#, we need to learn both the syntax and the semantics of that language.

Okay, so we learn the syntax and semantics of a programming language, and we’re ready to write a program to run on our computer. How do we do that? Well, the first step is to type our program in using either an *editor* (like Microsoft Notepad) or an *Integrated Development Environment (IDE)* (like Visual Studio Community 2019). Because IDEs are so much more powerful than basic editors, we’ll be using an IDE throughout this book. No matter how we type in our program (which is typically called *source code*), when we’re done with this step we have a program written in our programming language of choice (in our case, C#).

Now we need to somehow convert our program into instructions that the CPU can understand. Programming languages generally fall into two classes for this step: *interpreted languages* and *compiled languages*. In interpreted languages, an *interpreter* converts the program one statement at a time and the CPU executes the instructions as they’re converted. This can be very helpful as you’re developing your program, but it can also be fairly slow because you’re doing the conversion while you run your program.

In compiled languages, a *compiler* converts the entire program to the machine instructions; you can then have the CPU run those instructions. Generally speaking, compiled programs run faster than interpreted programs because the translation to machine instructions has already been done. We should note that both interpreters and compilers check the syntax of the program provided to them to make sure they're grammatically correct while they're translating them.

So is C# a compiled language or an interpreted language? Well, it's actually both! That's one of the characteristics that makes C# such a powerful programming language. When we compile our C# programs, the compiler doesn't actually generate machine instructions; instead, it generates something called *Common Intermediate Language (CIL)*<sup>3</sup>. The CIL, which you can think of as .NET instructions, isn't specific to any particular CPU, which makes the CIL portable to any machine. When it's time to actually run the program, the CIL is interpreted into machine instructions by the .NET Common Language Runtime (CLR). This hybrid approach gives C# great portability, and program execution time doesn't suffer much at all because a lot of the translation effort occurs when the compiler generates the CIL and the CLR does other optimizations as well. Figure 1.2 summarizes how this process works.



**Figure 1.2. C# Compilation and Execution Process**

The bottom line for computer software is that we can write that software using any of a large number of programming languages. Before the computer can run it, though, we need to translate the program from the language we're using into the machine instructions the CPU can understand. We can do that using an interpreter, a compiler, or a hybrid approach like C# uses. Once we have the machine instructions, we can actually have the computer run the software to do something useful. Remember – computer hardware is just an expensive paperweight without software to run!

### 1.3. Writing and Running a C# Program

At this point, we have a basic understanding of how computer hardware is organized and how computer software can be written to run on that hardware. Let's actually go through the process of writing, compiling, and executing (running) a C# program so you can actually see this really work.

There are numerous C# IDEs available for a variety of prices, but Visual Studio is an excellent IDE that you can use for free. It also easily integrates with Unity. In fact, now would be a great time to set up your entire development environment; you should go to the Appendix and follow the instructions there. Don't worry, we'll wait right here for you.

Let's start by typing in your program. Start up the IDE and click the Create a new project box on the lower right. Select the Console App (.NET Framework) C# button on the upper left, then click the Next button on the lower right. On the next screen, change the Project name from ConsoleApp1 to PrintRainMessage in the text box at the top left of the screen, change the Location to put the project wherever you want it to be saved, then click the Create button at the bottom right. Next, type in the program shown in Figure 1.3; most of the code has already been provided by the IDE, so you only have to type in the lines of code that start with `Console.WriteLine` in the figure. We've added some

---

<sup>3</sup> Common is included in the name because all the .NET languages (VB, C#, etc.) are compiled to CIL.

comments as well, but you don't have to type those in, just type in the four lines of code. We'll talk a lot more about this program, and C# programs in general, in the next chapter, so don't worry that a lot of it may seem strange to you at this point.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrintRainMessage
{
    /// <summary>
    /// Prints a rain message
    /// </summary>
    class Program
    {
        /// <summary>
        /// Prints a rain message
        /// </summary>
        /// <param name="args">command-line args</param>
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, world");
            Console.WriteLine("Chinese Democracy is done and it's November");
            Console.WriteLine("Is it raining?");
            Console.WriteLine();
        }
    }
}
```

**Figure 1.3. Program.cs**

Once you've typed the program in, you need to save the project. Select File > Save All from the menu bar at the top of the IDE or use Ctrl+S. The source code you just typed in is saved in a file named Program.cs. The cs extension tells the compiler, and anyone else looking at the file name, that the file contains C# source code<sup>4</sup>. Also, capitalization matters in C# as well, so make sure you pay attention to that. When you save the above program, the IDE should save everything fine.

Okay, we now have our C# project saved. Our next step is to compile the program into CIL. To do that, simply press F6 (or whatever your keyboard shortcut is for building the solution) or select Build > Build Solution from the top menu bar and wait for the "Build succeeded" message in the bar at the bottom of the IDE. If you end up with errors instead, the Error List window contains an error list. You can click in the Error List window on a line containing an error and you'll go to that line in the Code window. If this happens, go back and figure out what you typed in wrong. Computers are great because they do exactly what we tell them to do, but they're also merciless because they do exactly what we tell them to do! Perhaps in programming more than almost any other endeavor, the details are incredibly important. The code you type in from Figure 1.3 above needs to match exactly what's there, capitalization and all; pretty close won't cut it in a computer program.

---

<sup>4</sup> The programs that we write in C# and other programming languages are called source code because they're the initial source of the machine instructions that are ultimately generated.

## 6 Chapter 1

Once you have a program that compiles successfully (Build succeeded), you can run it by pressing Ctrl+F5. You should get the window shown in Figure 1.4 below (see the Appendix to find out how to change the background and text color in the Command Prompt window if you want to).



**Figure 1.4. Program Output**

Exit the IDE. How do you open and run your code again? You can certainly just open up Visual Studio again and scroll through the list of recent solutions to get to it, but that won't work if you've moved the folder your solution is saved in to a different location., but that won't work if you've moved the folder your solution is saved in to a different location. In that case, navigate to the solution folder. Your solution file has a .sln file extension, but in case you're not showing file extensions in Windows Explorer it's the file marked "Microsoft Visual St...", "Visual Studio Solu...", or something similar as the Type. Double-click that file. This will open up your solution in Visual Studio.

That's all there is to it! We type in our program using an IDE, compile it into CIL using the IDE's built-in compiler, then run it. We'll talk a lot more throughout the rest of the book about how we write useful C# programs to solve interesting problems, but at least now we know the steps we go through to compile and run them.

## Chapter 2. Starting to Code

In the previous chapter, you typed in, compiled, and ran your first C# console app. We had you do that because we wanted you to understand how those steps work before we start using C# to solve more problems. Now that you've done that, though, it's time to take a closer look at C# programs in general.

### 2.1. What Does a C# Program Look Like?

This is actually a trickier question than you might think! Because C# is an object-oriented language, we're going to end up writing lots of things called *classes*. We use classes as building blocks for our problem solutions, so they're a critical part of our C# programs. We'll talk about classes lots more throughout the book, including later in this chapter, so let's put off that discussion for a while longer.

Although our programs will use lots of classes, we also need a .cs file we can actually run. Lots of people call this class the *application class* because it's the class that essentially runs the program (another word for application). If we think of our program as a software application or app (which it is), the application class terminology makes sense. The Program.cs class you typed in followed the "typical" format for such an application class; the syntax for a C# application class is provided below. We'll discuss all the parts in further detail, so don't worry if you don't understand them all right away.

---

#### SYNTAX: C# Application Class

```
using directives for namespaces

namespace NamespaceName
{
    class documentation comment
    class Program
    {
        method documentation comment
        static void Main(string[] args)
        {
            constant declarations

            variable declarations

            executable statements
        }
    }
}
```

---

Since we'll be providing lots of "syntax descriptions" in the coming chapters, we should discuss the notation a bit. When we list items in italics (e.g., *class documentation comment*), that means you, the programmer, decide what goes there. When we list words without italics (such as `namespace`, `class`, `static`, and `void`), those words need to appear EXACTLY as written. That's because these are special words in C# (they're called *keywords*), so you need to use them just as they appear in the syntax descriptions.

Okay, let's discuss each of the parts of a C# application class in greater detail.

## 2.2. Using Other Namespaces and Classes

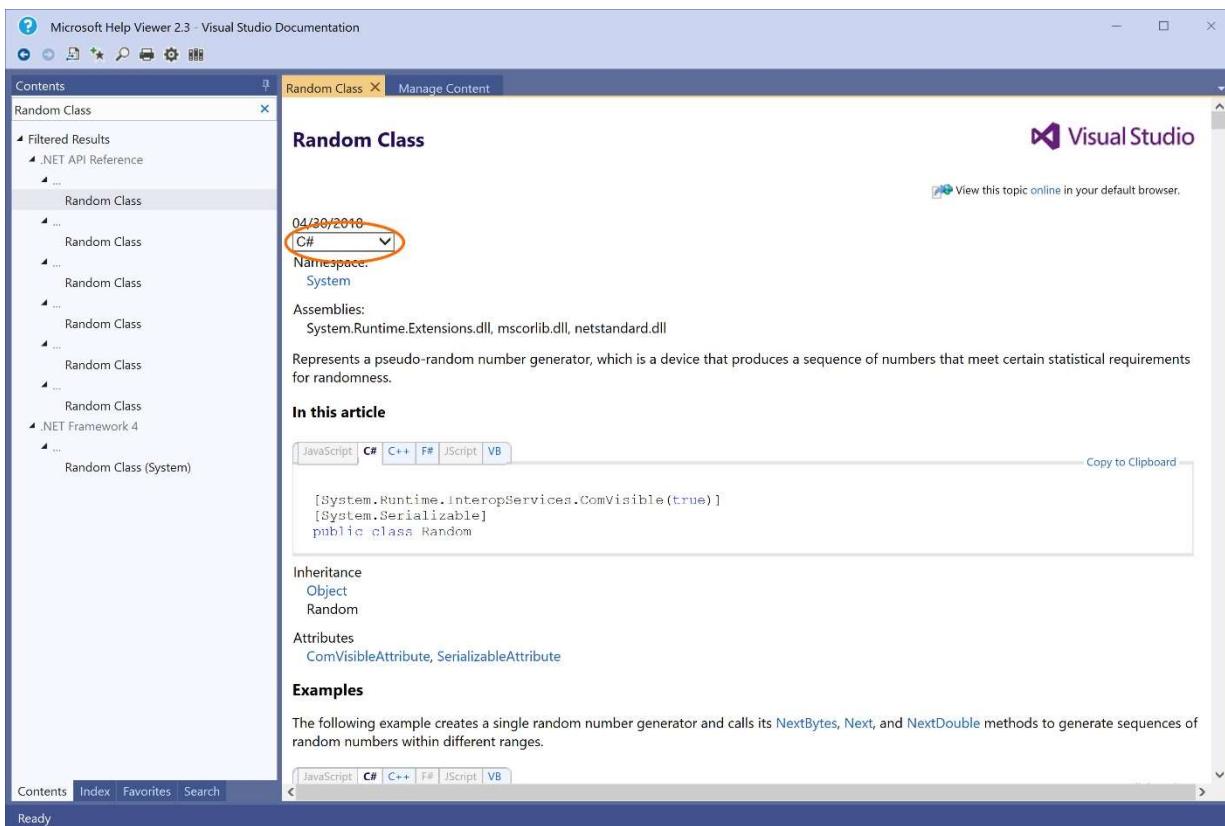
The first thing we see in the application class is the place where we say we'll be `using` other namespaces and classes. Namespaces and classes are collections of useful C# code that someone else has already written, tested, and debugged (so that you don't have to!). In fact, all C# development environments come with a set of namespaces and classes that you'll find very useful. One example that we'll use a little later is a class that will generate random numbers for us. That class (called, surprisingly enough, `Random`) is found in the `System` namespace, so when we need it, we'll include the following line in our program:

```
using System;
```

This *using directive* tells the compiler that you want access to the classes in the `System` namespace. We'll introduce you to other namespaces provided in C# as we need them. Of course, Unity gives us even more namespaces to use as we develop games, and we'll learn about those namespaces as we need them as well.

So there are lots of handy namespaces and classes available to us, but how do we find out more about them? By looking at the documentation, of course! You'll probably find that you use the documentation a lot as you learn to program in C# – and even when you're an experienced C# programmer – so let's walk through the process now.

The typical way you'll use the C# documentation is by selecting Help > View Help in Visual Studio to look for information about a specific class or namespace, but remember you can also go to <https://msdn.microsoft.com/en-us/library/> if you prefer. Let's try to find out more about the `Random` class. Selecting Help > View Help in Visual Studio, type Random Class into the Search box, and press <Enter>. After the search results are returned, clicking on the first "Random Class" result in the results pane brings you to the documentation for the `Random` class. You should always make sure you've selected C# in the dropdown – inside the orange circle in Figure 2.1 – to get the C# documentation. The documentation page you get should look like Figure 2.1 (without the orange circle, of course).



**Figure 2.1. Random Class Documentation Window**

You can then explore the functionality that's provided by the class by clicking on the links in the documentation. We'll look at using documentation in more detail later in this chapter.

## 2.3. Namespace

The next item in the syntax description is the line that tells what namespace this class belongs to. As discussed in the previous section, namespaces provide a way to group related chunks of C# code together. Most of the code you write as you work (slog?) through this book will be contained in a single namespace, and the IDE will automatically generate this part of your code based on the name you decided to use when you created the project. You generally won't have to touch this code at all.

## 2.4. Comments

The next item in the syntax description is the class documentation comment. Comments that start with three slashes, `///`, are called documentation comments; XML tags are used within the documentation comments.

Although comments aren't actually required by the language, it's always a good idea to provide documentation within your programs. The class documentation comment should contain a description of the class. Here's the class documentation comment from our program:

```
/// <summary>
/// A class to print a rain message
/// </summary>
```

The class documentation comment should start with the line `/// <summary>`. and end with the line `/// </summary>`. The good news is that when you type three slashes on the line above the line that starts with `class`, the IDE automatically generates and pre-populates the documentation comment; you just fill in the actual details. You should also notice that the IDE colors the text you enter for comments gray and green.

Now, you might be wondering why we call these documentation comments. They actually do more than just making it easier to understand the source code. There are tools available (Sandcastle is one of them) that can be used to process all your source code and generate navigable documentation. Although we won't actually teach you how to use Sandcastle in this book<sup>5</sup>, when you start developing games in larger teams of programmers (larger meaning more than just you!) you'll want to generate the documentation so that other programmers using your code don't have to actually read your source code to figure out how to use it. So you should definitely start using documentation comments now.

There are also two other kinds of comments we include in the programs we write. For each method in our solution (like the `Main` method), we write a documentation comment similar to the class documentation comment. Method documentation comments contain a description of the method and information about parameters and return values for the method. Don't worry; we'll cover all these ideas in great detail as we need them.

The last kind of comment we include in our program is called a *line comment*. Line comments start with a double slash, `//`, followed by whatever descriptive text you choose to add after the double slash. Every few lines of C# code you write should have a comment above them explaining what those lines are supposed to do. It's not necessary to provide a line comment for every line of code, because that actually just ends up making things more confusing. There are no set rules about how much or little you should comment your code with line comments, and some programmers believe you should never need any comments at all, but providing a line comment every 3-5 lines of code is probably a good rule of thumb. It's also possible to add a line comment at the end of a line that also contains other C# code, but we typically don't do that in this book.

On some occasions, you may find that you want to use several lines of comments to explain the code after the comment. We actually try to avoid that in this book (and our coding in general), but if you need to you can make your comment span multiple lines by starting with `/*`, inserting your comment, then ending it with `*/`.

## 2.5. Class

The next item in the syntax description is the line that defines the class. The IDE colors class names teal.

## 2.6. Identifiers

Whenever we need to name something in our program we need to use an *identifier* (which is just another word for name). C# has a few rules for which identifiers are legal and which aren't.

---

<sup>5</sup> We used Sandcastle to generate all the documentation we provide for the code we wrote for the book, though.

Identifiers can contain any number of letters, numbers, and underscores. They can't start with a number, and they can't contain any spaces. Identifiers can't be keywords (remember, those special C# words like `using`, etc.).

Here are some identifier examples:

Legal identifiers: `Weapon`, `playerName`, `Spell`, `DelayTime`

Illegal identifiers: `2_Names`, `int`

One other comment about identifiers. C# is *case sensitive*, which means that the identifier `playerName` is NOT the same as the identifier `playername`. Be really careful about this, and be sure to use a consistent capitalization style to avoid confusion.

## 2.7. The Main Method

All the application classes we write will have a method called `Main`. The `Main` method is the main entry point for the application; when we run our program, it will simply do the things we told it to do in the `Main` method. You should declare the first line of the `Main` method exactly as shown in the syntax description (which the IDE provides in the default template):

```
static void Main(string[] args)
```

We'll talk about the words in front of `Main` (`static` and `void`) as we need them. The part that says `(string[] args)` lets people use something called "command-line arguments." We won't be using command-line arguments in this book, but the default template we get from the IDE includes them for our `Main` method whether or not we use them. Bottom line – as long as you declare the first line of the `Main` method EXACTLY as shown in the syntax description, you'll be fine. This is made even easier by the fact that the IDE automatically provides this to you, so the only way to screw it up is for you to go change that line somehow.

## 2.8. Variables and Constants

Computers were originally designed to do mathematical calculations, so it shouldn't come as a surprise that many of the programs you write will also do calculations. You should remember from your studies of Einstein's work that

$$\text{Energy} = \text{mass} * c^2$$

Energy and mass are the variables in the equation, while `c` (for the speed of light) is a constant in the equation. C# lets us declare the required variables and constants as follows:

```
const double C = 299792458;
double energy;
double mass;
```

Don't worry about what `double` means yet – we'll get to data types in Chapter 3. At this point, you should simply realize that C# lets us declare both variables and constants.

## 2.9. Console Output

It's nice to have a program that goes off and crunches numbers, but it's even nicer to have the program tell us the answer after it figures it out! And how does the program get the numbers in the first place? That's what input and output are all about. In this section, we'll talk about how we do console output; we'll cover input and other forms of output as we need them throughout the book.

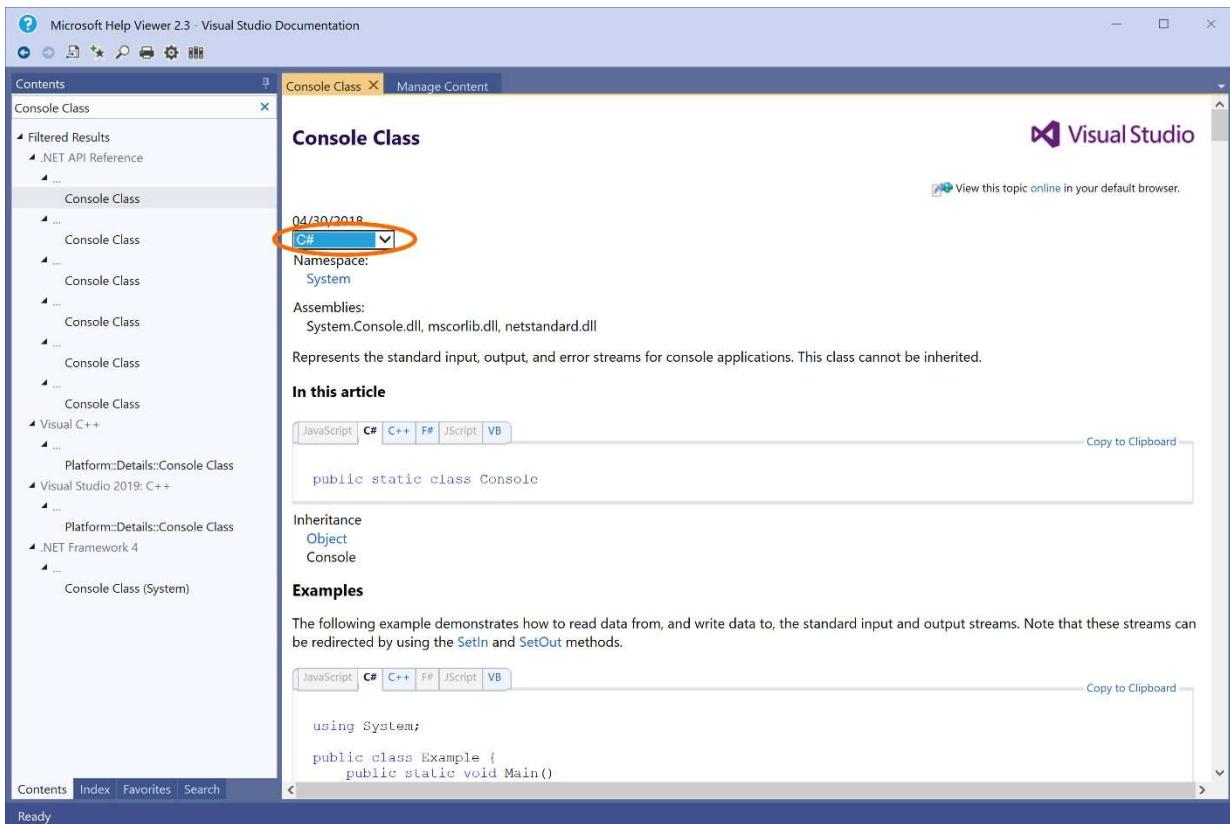
Input and Output (commonly called IO) is of course also important in game development. Gamers need to interact with the game world – that's pretty much the point of playing a game – so we'll talk about some standard ways to get and process user input once we start writing Unity scripts. We'll also obviously talk lots about game output as well. Although countless hours of productivity have been lost to people playing games that say things like "You've conquered the Orc. Do you want to leave the meadow or stay and rest?", modern gamers typically expect graphics, audio, and in some cases force feedback in their game output. This section only covers output to the console, but later in the book we'll cover more typical forms of game output.

To do our output, we'll use the `Console.WriteLine` methods, which are contained in the `Console` class. Basically, the `Console.WriteLine` method outputs whatever we tell it to print on a line on the screen, then leaves the cursor on that line so we can print more on that line. The `Console.WriteLine` method moves the cursor to the next line after printing what we tell it to.

Before we look at the details for using these methods, we should talk briefly about what methods are. A method is simply a self-contained piece of code that does something useful for us (like displaying output on a screen). You may also hear people talking about functions rather than methods; for our purposes, they're the same, though method is the correct term in the C# world.

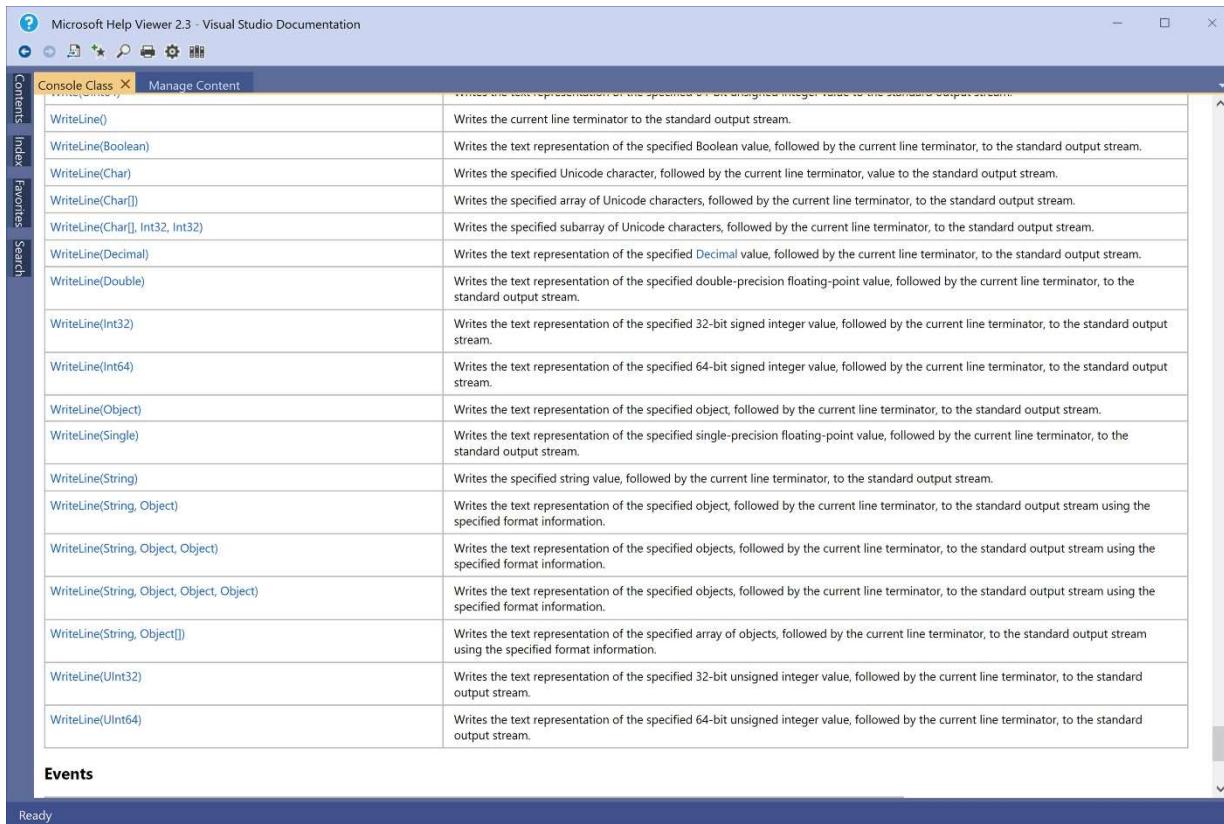
The thing that makes methods generally useful is our ability to provide information to the method when we use it. For example, a `Console.WriteLine` method that always prints "diaf, n00b" might be amusing for a while, but we're going to eventually need to output other text to the screen. From the method's perspective, information is passed into the method using *parameters*. From the perspective of the code using the method, information is passed into the method using *arguments*. Let's look at an example.

To figure out what information a method expects us to provide, we need to read the documentation. To get to the documentation page shown in Figure 2.2, we opened up help, searched on Console Class, then clicked on the first result labeled Console Class. We also made sure we selected C# in the dropdown – inside the orange circle in Figure 2.2 – to get the C# documentation. If you do the same thing, the documentation page you get should look like Figure 2.2 (without the orange circle, of course).



**Figure 2.2. Console Class Documentation**

It turns out that there are lots of things we can tell the `Console` class to do, but for now we're interested in how to use the `WriteLine` method. All the information we need is actually near the bottom of the documentation, so grab the little gray scroll box in the scroll bar on the right and drag it all the way to the end. Luckily, everything is in alphabetical order so we can easily scroll up to `WriteLine` in the Methods area of the documentation; see Figure 2.3 (we also collapsed the Contents pane on the left by clicking the "pin").



**Figure 2.3. Console.WriteLine Area**

Holy smokes! There appear to be lots of ways we can use the `WriteLine` method! We'll learn about the `String` class in Chapter 5, so for now you'll just have to trust us that we need to output a string.

The documentation tells us that the version of the `WriteLine` method we're going to use (near the bottom of Figure 2.3) has a single parameter that's a `String`. That means that when we call this method, we need to provide a single string argument to the method. For example,

```
Console.WriteLine("Hello, world");
```

will output the words "Hello, world" (without the quotes) on a single line on the screen.

So we're calling the method with a single argument, the string `"Hello, world"`. We could call the method with any other string we can think of (like, say, `"diaf, n00b"`) and the method would output that string instead. You might be wondering why we spent so much time talking about how to navigate the documentation instead of just telling you how to use the `WriteLine` method. We're glad you asked! You'll spend a significant amount of time as a game programmer reading documentation so you can use the C# and Unity classes properly. Learning how to use the provided documentation is therefore a really important thing for you to do.

## 2.10. A Word About Curly Braces

You'll notice that there are probably more curly braces in the syntax description than you're used to seeing. C# uses curly braces to block off areas of code. For example, there's an opening curly brace on

the line after the line containing the application class name, and the closing curly brace for that opening curly brace is the next to last line in the program. That means that everything between those two braces is part of the application class. Similarly, the `Main` method has an opening curly brace just below the start of the method and a closing curly brace at the end of the executable statements for the method. You'll want to make sure that you always have matching opening and closing curly braces, and that you always put them in the right places! The IDE helps with this by highlighting the corresponding opening curly brace when you type a closing curly brace.

There have been many wars (well, geek wars) over the years about which line the open curly brace should appear on. In this book, we put the open curly brace by itself on a line that starts a new block of code. This is the default Visual Studio setting, but even more importantly, it's the convention that's used in the Microsoft C# Reference.

There are actually some reasonable rules of thumb about curly brace placement (and all other coding style issues, like capitalization):

1. If you're working in a company that has coding standards or you're taking a class where the professor has coding standards, follow the coding standards whether or not they match your personal preference.
2. If you're working on an existing piece of code, follow the coding standards the existing code follows whether or not they match your personal preference.
3. If you're writing your own code "from scratch", use whatever your personal preference is, though you should use the conventions that are commonly used in the language you're coding in (C# in this book). Whatever coding standards you decide to use, be consistent following those standards throughout your code.

Unfortunately, the default Unity scripts don't follow the curly brace placement convention we want to use. That's okay, though, because we can actually change the script template that Unity uses when it creates a new C# script. In fact, we'd want to do that anyway if we're working on a Windows machine. The default line endings in the script template are for Unix (LF only) but Windows uses different line endings (CR and LF). If we don't change the line endings to Windows line endings in the script template, Unity will give us a warning about inconsistent line endings every time we edit a script.

The script template file we're going to change is named `81-C# Script-NewBehaviourScript.cs.txt`; ours is located in the `C:\Program Files\Unity\Hub\Editor\2019.3.14f1\Editor\Data\Resources\ScriptTemplates` folder. Run Visual Studio Community 2019 as an administrator and open that file. Change the contents of the file to the following:

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

/// <summary>
///
/// </summary>
public class #SCRIPTNAME# : MonoBehaviour
{
    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
```

```

{
    #NOTRIM#
}

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    #NOTRIM#
}
}

```

As you can see, we're adding documentation comments to the template, changing some of the spacing, and putting the curly braces where we want them.

Select File > Save 81-C# Script-NewBehaviourScript.cs.txt As ... Click the down arrow to the right of the Save button near the bottom right and select Save with Encoding... In the Advanced Save Options dialog, change Line Endings to Windows (CR LF) and click the OK button. Now Unity will give us new scripts with our preferred curly brace placement and those new scripts will have Windows line endings.

## 2.11. A Matter of Style

While beginning programmers tend to think that a program that works properly is perfect, more experienced programmers also recognize the importance of style. Good programming style tells us to use variable names that are descriptive (such as `firstInitial` instead of `fi`) and to use lower case letters at the start of variable names, capital letters at the start of class names, and capital letters to start the "internal words" in those names. It also tells us to use proper indentation (we use 4 spaces in this book), good commenting, and to include "white space" (blank lines) in our programs. Following these style guidelines makes your programs easier to read and understand.

Like most style matters, programming style can be largely a matter of taste. We've selected a particular style for all the examples in this book, but your teacher may want you to use a different style; most companies developing software have coding standards that specify the style that everyone in the company should use. In any case, using reasonable, consistent style guidelines will help you develop better code.

## 2.12. Solving Problems One Step at a Time

So far, we've been talking about coding details without really talking about how we actually go from a problem description to the code that solves the problem. In this book, we'll do that using the following steps:

1. Understand the Problem
2. Design a Solution
3. Write Test Cases
4. Write the Code
5. Test the Code

Problem solving is an iterative process no matter what set of steps we use. For example, we may get all the way to running our test cases before realizing that we didn't design a correct solution, so we'd need to go back to that step and work our way through to the end again. Similarly, we may realize as we write our code that we forgot something in our test cases, so we'd return to that step to make our corrections. There's nothing wrong with going back to previous steps; as a matter of fact, it's a natural part of the problem-solving process, and you'll almost always need to iterate a few times before reaching a correct problem solution.

In fact, we're going to plan on completing Steps 3, 4, and 5 many times as we solve our problems; we might even have to return to Steps 1 and 2 once in a while. Nobody in their right mind tries to write a complete *test plan* (set of test cases), write all their code, and then run all the tests against their code. Good programmers go through Steps 3, 4, and 5 with small pieces of their solution. Not only does this make solving large, complex problems more manageable, it also makes it easier to find where the bugs are when your tests fail. If you've only added a few lines of code since you last tested the code successfully, the bug is probably somewhere in those new lines of code.

Let's look at each of the five steps in more detail.

### *Understand the Problem*

This certainly seems like common sense – how can you possibly solve the problem correctly if you don't know what it is? In other words, understanding the problem requires that you understand WHAT is required. Surprisingly, lots of people try to jump right into finding a solution before they even know what problem they're trying to solve. This may be due, at least in part, to the feeling that we're not making progress toward the solution if we're "just" thinking about the problem<sup>6</sup>. In any case, you need to make sure you understand the problem before moving on.

As an example, consider the following problem: "Look through a deck of cards to find the Queen of Hearts." Do you REALLY understand the problem? Do you know what to do if you're not playing with a full deck <grin>? Do you know what to do if the Queen of Hearts isn't in the deck? Do you know what to do after you find the Queen of Hearts? Even the simplest problems require you to think carefully before proceeding to the next problem-solving step.

### *Design a Solution*

Once you know what the problem is, you need to formulate a solution. In other words, you need to figure out HOW you're going to solve the problem.

Designing your solution is, without a doubt, one of the hardest problem-solving steps. Because design is so important, we'll spend a lot of time throughout the book talking about how we go about doing this effectively. For now, let's just say that completing this step results in a set of classes, a set of objects, and a set of methods designed to solve the problem. The tricky part of this step is figuring out which classes and objects we need and how they'll interact with each other through their methods to solve the problem.

---

<sup>6</sup> One of my sons consistently gives me a hard time because he saw me sitting in a chair staring into space once and told me I should be working. I told him I was working because I was thinking about how to implement the artificial intelligence in one of our company's commercial games. I've never heard the end of that one!

## Write Test Cases

Our third problem-solving step helps you make sure you've actually solved the problem. Does your solution do what it's supposed to? It's reasonable to go back and run your program (which you'll write in the next step) a few times to make sure it solves the right problem. You're not really worried about making your program more efficient here; you just want to make sure it's correct. One way to try to make sure a program does what it's supposed to do is with *testing*, which is running a program with a set of selected inputs and making sure the program gives the correct outputs. To accomplish testing, we'll write a set of *test cases* you run against your code to make sure it's working properly.

But how can you decide how to test your program if you haven't even written it yet? We'll talk a lot more about software testing throughout the book, but it turns out that you can build most, if not all, of your test cases just by knowing what it's supposed to do. And remember from Step 1 – if you don't actually know what your code is supposed to do, you have bigger problems than coming up with the test cases!

We'll also talk about additional test cases you might want to add to your test plan based on the actual code you write in Step 4, which means we'll regularly re-visit this step after Step 4 before moving on to running the test cases.

So we know we need to write test cases, and each test case has a set of inputs we want to use for the code being tested and the outputs we expect when we use those inputs. Choosing which inputs to use is the trick, of course, because most programs can accept a huge number of inputs; if we use all of them, testing will take a LONG time! So how do we pick which inputs to use? We'll cover that in more detail soon, but at a high level we can either use the list of requirements (the things our solution is supposed to do) to pick our inputs (typically called *black box testing*) or we can use the structure of our solution to pick our inputs (typically called *white box testing*). We'll actually use a combination of these two techniques in this book. Let's see how.

There are many kinds of testing we can do for software, but in this book we're going to use two specific kinds: *unit testing* and *functional testing*. We discuss classes in much more detail in the Chapter 4, but for now, think of a program as the application class plus a bunch of other classes that the application class uses to solve the problem. If we think of each of those classes as a small unit in our solution, it makes sense to think of the testing we do on each individual class as unit testing. For unit testing, we'll add the black box test cases when we complete this step the first time, then after we write the code we'll re-visit this step to add the white box test cases based on the structure of the code we added.

So how do we document those test cases so we can use them in Step 5 to actually test the code? There are a variety of ways we can do that. For most of the test cases in this book, we'll simply write this all down in a concise set of steps for each test case. We can then execute the test case each time we need to test our code by manually following the steps in the test case.

In practice, though, it's much more useful to automate our test cases so we can rerun them whenever we change our code without having to manually pound keys for each test case. That's where a tool like NUnit comes in. We can use NUnit to document the inputs and expected outputs for each test case, then run the NUnit test cases whenever we need to. We won't use NUnit in this book, but we regularly use NUnit for testing in our company's game development work. Unity also provides the Unity Test Framework to help automate testing for Unity games, but exploring that is also beyond the scope of this book.

We're making this testing stuff sound pretty straightforward, but in the game domain there are lots of things that aren't nearly as simple to test. For example, we haven't talked at all about how we make sure game entities move properly graphically when we run the program. We haven't even talked about testing if our print rain message program prints the correct message to the screen. Those two examples have something in common – these kinds of tests are checking the overall functionality of the program rather than the behavior of specific classes in our solution. That's why this kind of testing is called – wait for it – functional testing.

Unfortunately, NUnit isn't a help to us here, because our functional tests need a person to watch the program while it runs to make sure it's behaving properly. That means we can't really automate these tests (there are functional testing tools available, but they're way more complicated than we want to take on here), so we have to manually run each functional test case every time. We still need to document the inputs and expected outputs for each of our functional tests, though; we'll show you the format we'll use for that soon.

Writing test cases probably sounds like a big job to you, but that's only because it IS a big job! Thoroughly testing your code is critical, though, because releasing buggy code can really hurt you and your company. This seems to be particularly true in the game industry; lots of people using office software will accept minor problems with that code, but gamers tend to be less forgiving when the game doesn't work exactly the way they think it should. We'll write both unit test cases and functional test cases as appropriate throughout this book.

### *Write the Code*

This step is where you take your design and implement that design in a working C# program (computer programs are commonly called *code*). The hard part of this step is doing the detailed problem solving to figure out the precise steps you need to implement in your code to solve the problem. Certainly, you'll have to learn the required syntax for C# (just as you have to learn the grammar rules for a foreign language you try to learn), but syntax is easy. If you do a careful job figuring out the steps you need to accomplish, writing the code will be easier than actually designing the solution. One more time – the hard part in the process is figuring out the correct steps, not figuring out where all the semicolons go.

Before we discuss C# source code, though, let's take a little time now to discuss *algorithms*. We'll define an algorithm as "a step-by-step plan for solving a problem"; you might hear the algorithms we write below referred to as *pseudocode* as well. Rather than formally writing algorithms or pseudocode for our solutions, we'll implement the code directly because the code is really just an expression of the steps we need to take in the C# programming language. We will, however, have to figure out those steps in a very precise and detailed way. Because we don't know enough C# to write the code directly for our current problem (finding the Queen of Hearts), let's actually work on a detailed algorithm instead to start practicing the thought process we'll end up going through when we write the code.

The important thing to remember is that code (okay, the algorithm in this case) needs to be very detailed – if you handed the algorithm to your crazy Uncle Leeroy, could he follow it correctly? One good technique for writing correct, detailed algorithms is to think of other, similar problems you've already solved and re-use parts of those solutions. As you solve more and more problems, you'll develop a large collection of sub-problem solutions that you can borrow from as appropriate.

OK, let's continue with our example of looking for the Queen of Hearts. We would probably solve this problem by using an object for the deck of cards, with a method that lets us search for the Queen of Hearts, so let's assume we're trying to actually figure out how to provide the method. Quick comment – no matter what design methodology you use (object-oriented or otherwise), sooner or later you have to figure out HOW to build the parts of your problem solution; that's where algorithms come in! Let's try to write an algorithm to solve the problem. Your first try might look something like this:

```
Look through the deck
If you find the Queen of Hearts, say so
```

What do you think? Is this a good algorithm? Stop nodding your head yes! This is not nearly detailed enough to be useful. Your crazy Uncle Leeroy has no idea how to "Look through the deck." Let's try to make the algorithm more detailed:

```
If the top card is the Queen of Hearts, say so
Otherwise, if the second card is the Queen of Hearts,
    say so
Otherwise, if the third card is the Queen of
    Hearts, say so
    . . .
```

Well, this is better, but now it looks like our algorithm will be REALLY long (52 steps for a deck of 52 cards). Also, our algorithm has to know exactly how many cards are in the deck. However, we can look at our algorithm and see that we're doing the "same" thing (looking at a card) many times. Because we're doing this repeatedly, we can come up with a much cleaner algorithm:

```
While there are more cards in the deck
    If the top card is the Queen of Hearts, say so and
        discard the Queen
    Otherwise, discard the top card
```

We now have a correct, detailed algorithm that even Uncle Leeroy can follow, and it doesn't make any assumptions about the number of cards in the deck.

When we write our code, we use three types of *control structures* - *sequence*, *selection*, and *iteration*. Sequence simply means that steps in the solution are executed in sequence (e.g., one after the other, in order). Selection means that steps in the solution are executed based on some selection, or choice. Iteration (also commonly called looping) means that certain steps in the solution can be executed more than once (iteratively). We'll look at these control structures and how we use them in C# in more detail in a few chapters, but you should understand that we simply use these structures as building blocks to develop our code (no matter what language we're programming in).

### *Test the Code*

Finally, the last step in our problem-solving process. Now that you've implemented your solution, you run your test cases to make sure the program actually behaves the way you expected it to (and the way it's supposed to). Now, you may be surprised to discover that your code may not always work correctly the first time you try it! That's when it's time to *debug* – to find and fix the problems you discover as you test your code. Debugging is a part of every programmer's life, so we'll talk about debugging techniques throughout the book as well.

## 2.13. Sequence Control Structure

The simplest problem solutions just start at the beginning and go to the end, one step at a time – that's what our print rain message program does. Let's take a look at another example:

### *Example 2.1. Reading In and Printing Out a Band Name*

Problem Description: Write an algorithm that will read in the name of a band, then print that name out.

We know, you want to start slinging code, and we're still talking about algorithms. Don't worry, we'll get there soon!

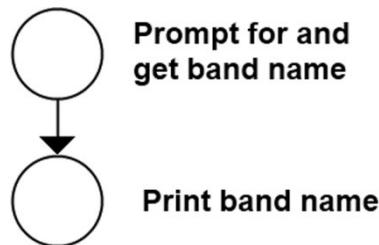
Here's one way we could solve this problem:

```
Prompt for and read in band name
Print out band name
```

Note that we have to make sure we ask (prompt) for the band name before reading it in; if you walked up to someone and simply stared at them, would they know you were waiting for them to give you the name of a band? Try it and you'll see what we mean.

For many people, a "picture" of their solution helps clarify the steps and may also help them find errors in their solution. We'd therefore also like a way to represent our algorithms pictorially. One popular way to pictorially represent problem solutions is by using flowcharts, which show how the solution "flows" from one step to the next. We'll use the same idea in something called a *Control Flow Graph (CFG)*, which captures the same information as a flowchart without requiring a bunch of special symbols. A CFG graphically captures the way control flows through an algorithm (hence the name).

Let's look at the CFG for our algorithm. The CFG looks like:



**Figure 2.4. Sequence CFG**

Because we have a sequence control structure (the program simply does one step after another), we have a sequence of *nodes* in the CFG. Each node represents a step in our algorithm; the *edges* in the CFG represent ways we can get from one node to another. In the algorithm above, we only have one way to get from the first step to the last step (since this is a sequential algorithm), so we simply add an edge from the first node to the last node.

The sequence control structure gives us our first building block for creating our problem solutions, but many of the problems we try to solve will also require us to make decisions or do things repetitively in

our problem solution. We'll get to those control structures (selection and iteration, respectively) as we need them.

## 2.14. Testing Sequence Control Structures

Testing the code we write for the problem in Example 2.1. will be pretty easy – all we have to do is run it once to make sure it prompts for, gets, and correctly prints out a band name. Notice that this is a functional test case, since we're checking the functional behavior of the program rather than a specific class (unit) in the solution. We simply pick a band name and run the program once. Here's an example test case for this simple program:

### **Test Case 1**

#### **Checking Input and Output**

Step 1. Input: The Clash for band name.

Expected Result:

Band Name : The Clash

For our functional test cases, each test case represents one complete execution of the program; no more, and no less. We only need one test case here because we only have to run the program once to fully test it.

## 2.15. Putting It All Together

Let's walk through the entire problem-solving process for the print rain message program. Here's the problem description:

Print the lines:

Hello, world  
Chinese Democracy is done and it's November  
Is it raining?

to the screen.

### *Understand the Problem*

There's not much to worry about here, since the problem seems to be easy to understand.

### *Design a Solution*

We can easily solve this problem using an application class that prints out each line in the message, so that's what we'll do here. Don't worry; the design step will become more interesting soon! We're just going to use a single class ([Program](#)) with a single method ([Main](#)), so we can move on to the next step.

### *Write Test Cases*

Since this program won't have any user input, all we have to do is run it to make sure it prints out the required message.

**Test Case 1****Checking Message**

Step 1. Input: None.

Expected Result:

```
Hello, world
Chinese Democracy is done and it's November
Is it raining?
```

*Write the Code*

Here's the completed code for the program:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrintRainMessage
{
    /// <summary>
    /// A class to print a rain message
    /// </summary>
    class Program
    {
        /// <summary>
        /// Prints a rain message
        /// </summary>
        /// <param name="args">command-line args</param>
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, world");
            Console.WriteLine("Chinese Democracy is done and it's November");
            Console.WriteLine("Is it raining?");
            Console.WriteLine();
        }
    }
}
```

**Figure 2.5. Program.cs***Test the Code*

Now we simply run our program to make sure we get the results we expected. Figure 2.6 has a screen snap of the output when we run Test Case 1.

And that's it – we've used our five problem-solving steps to complete our first C# program!

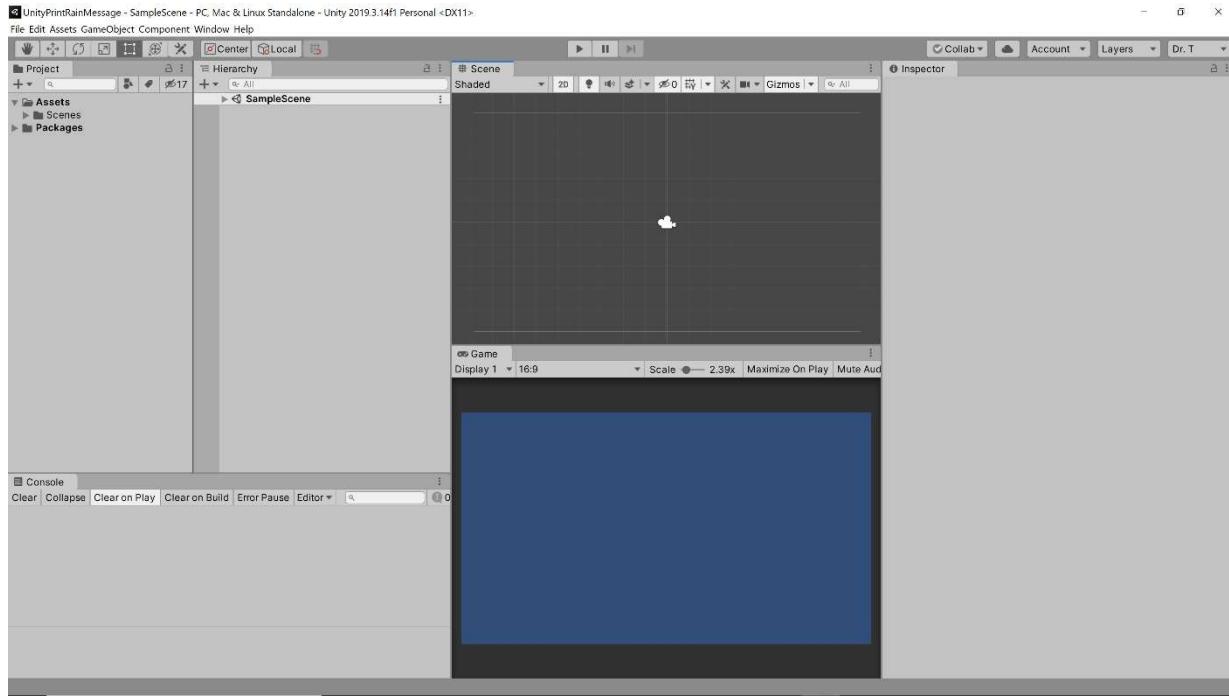


**Figure 2.6. Test Case 1: Checking Message**

## 2.16. Adding a Unity Script

Throughout the book, we'll teach you how to program in C# using both console apps and Unity scripts. In the previous section, you developed a console app to output a very exciting (if somewhat outdated) message. In this section, we'll implement a Unity script that outputs the same message.

To start, we'll create a new Unity 2D project called `UnityPrintRainMessage`; when we do, we reach the window shown in Figure 2.7.

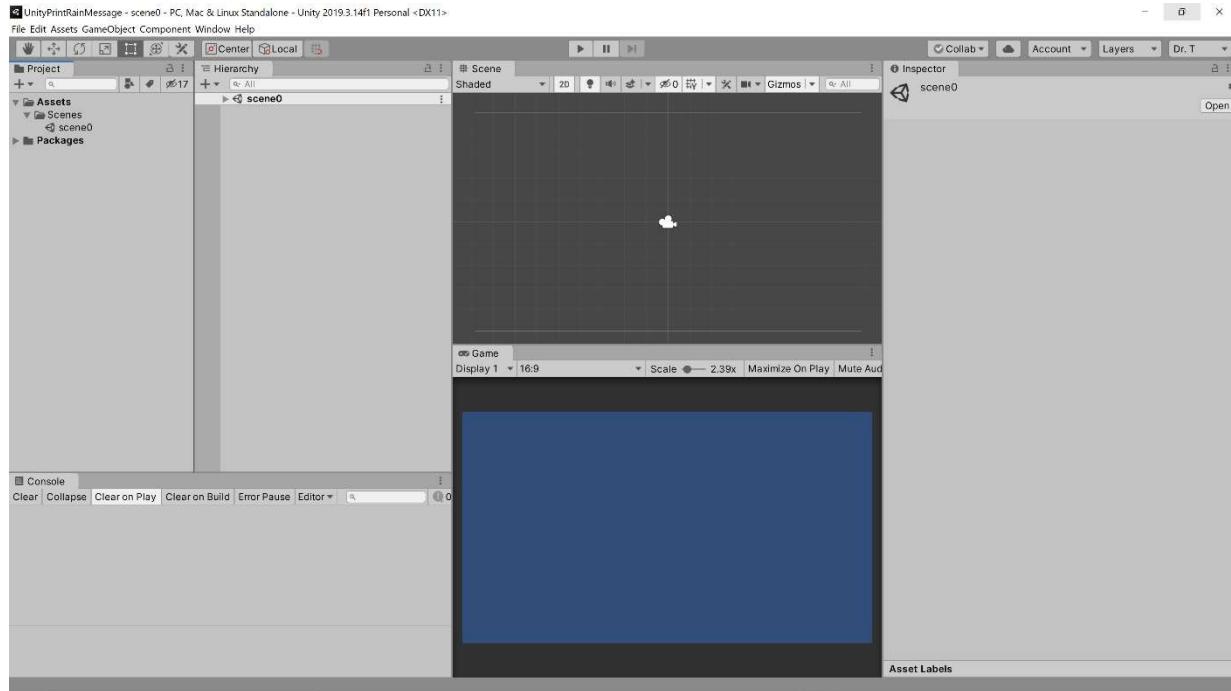


**Figure 2.7. New Unity Project**

Before we add our script, let's talk a bit about what the different windows in the Unity editor are for. The Project window on the left holds all the assets (scripts, art assets, sound effects, and so on) for our entire game. We can also use folders here to organize all our assets so we can keep them organized. Even

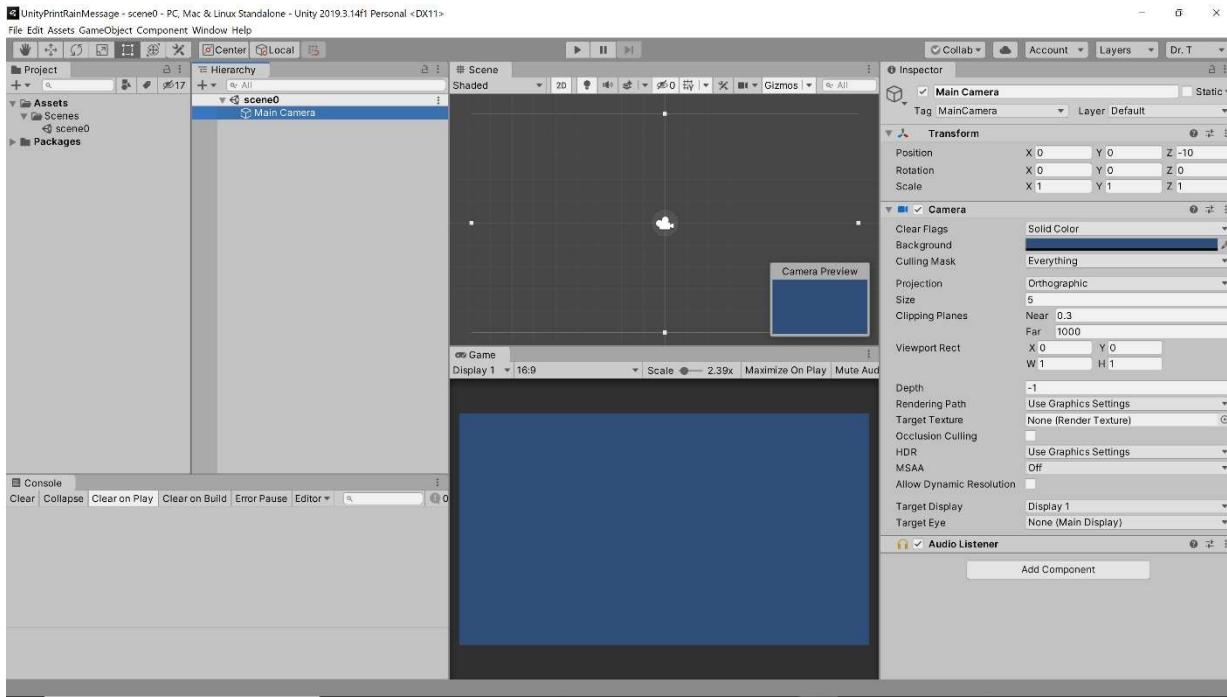
though that's not really necessary for this small example, we'll do it anyway because this is the way we'll always organize the Unity projects in this book.

If you expand the Scenes folder (by clicking the gray arrow to the left of the folder) you'll see the SampleScene that Unity provides by default when we create a new project. Let's rename that scene to scene0 instead. Right click SampleScene in the Project window, select Rename in the popup, and change the name to scene0. The editor should now look like Figure 2.8.



**Figure 2.8. Unity Project with Scene0**

The Hierarchy window, just to the right of the Project window, shows a list of all the objects that are in the current scene; at this point, that's just the Main Camera object that Unity provided by default when we created the project. Expand scene0 in the Hierarchy window, then left-click the Main Camera in the Hierarchy window (or in the Scene view to the right of it) to get the window shown in Figure 2.9.



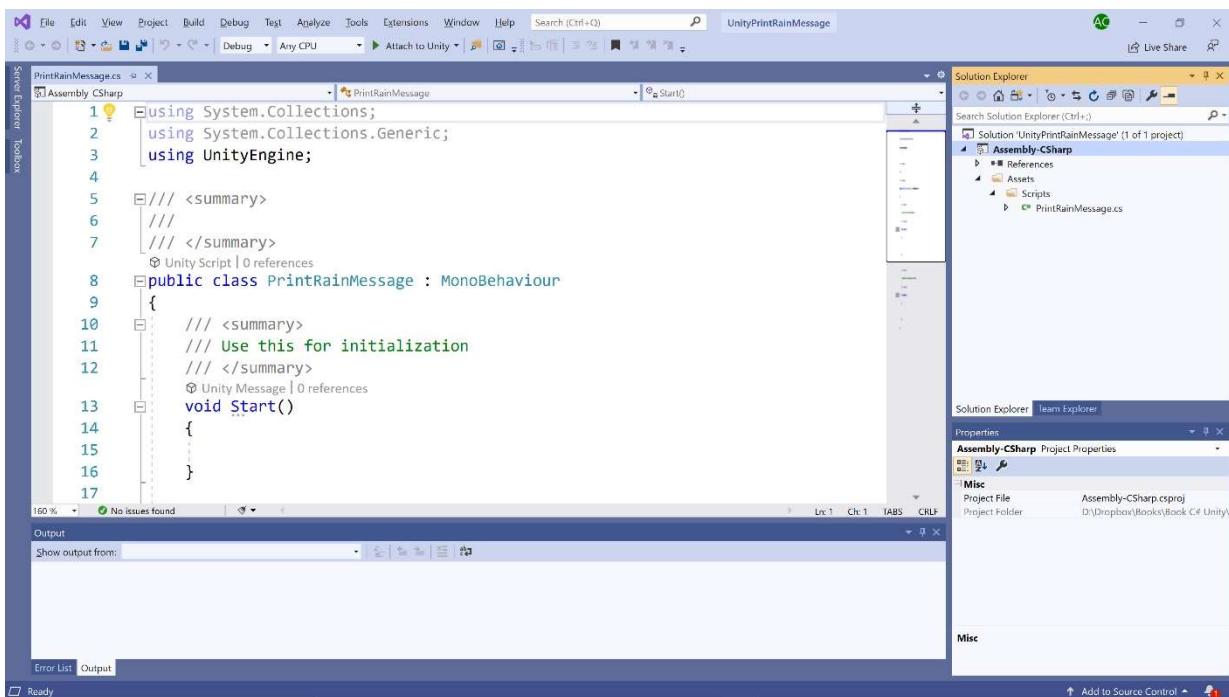
**Figure 2.9. Main Camera Selected**

As you can see, the Inspector (all the way on the right) has been populated with characteristics of the Main Camera. We can use the Inspector to look at these characteristics for any object in the current scene or for any object listed in the Project window (though we don't have any of those for this example).

Unity uses a *component-based* system for the objects we include in our games. This is a very powerful technique, because it lets us attach behaviors to our objects just by attaching the appropriate components to those objects. In examples later in the book, you'll see us attach colliders (so our objects run into stuff), rigidbodies (so our objects obey the laws of physics), and other components that we need. This example and this book as a whole are focused on the C# scripts we use in Unity, though, so why are we talking about components? Because a script is also just a component we attach to one or more objects.

Add a folder called scripts to the Project window. Right-click that folder, select Create > C# Script, and name the new script PrintRainMessage. Now double-click the new script in the Project window.

Whoa, how awesome is that! Unity automatically opens up Visual Studio for us, because it "knows" we want to edit the script. At this point, your new Visual Studio window should look like Figure 2.10 (after you expand the appropriate folders in the Solution Explorer on the right).



**Figure 2.10. Starting PrintRainMessage Script**

This doesn't look like what Visual Studio gave us when we created a new Console Application project, but don't be alarmed; we'll cover the differences as we do more complicated examples as we move forward in the book. For now, we're just going to add the code we need to print our message to the `Start` method (instead of the `Main` method like we did for our console app). Go ahead and copy and paste the four lines we used in our console app into the body between the open and close curly braces of the `Start` method. You should also delete the `Update` method (including the comment above it) because we won't be using that method for this example.

Caution: Because we named the new script `PrintRainMessage` when we created it, the name of the class that's created (the name right after `public class` in the script above) is also `PrintRainMessage`, which is exactly correct. Unity requires that the name of our `.cs` file and the name of the class in that `.cs` file match exactly. This is good programming practice and is perfectly reasonable.

Unfortunately, if you leave the script name `NewBehaviourScript` (the default script name) when you create the script and then rename the script to the name you really want later in the Unity editor, Unity changes the name of the `.cs` file but leaves the name of the class in that file `NewBehaviourScript` instead of changing it to the new name. The script will compile fine in Visual Studio, but Unity will give you an error when you try to attach it to a game object in your scene.

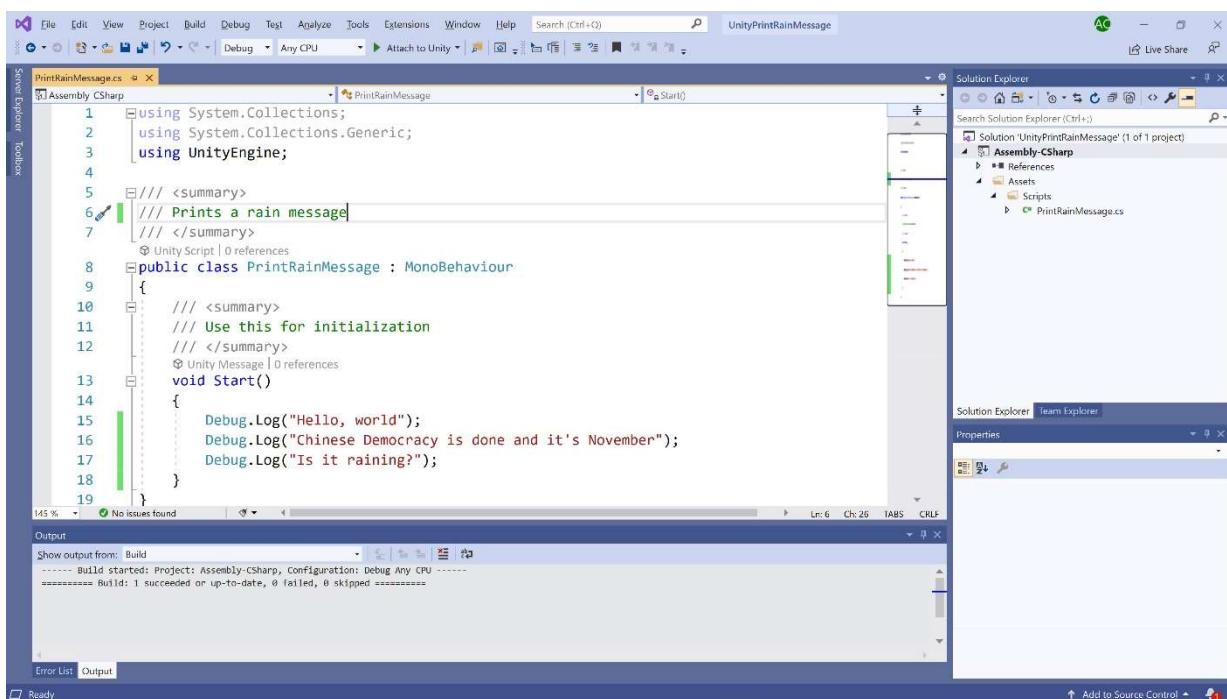
To fix this, open up the script in Visual Studio. Right click on the `NewBehaviourScript` class name in the script and select `Rename...` (the second choice from the top) in the popup menu. Type in the name of the script as the class name and press `<Enter>` or click the `Apply` button in the renaming dialog. Now the name of class and the name of the `.cs` file match, so Unity will let you attach the script to game objects in your scene.

Fill in the documentation comment near the top of the code saying the code prints a rain message. If you F6 to build at this point, you'll get an error message for each of your calls to the `Console.WriteLine`

method. Remember how we had to include a using directive for the `System` namespace to get access to the `Console` class in our console app? If you look at the using directives at the top of the source code, you'll see that there isn't a using directive for the `System` namespace.

We discussed that error here because forgetting to use a namespace you need is a fairly common mistake for beginning programmers. Unfortunately, in this particular case adding a using directive for the `System` namespace won't solve our problem. The code would compile fine, but it wouldn't actually print the message to the Console window in the Unity editor.

Delete the last call to the `Console.WriteLine` method (that just prints a blank line, which we don't need in the Console window) and change the other 3 lines of code to call the `Debug.Log` method instead. The Unity Scripting Reference tells us this method logs a message to the Unity Console, which is exactly what we want. Your code should now look like Figure 2.11.



**Figure 2.11. Final PrintRainMessage Script**

Build one more time to make sure everything compiles, then save, close Visual Studio, and go back to the Unity editor. Now click the arrow facing to the right (the Play button) near the top center of the Unity window, look at the Console window, and wait for the rain message ...

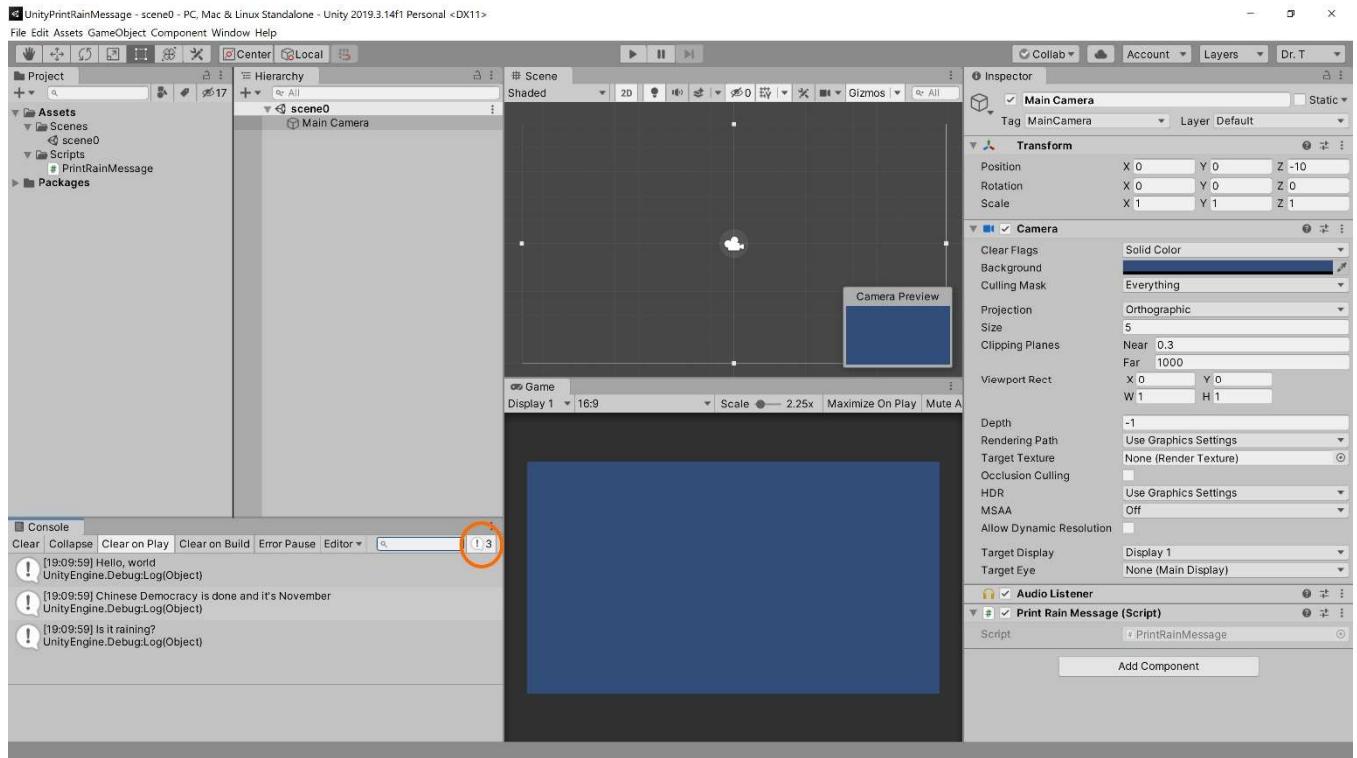
Well heck. Click the Play button again to stop the game. Note: you can also use `Ctrl + P` to start and stop your game.

What's going on? The problem is that even though we wrote a beautiful C# script, we haven't actually added it as a component to any of the objects in the current scene. Drag the script from the Project window onto the Main Camera in the Hierarchy window or the Scene view. If you select the Main Camera, you can see in the Inspector that the Print Rain Message script has been added as a component to the Main Camera.

Click the Play button and watch the Console window, and you should see the rain message. Awesome.

Your Unity window should now look like Figure 2.12 (without the orange circle). If you've done everything discussed above properly and you still don't see the rain message in the Console window, try clicking the "log toggle" (circled in orange below) to enable logging to that window. Finally, be sure to Ctrl+S to save your scene.

Great job making your first C# script in Unity!



**Figure 2.12. Final Unity Project**

# Chapter 3. Data Types, Variables, and Constants

Computer programs are all about storing, processing, and displaying information. We've already discussed processing information (by calling methods) and displaying information (by writing to the console, for example), but we haven't really discussed how information is stored in memory. That's what this chapter is about.

## 3.1. Variables

Whenever we need to store something (such as a name, the value of Pi, or a `GameObject`) in our program, we need to *declare* a variable or a constant. We declare variables and constants using the syntax shown below.

---

**SYNTAX:** Variable and Constant Declaration

### Variables

```
dataType variableName;
```

`dataType`, the data type for the variable  
`variableName`, the name of the variable

We can also give a variable its initial value when we declare it by using

```
dataType variableName = value;
```

`value`, the initial value for the variable

### Constants

```
const dataType ConstantName = value;
```

`dataType`, the data type for the constant  
`ConstantName`, the name of the constant  
`value`, the value for the constant

---

As we go through the book, we'll discuss appropriate capitalization style for the different programming constructs we use. These aren't required by C#, but they are the most commonly-used styles in the C# world.

For variable names, we use something called *Camel case*, which means that the variable name starts with a lower case letter, then each of the following words starts with a capital letter (like `variableName` in the syntax description above). For constant names, we use something called *Pascal case*, where all the words, including the first one, start with a capital letter (like `ConstantName` in the syntax description above). There are certainly exceptions to these style conventions, but we'll use them consistently throughout this book.

Data types in C# fall into two categories: *value types* and *reference types*. We'll look at value types first, since they're a little easier to understand. Before we start, though, let's look at a simplified representation of memory (the main memory from Chapter 1); see Figure 3.1. As you can see in the figure, we can conceptualize memory as a whole bunch of storage locations or boxes. The numbers to the left of the memory locations are the addresses of those locations. By giving each location a unique address, we can easily access specific locations in memory. You won't actually need to explicitly use memory addresses for the programs in this book, but it's important that you understand this basic concept. We don't have to use the actual memory addresses because when we declare a variable or constant, we're really just naming a particular memory location with the name. That way, we can refer to the memory location by using its name instead of the actual memory address.

		• • •
40	01000001	
41	11010101	
42	00000000	
43	01110101	
44	10011001	
45	01010101	
		• • •

**Figure 3.1. Simplified Memory Representation**

So what actually gets stored in each location in memory? Ones and zeros, of course! Modern computers use binary – 1s and 0s – for everything they store and do. Luckily, we don't have to deal directly with

binary, at least most of the time, though there are occasions where we might want to deal with specific bits (a single 1 or 0). The important thing to remember, though, is that a memory location can never be empty; it always has 1s and 0s in it.

Okay, let's say that memory location 52 holds the following 1s and 0s:

```
00000000000000001111111111111111
```

What do these 1s and 0s mean? We don't actually know until we know what *data type* is being stored in that memory location. For example, if memory location 52 has been set up to store an integer, the sequence of 1s and 0s would be interpreted as 65,535. If instead memory location 52 has been set up to store the Red, Green, Blue, and Alpha values for a particular pixel on the screen, the sequence of 1s and 0s would be interpreted as a completely blue, completely opaque pixel value. All memory locations hold 1s and 0s, but it's the data type associated with the memory location that determines how those 1s and 0s will be interpreted.

There's one more important thing that a data type tells us – what operations are valid for that data type. For example, it makes sense to be able to add integers in the normal way, but adding an integer to a true or false value (a Boolean) doesn't make sense at all. Let's start looking at the various C# value types.

### 3.2. Value Types, Variables, and Constants

It turns out that there are some data types that are commonly used no matter what programming language we're using. Most of the data types discussed in this chapter are called *value types*. They're the most basic data types there are in C#, and their values are stored directly into memory locations as described above.

So what are we really doing when we declare a variable or constant in our program? We're setting aside a memory location to hold that variable or constant. Remember our discussion about main memory back in Chapter 1? We said that we run our programs from main memory, and that includes using main memory to store information our programs need to use.

For a variable, we shouldn't make any assumptions about the initial value of the variable – the contents of the location are called the *value* of the variable – and we can change what's in that location as many times as we want as the program executes. For a constant, the memory location contains whatever we said the constant's value would be, and we're never allowed to change it as the program executes. When we decide to declare a variable or constant in our program, we use the syntax shown in the previous section.

The variable or constant names need to be legal identifiers (remember we discussed the rules for those in Chapter 2). As discussed above, the data type for a variable or constant tells us two things:

1. What values the variable or constant can have, and
2. What operations are valid for the variable or constant

Let's look at these a little more closely in the context of the following variable declaration:

```
double energy;
```

The `energy` variable is declared as a `double` (a floating point number, which is a number with a decimal point, such as 3.2 or 12.75), which tells us that the only values this variable can have are floating point numbers. In other words, we can't store a character, a string of characters, or any other value that's not a floating point number in this variable.

So what values do our variables have right after we declare them? It actually doesn't matter, because it's simply good programming practice for us to explicitly give a variable a value before we try to use it. We don't have to give the variable a value when we declare it, but we do need to give it a value before we use it.

We also know which operations are valid for this variable. We know we can add this variable to another floating point number, but we can't add it to a character (for obvious reasons). We can also include the variable in any other mathematical operations that are valid for floating point numbers.

Similarly, if we wanted a constant for the speed of light, we could use:

```
const double C = 300000000;
```

and we could use `C` in floating point operations in our program.

It's also possible to declare multiple variables on the same line, as in:

```
double radius, area;
```

which declares two `double` variables, one called `radius` and one called `area`. Although this is allowed in C#, and is in fact useful in some specialized cases, most programmers consider this poor style. Throughout the book, we'll only declare one variable or constant per line.

### 3.3. Integers

Integers are whole numbers: -2, 0, and 42 are some examples. We might use an integer variable to count things, store the number of points a student gets on a test, keep track of the current pixel location of a game object, or indicate the amount of damage a particular weapon inflicts. Here are a couple of example declarations:

```
int numCars;
int health;
```

The valid operations for integers are generally as you'd expect; addition, subtraction, and multiplication work in the usual ways. Division, however, is a little different. When we divide one integer by another in C#, the result is always an integer as well. For example, `6 / 4` will give us 1, not the 1.5 you might expect. Just think of this as the math you did a long time ago, where you'd do integer divisions to get a quotient and a remainder. If we use that kind of math, 6 divided by 4 is 1 with remainder 2. C# also lets us calculate the remainder: `6 % 4` will give us 2. In addition, C# gives us increment (`++`) and decrement (`--`) operators. These operators let us easily add 1 to or subtract 1 from an integer. We'll show how to use them later.

C# actually gives us four different data types for integers: `byte`, `short`, `int`, and `long`<sup>7</sup>. So why are there four different integer data types? It all comes down to how many bits the computer uses for a variable or constant of that data type. Remember the "box in memory" analogy? The different integer data types in C# get different size boxes (8 bits for `byte`, 16 bits for `short`, 32 bits for `int`, and 64 bits for `long`). Why do people care about that? Because if you pick a data type that's too small, you'll get errors if you do math with large numbers. On the other hand, if you pick a data type that's too large, you'll waste memory space, which is inefficient and could also cause problems, especially in games because memory can be limited on some platforms. For the problems in this book, though, we won't really need to worry about the size of our integers, so we'll just use the `int` data type whenever we need an integer. For your reference, though, we provide the range of values each of the integer data types can hold below.

`byte`

Minimum Value: 0  
Maximum Value: 255

`short`

Minimum Value: -32,768  
Maximum Value: 32,767

`int`

Minimum Value: -2,147,483,648  
Maximum Value: 2,147,483,647

`long`

Minimum Value: -9,223,372,036,854,775,808  
Maximum Value: 9,223,372,036,854,775,807

### 3.4. Floating Point Numbers

We obviously also need to store and process numbers that have a decimal part as well as a whole number part: -3.7, 0.00001, 5.0, and 483.256 are some examples. These numbers are called real numbers in mathematics, and C# provides several data types to store real numbers. We'll discuss the floating point types first.

We might use a floating point variable to sum a set of floating point numbers, to hold a student's class average, or to hold the x and y components of a 2-dimensional velocity vector. Here are some examples:

```
double sum;
double average;
```

The valid operations for floating point numbers are as you'd expect, with addition, subtraction, multiplication, and division defined in the normal ways.

C# gives us two different data types for floating point numbers: `float` and `double`. A `float` variable or constant gets 32 bits of memory, and a `double` variable or constant gets 64 bits. You can use `float` if you're worried about how much memory your program uses, but we don't need to worry about that for

---

<sup>7</sup> In fact, C# gives us four additional integer data types, but because not all the .NET languages have the additional data types, using them can affect how portable our code is. We'll therefore avoid using those additional integer types.

the problems in this book. On the other hand, Unity tends to favor `float` over `double`, so we'll use `float` whenever we need a floating point number. We provide the range of values for each of the floating point data types below. The provided numbers consist of a mantissa and an exponent.  $1.5 * 10^{-45}$ , for example, is a very small number!

**`float`**

Smallest Value: positive or negative  $1.5 \times 10^{-45}$

Largest Value: positive or negative  $3.4 \times 10^{38}$

**`double`**

Smallest Value: positive or negative  $5.0 \times 10^{-324}$

Largest Value: positive or negative  $1.7 \times 10^{308}$

### 3.5. Decimals

It might seem like the floating point data types could handle all our real number needs, but that's not quite true. To understand why, you need to realize that the binary representation of a floating point number – the 1s and 0s in memory – is actually an approximation of the number. Let's talk about the `double` data type as an example to see why.

In the real world, there are an infinite number of numbers in the range that a `double` can represent because the real world is *continuous*. Because everything in a computer is stored using a limited number of bits, the computer works in the *discrete* domain. Why does this matter? We need just a little more understanding of how binary works before we can answer that.

Let's say we have a single bit, which can be 1 or 0. How many unique values can we represent? Of course, the answer is 2. If we have 2 bits instead, we can represent 4 unique values using all possible combinations of those 2 bits: 00, 01, 10, and 11. We could do this manually ad infinitum (or even ad nauseum), but we don't have to. It turns out that there's a mathematical relationship between the number of bits we need and the number of unique values we're trying to represent. So far, we have:

1 bit can represent 2 unique values

2 bits can represent 4 unique values

The relationship becomes much clearer if we rewrite it as follows:

1 bit can represent  $2^1$  unique values

2 bits can represent  $2^2$  unique values

So the general relationship is that

$b$  bits can represent  $2^b$  unique values

If you know how many bits you have, you can use the above relationship to determine how many unique values you can represent. What if you know how many unique values you want to represent and need to know how many bits you need? You can just use the inverse relationship:

$n$  unique values can be represented using  $\log_2 n$  bits

Okay, back to our discussion about `double`. We know that a `double` is stored in 64 bits, so the total number of unique values that we can store in a double is  $2^{64}$ , which is about  $1.8 * 10^{19}$ . That's certainly a really big number, but it's not infinite! That means that we have to approximate lots of those infinite numbers in the real world with a single sequence of bits in the computer, making them indistinguishable from each other.

Because of this, we can't get perfect precision with our floating point data types. The precision we do get is quite often close enough, though, so at this point you could reasonably be asking "What's the big deal?"<sup>8</sup> The big deal is that there are some areas where precision actually matters more.

Think about a program that runs a cash register. That program needs to be able to precisely store dollars and cents (or whatever currency is commonly in use where the cash register is installed), so allowing imprecision in the variables we use won't be acceptable. Luckily, the `decimal` type helps solve that problem.

How can it possibly do that given our long binary discussion above? By reducing the range of numbers it stores (they can range from -79,228,162,514,264,337,593,543,950,335 to 79,228,162,514,264,337,593,543,950,335) and by reducing the number of decimal places it can store (a `decimal` stores up to 28 decimal places). Given those constraints, the 128 bits used for the `decimal` type can more precisely represent the possible values in its range.

Finally, an example:

```
decimal price = 99.99m;
```

Notice that we append an `m` (or an `M` if you prefer) after the real number literal to explicitly indicate that it's a `decimal` value. The valid operations for `decimal` are as you'd expect, with addition, subtraction, multiplication, and division defined in the normal ways

The tradeoff we make between `decimal` and the floating point types is that `decimal` gives us more precise numbers in a smaller range. That makes the `decimal` type the best choice for some domains, like cash register software, but a worse choice for other domains, like quantum physics. By considering what range of numbers your program needs to store and with what precision, you can make a reasoned choice about which data type to use.

## 3.6. Characters

A variable or constant declared as a `char` (short for character) can hold a single 16-bit Unicode character (such as A, 7, and #). In many cases, this character will be a letter or a digit, but it can also be a space, a punctuation mark, or other characters. We might use a character variable to store a menu choice or a student's first initial. And, of course, here are some examples:

```
char menuChoice;
char firstInitial;
```

In this book, we won't worry too much about the valid operations for characters. As long as we don't try to do anything strange (such as multiplying two characters!), we should be fine.

---

<sup>8</sup> Or, if you're really cruel, "Who cares?" or even "Whatever"

## 3.7. Booleans

A `bool` (short for Boolean) variable or constant can only have one of two values: `true` or `false`. While this might not seem particularly useful to you right now, you'll find that `bool` variables are handy as flags to tell us whether or not to do certain things. Here are a couple of example declarations:

```
bool timeToMowTheGrass;
bool active;
```

The valid operations for `bool` variables are probably somewhat unfamiliar to you: the ones we'll use in this book are called *and*, *or*, and *not*. Let's consider each of these.

When we *and* two Boolean operands, we're really trying to find out if they're both `true`. The operator that we'll use for *and* in C# is `&&`. That means that

```
true && true is true
true && false is false
false && true is false
false && false is false
```

In other words, the result of an *and* will only be `true` if both operands are `true`.

When we *or* two Boolean operands, we're really trying to find out if at least one of them is `true`. The operator that we'll use for *or* in C# is `||`. That means that

```
true || true is true
true || false is true
false || true is true
false || false is false
```

In other words, the result of an *or* will be `true` if one or both of the operands is `true`.

The last Boolean operator we'll consider, *not*, is slightly different because it's a unary operator (it takes only one operand). The idea behind *not* is that it will "flip" the operand; the operator that we'll use for *not* in C# is `!`. That means that

```
!true is false
!false is true
```

That's all you need to know about valid Boolean operators.

## 3.8. Operations

Recall that a data type tells us the valid values and operations for variables and constants of that data type. For easy reference, we provide a list of the operations you're most likely to use with the value types in an introductory course below.

### Add

Operator: `+`

Data Types: `byte`, `short`, `int`, `long`, `float`, `double`, `decimal`

### Subtract

Operator: -

Data Types: byte, short, int, long, float, double, decimal

### Multiply

Operator: \*

Data Types: byte, short, int, long, float, double, decimal

### Divide

Operator: /

Data Types: byte, short, int, long, float, double, decimal

### Remainder

Operator: %

Data Types: byte, short, int, long

### Increment, Decrement

Operator: ++, --

Data Types: byte, short, int, long

### Equal, Not Equal

Operator: ==, !=

Data Types: byte, short, int, long, float, double, decimal, char, bool

### Less Than, Greater Than

Operator: <, >

Data Types: byte, short, int, long, float, double, decimal, char, bool

### Less Than or Equal To, Greater Than or Equal To

Operator: <=, >=

Data Types: byte, short, int, long, float, double, decimal, char, bool

### And, Or, Not

Operator: &&, ||, !

Data Types: bool

## 3.9. Choosing Between Variables and Constants

Let's think about variables and constants for a moment. Variables are things that can change, or vary; therefore, any variables we declare in our program can be changed as many times as we want as the program executes. Constants, however, are unchanging, or constant; therefore, once we declare a constant in our program, we can't change its value in other parts of the program. If we need a memory location to store some value or multiple values that change during the execution of the program, we should declare that location as a variable (because its contents change). If we need a memory location to hold some value that will never change during program execution, we should declare that location as a constant (because its contents don't change).

Most beginning programmers understand the need for variables, but the need for constants is less clear. If we know that the maximum speed for a game character is 75, for example, why not simply use 75 wherever the maximum speed is used in the program? We'd certainly save typing

```
const int MaxSpeed = 75;
```

so why not just use 75 everywhere we need it? By not using a constant, we introduce two problems. First, what happens if we decide while tuning the game that we really need the maximum character speed to be 100 rather than 75? We'll have to go through all our code, changing each 75 to 100. It's even worse than that, though! Our second problem is that the 75 is commonly called a *magic number* because someone reading the code needs to magically know what the 75 actually means (in this case, max speed). What if we've used 75 both for the max character speed and the damage value for a particular weapon? When we go through our code changing each 75 to 100, we have to make sure we only change the correct 75s – only those for max speed, not those for weapon damage. This is WAY more trouble than it's worth! Just use a constant for each unchanging value you use in the program and you'll save yourself lots of grief.

### 3.10. Giving Variables a Value

We already know how to give constants a value – we do that when we declare the constant. But how do we give variables a value? There are actually two ways we can do this: with an *assignment statement* and by reading in the value as input. Let's look at assignment statements now and defer the input discussion until later. Here's the syntax for assignment statements:

#### SYNTAX: Assignment Statements

```
variableName = value;
```

*variableName*, the name of the variable  
*value*, the initial value for the variable

By the way, the value can be a *literal*, like 3 or 5.7, or it can be an *expression*, like 7 \* 3 or oneVariable - anotherVariable.

So what really happens with an assignment statement? First, the computer figures out what the value to the right of the = is. For literals, it's just the value of the literal. For expressions, the computer needs to evaluate the expression to figure out what its value is<sup>9</sup>. It then takes this value and puts it in the variable on the left of the =. Some example assignment statements using a variety of data types are provided below.

```
byte
byteAge = 38;
```

```
short
shortAge = 38;
```

<sup>9</sup> The computer actually evaluates literals also, but of course figuring out the value of a literal like 1 is pretty easy!

```

int
intAge = 38;

long
longAge = 38;

float
floatGPA = 3.99f;

double
doubleGPA = 3.99;

decimal
price = 99.99m;

char
firstInitial = 'A';

bool
likesSpinach = true;

```

One warning about assignment statements – the data type of the expression must match the data type of the variable (with a few exceptions, discussed below). In other words, if the expression evaluates to a `float` value, you'd better be putting that value into a `float` variable!

This can get a little tricky, especially with the way C# evaluates the literals (such as `3.99`) that you use. For example, when C# sees a decimal number, it assumes that it's a `double`. That means that if we tried to use

```
floatGPA = 3.99;
```

the compiler would complain, because we're trying to put a `double` into a `float` variable. To get around this problem, we add an `f` to the end of our numeric literal (using `3.99f` instead of `3.99` in this example), which tells C# to treat that literal as a `float`. In this book, we'll almost always be using `int` and `float` as our numeric data types. When C# sees a whole number it interprets it as the appropriate integer data type so we don't have to worry about adding anything to the end of our integer literals, but we will need to be more careful with our `float` literals.

## 3.11. Type Conversions

The type checking provided by C# is generally a very good thing. There are times, however, where our inability to mix numeric types like `int` and `float` causes us trouble. Luckily, C# gives us both *implicit type conversion* and *explicit type conversion* capabilities.

C# has some well-defined rules for when it will implicitly convert data types for us; when we say C# does this implicitly, we mean we don't have to do anything special in our code to make it happen. For example, say we're adding a `float` to a `double` in an expression; C# will automatically convert the `float` to a `double` so the result is a `double`. Similarly, if we're multiplying an `int` by a `float`, C# will automatically convert the `int` to a `float` so the result is a `float`. The general rule is that C# will *promote* the smaller type to match the larger type in an expression. This rule is used because it ensures

that the conversion will never cause you to lose information because we never reduce the size of the type we're converting.

There will also be times, though, when we need to explicitly tell C# we need to do a type conversion. For example, say we've added up the number of Compact Discs (CDs)<sup>10</sup> each of our friends has, and we've counted our friends at the same time (both of these are integers). Now we want to calculate the average number of CDs (a `float`) for our friends. The intuitive thing to try would be

```
averageCDs = sumCDs / numFriends;
```

but this won't work. Remember, integer division only gives the quotient, so we'd be missing the fractional part of `averageCDs`.

It turns out that there's a very easy way to handle this in C# with explicit type conversion, which is also called *type casting*. For example, to treat `sumCDs` as a `float` in our equation, we simply use the following:

```
averageCDs = (float)sumCDs / numFriends;
```

Doing this makes the computer temporarily treat `sumCDs` as a `float` in the expression, solving our integer division problem. You should note that, once we say we want to treat `sumCDs` as a `float`, the implicit type conversion rules we discussed above also cause `numFriends` to be converted to a `float` before the division is actually completed. It's important to remember that `sumCDs` is still an integer variable (so we'd have to type cast it again later if we needed to treat it as a `float` again). In other words, the type cast doesn't change the actual data type of the variable; the variable is just temporarily treated as a `float` while the computer evaluates the expression above.

Let's look at how this works using actual numbers. Say that `sumCDs` is 403, and `numFriends` is 5. Using the type cast `(float)sumCDs` makes the numerator the float value 403.0, and implicit type conversion makes the denominator the float value 5.0. The division is completed, and the result (80.6) is put in the `averageCDs` variable.

Now, imagine that we mistakenly did the following instead:

```
averageCDs = (float)(sumCDs / numFriends);
```

The integer division of  $403 / 5$  would happen first, yielding the value 80. This value would then be converted to the `float` value 80.0 by the type cast, and this value would be put in the `averageCDs` variable. But this is incorrect! When we include a type cast, that type cast applies to the value immediately to the right of the type cast. We have to do this carefully to make sure we get the results we need from the type cast. A good rule of thumb is to only type cast specific variables (like `sumCDs`) rather than trying to type cast parenthesized expressions (like `sumCDs / numFriends`). This rule doesn't always work, though; you still have to think about what you're trying to accomplish when you do the type cast.

---

<sup>10</sup> In the old days, people actually bought music this way, or even on vinyl, cassettes, or (yikes!) 8-track tapes. You may need to go to a museum to see music on physical media, but the example still works!

Why does C# require that we explicitly indicate our requirement for a type cast anyway? Because we may lose information when we do the type cast. For example, say our calculation above set `averageCDs` to 80.6, and then we did the following:

```
approximateAverageCDs = (int)averageCDs;
```

The `approximateAverageCDs` variable would be given the value of 80 because the type cast to `int` causes the computer to throw away (or truncate) the decimal portion of the value in `averageCDs`. This may be OK with us, but it is a loss of information, so the compiler requires that we explicitly say we want it to happen.

We should be careful when using type casting, of course. Most of the type casting you're likely to do in an introductory course will be casting `int` variables to `float` and type casting `float` variables to `int`. Remember, when you type cast a `float` to an `int`, the floating point number is truncated, not rounded. Also, you can never type cast the variable on the left hand side of the `=`; only variables on the right hand side can be type cast.

## 3.12. What About Reference Types?

Reference types are called that because variables that are reference types refer to an object that's been created in memory. The data type for an object is a class (these are also called *reference data types* in C#); we'll talk lots more about objects and classes in the Chapter 4, so let's actually defer our detailed discussion of reference types until the next chapter.

## 3.13. Putting It All Together in a Console App

Let's go through the entire problem-solving process for a problem that needs to use constants and variables of various types. Here's the problem description:

Calculate the area of the circles with integer radii from 0 to 5, then print out the radius and area for each circle.

### *Understand the Problem*

Do we understand the problem? We may not understand HOW to do everything yet, but do we understand WHAT our program needs to do? It seems pretty straightforward, but you should be asking yourself if "from 0 to 5" includes 0 and 5. In general, we should assume that ranges like this are inclusive (include both ends of the range), so let's move on to the next step.

### *Design a Solution*

At this point, it makes sense to simply include the code we need in the `Main` method of our `Program` class, so that's what we'll do here. We'll revisit this problem in Chapter 4 to build a true object-oriented solution.

### *Write Test Cases*

Since this program won't have any user input, all we have to do is run it to make sure it prints out the required circle info. This is a functional test.

Notice that we made sure we figured out EXACTLY what numbers we expect when we run our program. It would certainly have been easier to just put "Correct area for each radius" in our expected results, but how would you know if the area was correct? You're going to have to do the calculation at some point to make sure your program is working properly; when you Write Test Cases is the correct time to figure out what you really expect for the program output!

### Test Case 1: Checking Radius and Area Info

Step 1. Input: None.

Expected Result:

```
Radius: 0, Area: 0
Radius: 1, Area: 3.14
Radius: 2, Area: 12.57
Radius: 3, Area: 28.27
Radius: 4, Area: 50.27
Radius: 5, Area: 78.54
```

We'll actually expect each radius and area to appear on a separate line in our output; that's why we moved our Expected Result into separate lines above.

#### *Write the Code*

Let's start by deciding what variables we're going to need to include. Although we could actually solve this problem without using any variables, let's start practicing declaring and using variables in our solution.

We already know that the radius will be an integer, because that was in our problem specification. So one of our variables should be declared as

```
int radius;
```

We could have called the variable `r`, `rad`, or even `joeCool` instead of `radius`, but `radius` is a pretty good name for the variable that holds the radius, don't you think? You should always use descriptive variable names because doing so makes your code much easier to read and understand.

Okay, what about the area of the circle? Well, we know we're going to be squaring the radius and multiplying by PI, so we should plan on using either a `float` or a `double` for the area. The value for PI we're going to use is a `double` (more about that presently), but a `float` will be fine for the precision we need for the area so let's make area a `float` with

```
float area;
```

Because we know that the area of a circle is defined as  $\text{PI} * \text{radius squared}$ , we might start out with

```
area = 3.1415 * radius * radius;
```

It sure seems like C# must have a constant somewhere that gives us a more accurate number for PI, though, doesn't it? It turns out that the `Math` class gives us just such a constant, called (not surprisingly) `Math.PI`. So now we have

```
area = (float)Math.PI * radius * radius;
```

Because `Math.PI` is declared as a `double` in the `Math` class, we need to use the explicit type cast to change it to a `float`. Why don't we have to type cast `radius` to `float`? Because the implicit type conversion rules automatically promote the variable to a `float` before evaluating the expression.

Given what we've figured out so far, we can write the code that calculates and provides the output for a circle with radius 0; that code is provided in Figure 3.2.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CirclesApplication
{
    /// <summary>
    /// Outputs a variety of circle characteristics
    /// </summary>
    class Program
    {
        /// <summary>
        /// Outputs radius and area for circles with radii 0 through 5
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            int radius;
            float area;

            // circle with radius 0
            radius = 0;
            area = (float)Math.PI * radius * radius;
            Console.WriteLine("Radius: " + radius +
                ", Area: " + area);

            Console.WriteLine();
        }
    }
}
```

**Figure 3.2. Program.cs**

You may be wondering how the `Console.WriteLine` for the output works in the code above. After all, we know that we can pass a single `string` to the `WriteLine` method, but there are a bunch of `+` symbols in our argument above. It turns out that we can “hook strings together” – the correct term is *concatenate* – using the `+` symbol to form a new string. So in the method call above, we’re concatenating four different strings together (the values of `radius` and `area` are automatically converted to strings in this case) to build the single `string` argument that gets passed to the `WriteLine` method.

When we run the code, we get the output shown in Figure 3.3.



**Figure 3.3. Initial Circle Program Output**

So far, so good! Let's add the rest of the circles; the resulting code is shown in Figure 3.4. Don't worry, we know that all the repeated code is really irritating. We'll learn how to do this much more effectively when we learn about the iteration control structure in a few chapters.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CirclesApplication
{
    /// <summary>
    /// Outputs a variety of circle characteristics
    /// </summary>
    class Program
    {
        /// <summary>
        /// Outputs radius and area for circles with radii 0 through 5
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            int radius;
            float area;

            // circle with radius 0
            radius = 0;
            area = (float)Math.PI * radius * radius;
            Console.WriteLine("Radius: " + radius +
                ", Area: " + area);

            // circle with radius 1
            radius = 1;
            area = (float)Math.PI * radius * radius;
            Console.WriteLine("Radius: " + radius +
                ", Area: " + area);
        }
    }
}
```

```

// circle with radius 2
radius = 2;
area = (float)Math.PI * radius * radius;
Console.WriteLine("Radius: " + radius +
    ", Area: " + area);

// circle with radius 3
radius = 3;
area = (float)Math.PI * radius * radius;
Console.WriteLine("Radius: " + radius +
    ", Area: " + area);

// circle with radius 4
radius = 4;
area = (float)Math.PI * radius * radius;
Console.WriteLine("Radius: " + radius +
    ", Area: " + area);

// circle with radius 5
radius = 5;
area = (float)Math.PI * radius * radius;
Console.WriteLine("Radius: " + radius +
    ", Area: " + area);

Console.WriteLine();
}

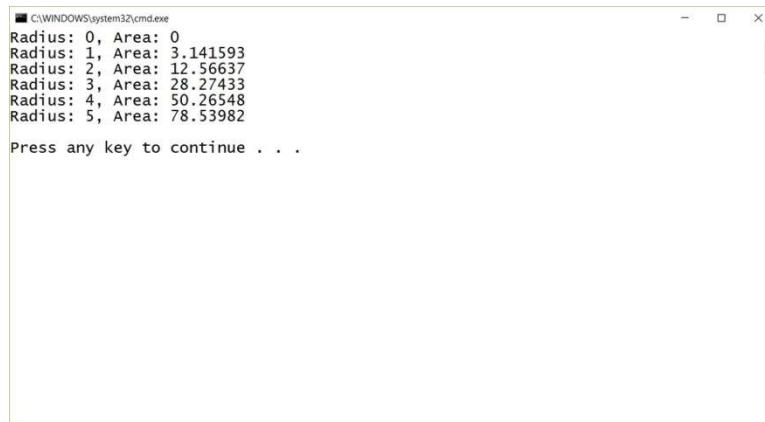
}

}

```

**Figure 3.4. Program.cs***Test the Code*

So now we need to run our test cases to make sure the program works. The actual results from running the code are provided in Figure 3.5.

**Figure 3.5. Test Case 1: Checking Radius and Area Info Results**

Uh Oh. We made a big point of saying that you have to make sure you figure out EXACTLY what your expected results are, but our expected results don't exactly match our actual results. While we could say

that the program works correctly if we ignore the less significant digits in the output, we should really make our expected results more precise or we should make our program only output the area with 2 digits after the decimal point. It's probably reasonable to expect someone to really only be interested in the first couple digits, so let's change our program to provide that output.

Before we do that, you should know that any time we identify problems with our programs and try to find and fix them, this is typically called *debugging*. Why is it called that? Because back when computers actually used switches (called relays) that physically opened and closed, a moth became caught in one of those switches, causing the computer to behave incorrectly. The computer technicians had to literally debug the computer so it would work properly, and the name (like the moth <grin>) stuck. There will be lots of times when we need to debug our programs, so we'll try to demonstrate that process to you as much as possible (within reason!).

The current version of our test case shows most of the areas with 2 decimal points, but the 0 doesn't have a decimal part at all. If we're going to change the behavior of the program to print all of the areas with two decimal places, we need to change the expected results in our test case to include those for 0 as well:

### **Test Case 1: Checking Radius and Area Info**

Step 1. Input: None.

Expected Result:

```
Radius: 0, Area: 0.00
Radius: 1, Area: 3.14
Radius: 2, Area: 12.57
Radius: 3, Area: 28.27
Radius: 4, Area: 50.27
Radius: 5, Area: 78.54
```

Okay, how do we fix our program? All of our work here will be in our calls to `Console.WriteLine`, so we'll just show the steps we use to modify the first call to that method then show the complete (modified) application class again.

The first thing we'll do is change the way we call the method to make formatting the output a little easier. Consider the following code:

```
Console.WriteLine("Radius: {0}, Area: {1}", radius, area);
```

This prints exactly the same output as our previous code. The way this new format works is that we indicate within our string literal the places where we want to substitute other values. For example, when `Console.WriteLine` reaches the `{0}`, it substitutes the first value following the literal; in this case, that's `radius`. Programmers almost always start counting at 0, that's why the first value following the literal is numbered 0. The same substitution idea applies across the rest of the literal as well.

You may be thinking, though, that this new format hasn't really helped us at all, since the output looks exactly the same as it did before. Get ready, though, because we're about to make it all worthwhile! Look at the code below, which changes the `{1}` to `{1:N2}`:

```
Console.WriteLine("Radius: {0}, Area: {1:N2}", radius, area);
```

What does that do? It tells `Console.WriteLine` that the second value (`area`) should be printed as a number (from the `N`) with two decimal places (from the `2`). This is exactly the way we want our output to look, so we just need to make the same changes to all our calls to the `Console.WriteLine` method.

But how do you know what all the formatting possibilities are (and there are lots of them)? Go to the MSDN help and search on Standard Numeric Format Strings. You can find just about any formatting option you might need there.

Okay, so we make the changes above to our code and we get the output shown in Figure 3.6. This output matches the expected results listed in our test case, so strictly speaking, we're done at this point. You might have noticed, however, that it would be more intuitive to have all the area values right-justified (all lined up on the right). Let's make one more change to make those areas right-justified. We do need to point out, though, that unless the problem description explicitly addresses the alignment of the output values – and ours doesn't – this is purely a matter of personal preference.

```
C:\WINDOWS\system32\cmd.exe
Radius: 0, Area: 0.00
Radius: 1, Area: 3.14
Radius: 2, Area: 12.57
Radius: 3, Area: 28.27
Radius: 4, Area: 50.27
Radius: 5, Area: 78.54

Press any key to continue . . .
```

**Figure 3.6. Revised Output**

Remember, though, that our test case as currently written has the areas left justified in the expected results. If we're going to change the way we expect the code to behave, we need to change the test case to reflect the new behavior:

### **Test Case 1: Checking Radius and Area Info**

Step 1. Input: None.

Expected Result:

```
Radius: 0, Area: 0.00
Radius: 1, Area: 3.14
Radius: 2, Area: 12.57
Radius: 3, Area: 28.27
Radius: 4, Area: 50.27
Radius: 5, Area: 78.54
```

Our final change to the first call on the `Console.WriteLine` method is

```
Console.WriteLine("Radius: {0}, Area: {1,5:N2}", radius, area);
```

The new format string for the second (number 1) value following the string literal says to print the value as a number, with 2 decimal places, in a field that's 5 characters wide. This right-justifies values that turn out to be less than 5 characters wide. Note that we typically figure out what the largest possible value

we'll be outputting will be so we pick a large enough field width to make sure everything is right-justified.

The final application class code is shown in Figure 3.7., with the final output shown in Figure 3.8.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CirclesApplication
{
    /// <summary>
    /// Outputs a variety of circle characteristics
    /// </summary>
    class Program
    {
        /// <summary>
        /// Outputs radius and area for circles with radii 0 through 5
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            int radius;
            float area;

            // circle with radius 0
            radius = 0;
            area = (float)Math.PI * radius * radius;
            Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
                radius, area);

            // circle with radius 1
            radius = 1;
            area = (float)Math.PI * radius * radius;
            Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
                radius, area);

            // circle with radius 2
            radius = 2;
            area = (float)Math.PI * radius * radius;
            Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
                radius, area);

            // circle with radius 3
            radius = 3;
            area = (float)Math.PI * radius * radius;
            Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
                radius, area);

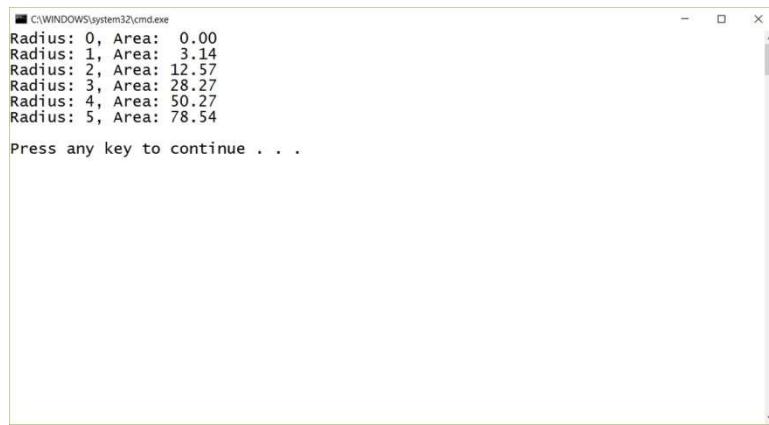
            // circle with radius 4
            radius = 4;
            area = (float)Math.PI * radius * radius;
            Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
                radius, area);
        }
    }
}
```

```

    // circle with radius 5
    radius = 5;
    area = (float)Math.PI * radius * radius;
    Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
        radius, area);

    Console.WriteLine();
}
}
}

```

**Figure 3.7. Final Application Class****Figure 3.8. Test Case 1: Checking Radius and Area Info Results**

### 3.14. Putting It All Together in a Unity Script

Let's solve essentially the same problem we solved in the previous section in Unity using the same problem description:

Calculate the area of the circles with integer radii from 0 to 5, then print out the radius and area for each circle.

#### *Understand the Problem*

This is the same problem description, so we should still understand it. We will, of course, provide our output in the Console window in the Unity editor.

#### *Design a Solution*

For our Unity solution, we'll write a new `PrintCircleInformation` script and implement the required functionality in the `Start` method in that script. As in the previous chapter, we'll need to add our script as a component of the Main Camera to make it run when we run our game.

#### *Write Test Cases*

We'll use the same (final) test case we used in the previous section:

## Test Case 1: Checking Radius and Area Info

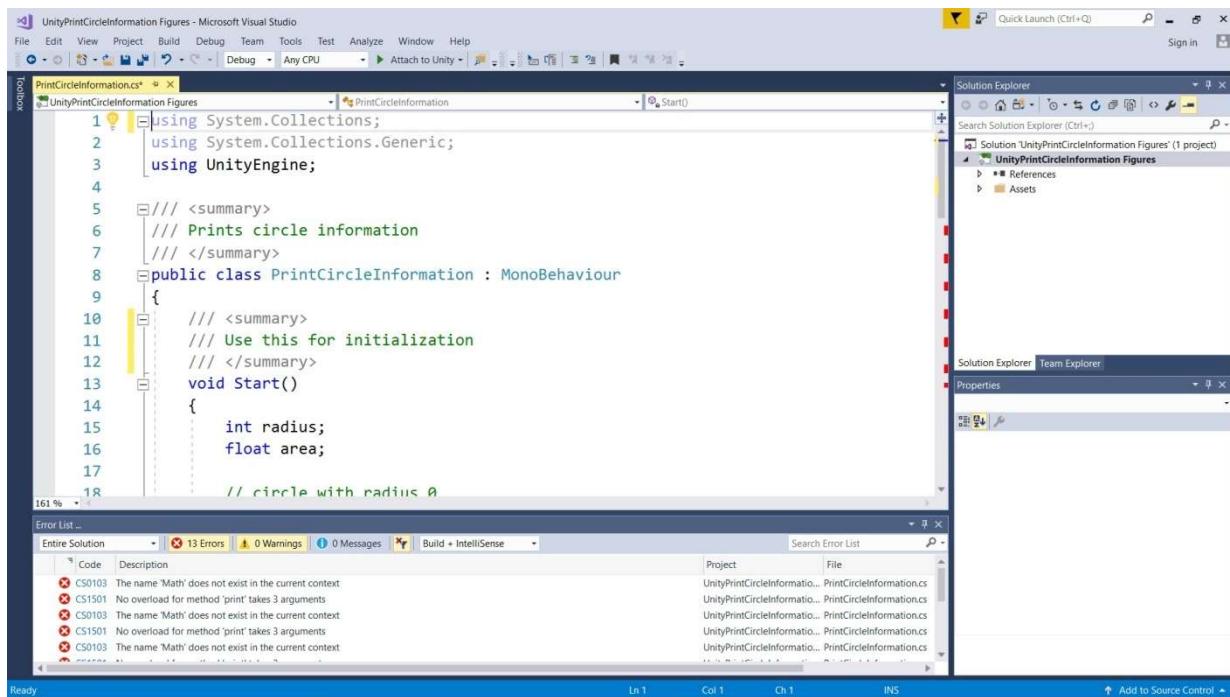
Step 1. Input: None.

Expected Result:

```
Radius: 0, Area: 0.00
Radius: 1, Area: 3.14
Radius: 2, Area: 12.57
Radius: 3, Area: 28.27
Radius: 4, Area: 50.27
Radius: 5, Area: 78.54
```

*Write the Code*

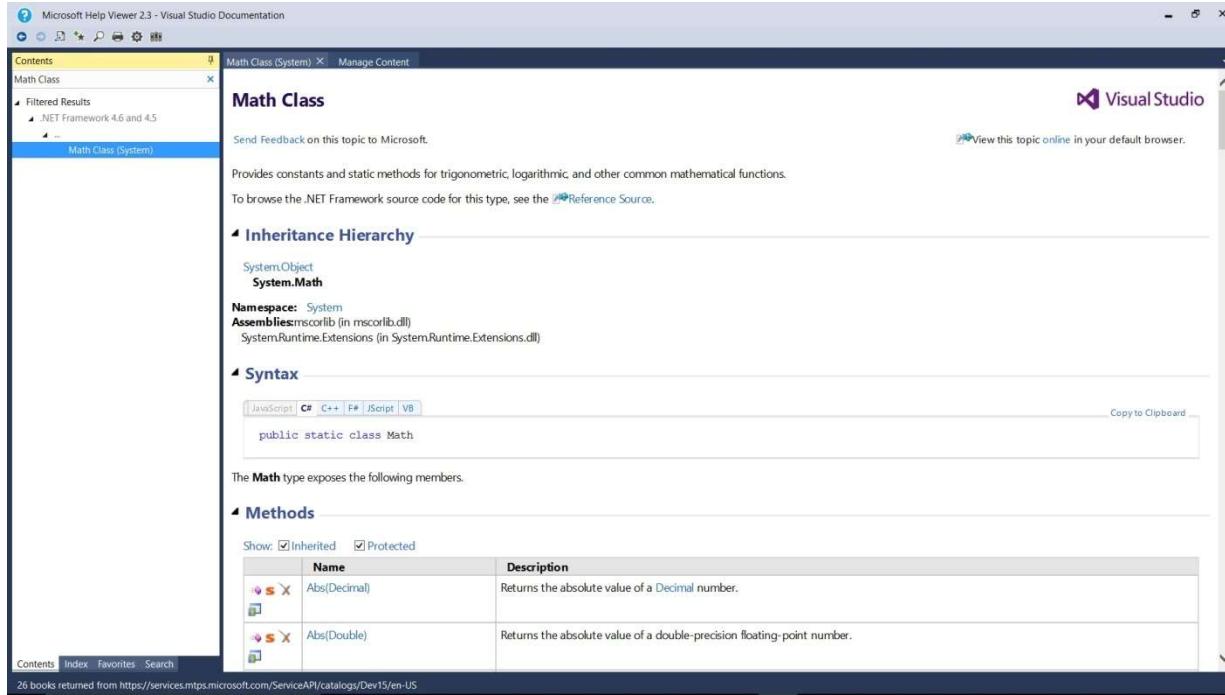
As we did in the previous chapter, we'll copy the code from the `Main` method in our console app into the `Start` method in our new script and change all the calls to the `Console.WriteLine` method to calls to the `print` method instead. Unfortunately, when we try to compile we get the errors shown in Figure 3.9.



**Figure 3.9. Initial Unity Script Compilation Errors**

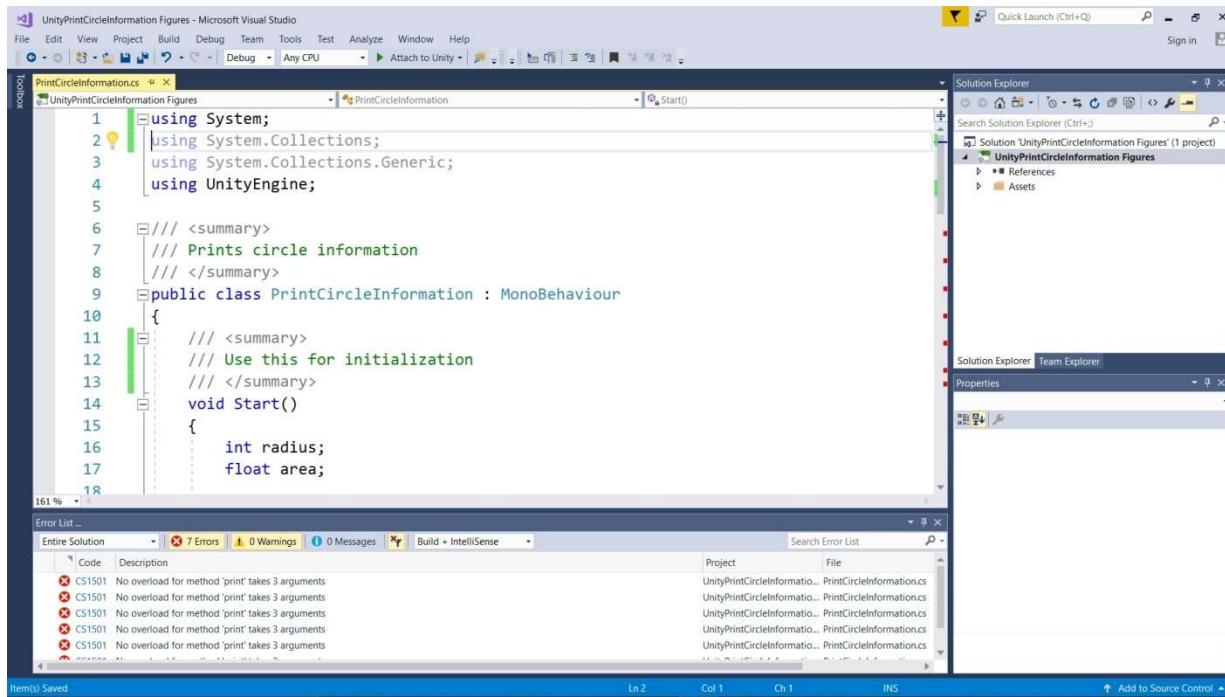
Usually, the best approach to take when you have lots of compilation errors is to fix the first error, then recompile to get a new (hopefully empty!) list of errors. Remember from the previous chapter that an error message that includes "does not exist in the current context" often means that you're missing a `using` directive. To fix our first error, we have to figure out what namespace the `Math` class is in so we can add the required namespace.

How do we figure that out? The documentation, of course. If you go to help, search on "Math class", and select the search result for Math Class (System) you get the page shown in Figure 3.10. If you look on that page in the Inheritance Hierarchy section, you'll see a line that says **Namespace:** `System`. That tells us that the `Math` class is in the `System` namespace, so we need to add a `using` directive for that namespace at the top of our script.



**Figure 3.10. Math Class Documentation**

Once we've added the new using directive and recompiled, we get the errors shown in Figure 3.11.



**Figure 3.11. Revised Unity Script Compilation Errors**

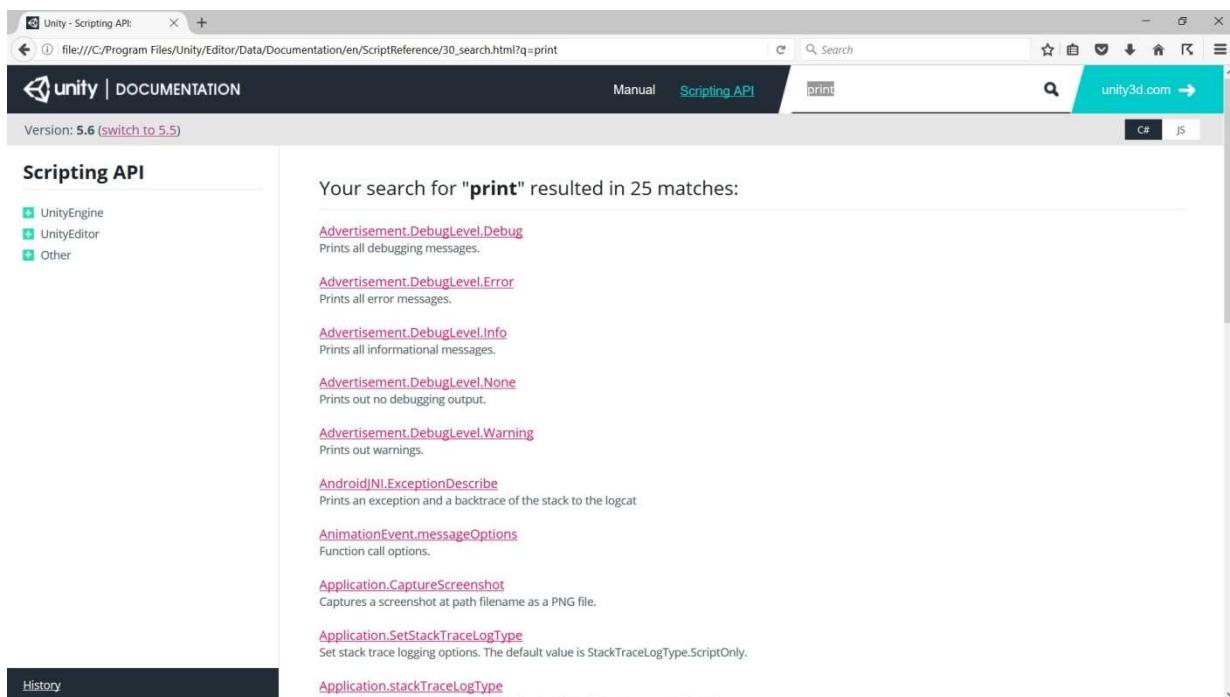
You may also see this kind of error a lot in your programming career. We'll talk about what an overload is a little later in the book, but basically this error message says that you're trying to call the `print` method with 3 arguments, but you're not allowed to provide 3 arguments to that method.

Okay, now what? We can't check the Visual Studio help or the MSDN documentation because the `print` method is provided in Unity, not in standard C#. The good news is that we have a Unity Scripting Reference available to us as well, we just have to get to it.

We can access the Unity Scripting Reference by selecting Help > Unity API Reference from the Unity editor menu bar, but it's even more convenient to access that documentation from within Visual Studio by selecting Help > Unity API Reference.

The default is that the documentation will open up in a tab in Visual Studio, but we actually prefer opening it up in a browser. To set that up, select Tools > Options from the top menu bar. Expand the Tools for Unity area in the pane on the left and select General. In the pane on the right, change the Use external browser value in the Documentation area to true. Click the OK button near the lower right of the dialog.

Select Help > Unity API Reference from the top menu bar and search on `print`; that brings us to the page shown in Figure 3.12.

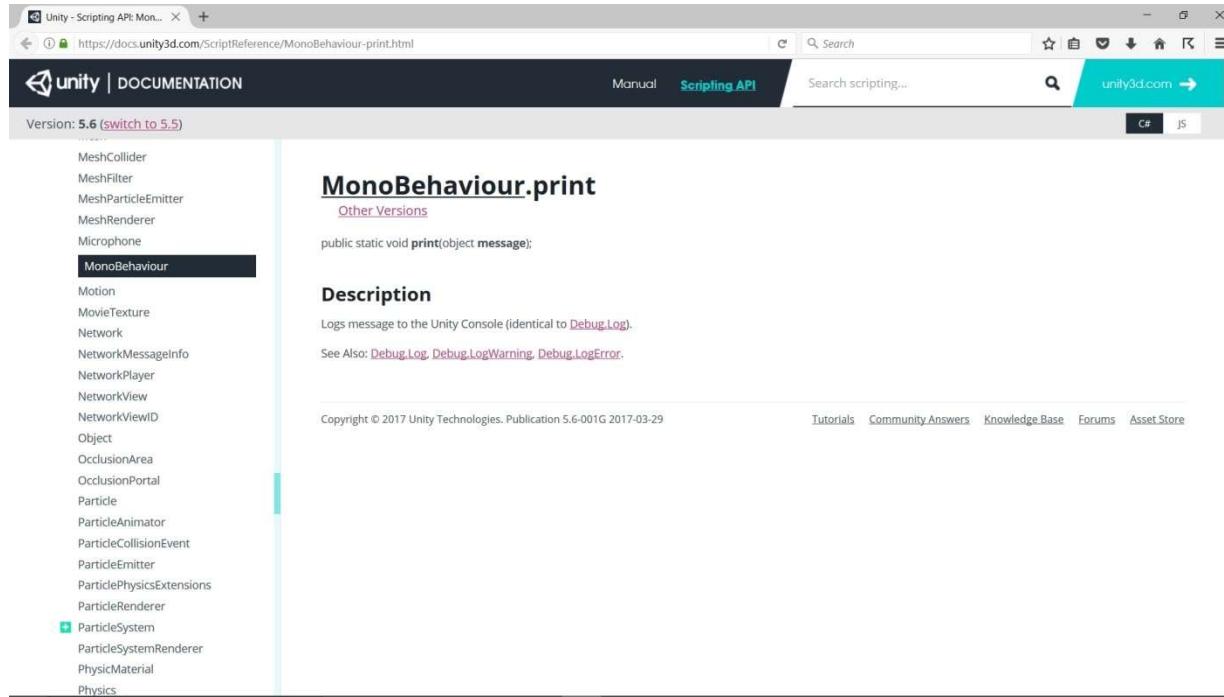


**Figure 3.12. Unity API Reference Help Page**

Unfortunately, in this particular example, none of the search results actually help us, even if we scroll down through all of them. The bad news here is that we have to know which class contains the `print` method to get to the documentation for the method, and we don't know which class to look at<sup>11</sup>. What now?

<sup>11</sup> Of course, once we learn about inheritance we'll realize that our `PrintCircleInformation` script inherits from `MonoBehaviour`, which in fact contains the `print` method. We haven't learned about inheritance yet, though, so we need to find the `print` method documentation a different way here.

You'll also discover that all programmers – beginning through professional – use this World Wide Web thing all the time to solve programming problems. Go to your favorite search engine and search on Unity "print method documentation". For Google, Yahoo, and Bing, the top search result leads you to the page shown in Figure 3.13.



**Figure 3.13. Unity Scripting Reference Page**

If you look at the second line in the pane on the right and look between the open and close parentheses just before the semicolon, you'll see that we're only allowed to pass one argument to the `print` method: the message we want to be displayed.

This may have seemed like a really long path to follow to say "we can't format the output the way we did in our console app", but it's important to learn how to navigate the documentation to discover precisely how to do the things we want – or even to discover we can't do them at all!

For this example, we'll just change our test case so that the numbers that are output match the non-formatted float output provided by the `print` method. Don't worry, you'll be able to do lots of awesome stuff in Unity, and let's face it, players don't actually see the Console window in the Unity editor when they're playing games that were developed with Unity, so this `print` method limitation really isn't important in the big scheme of things.

### *Write Test Cases, Revisited*

Here's our revised test:

## Test Case 1: Checking Radius and Area Info

Step 1. Input: None.

Expected Result:

```
Radius: 0, Area: 0
Radius: 1, Area: 3.141593
Radius: 2, Area: 12.56637
Radius: 3, Area: 28.27433
Radius: 4, Area: 50.26548
Radius: 5, Area: 78.53982
```

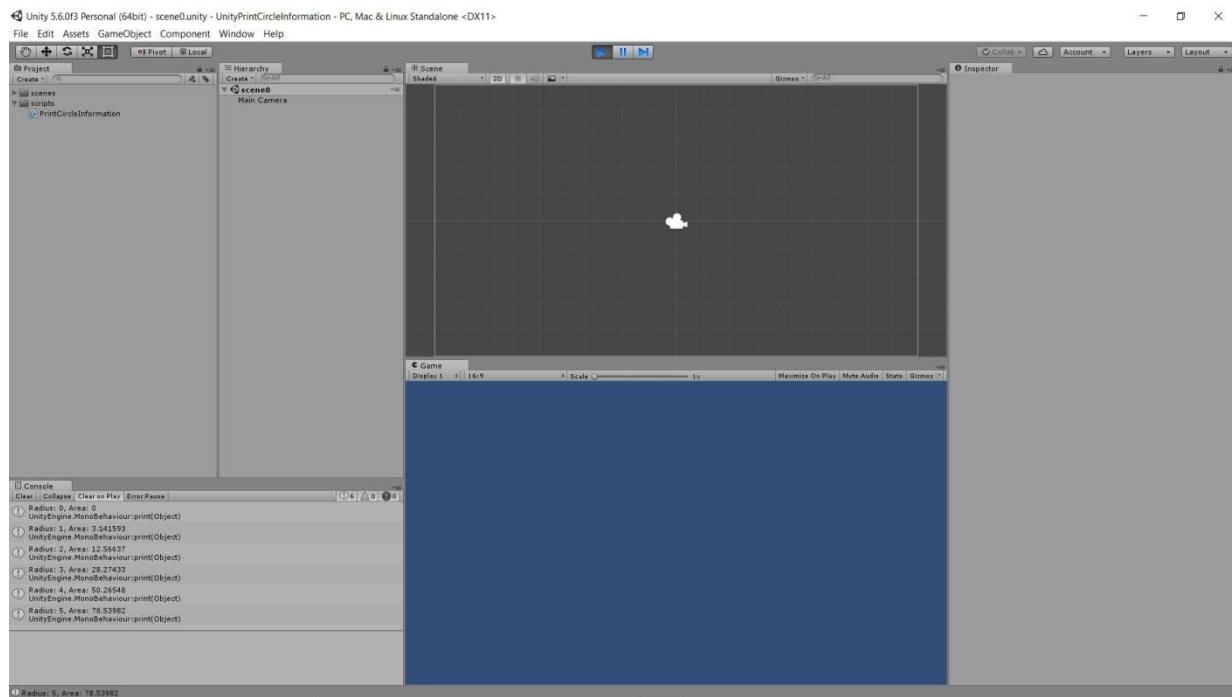
*Write the Code, Revisited*

Next, we change our calls to the `print` method back to our original approach using concatenation instead of formatting. We also got rid of the final call to the `print` method, where we didn't provide any arguments, because the `print` method requires that we pass exactly one argument.

Remember that we actually need to add our script as a component to the Main Camera to make it run when we play the game. Although this step isn't strictly "writing code", it is setting things up so the code executes properly so we include it here.

*Test the Code*

Now we run the game in Unity and get the results shown in Figure 3.14. As you can see, our actual results match our expected results, so we're done with this problem.



**Figure 3.14. Test Case 1: Checking Radius and Area Info Results**

### 3.15. Common Mistakes

#### *Forgetting that C# is Case Sensitive*

This may be the most common error of all for beginning C# programmers. Remember that C# is case sensitive, so a variable named `firstInitial` is NOT the same as a variable named `firstinitial`. If you declare `firstInitial` as your variable and then try to use `firstinitial` in your program, you'll get an error.

#### *Forgetting How Integer Division Works*

This seems to be one of those things that you understand when you read it, then forget about when you actually write your programs! Remember, when you divide two integers, you get the integer quotient, not a floating point result from the division. This is actually very helpful in a lot of cases, which is why C# does it this way in the first place, but you need to remember that's how it works. If you do need a floating point result from dividing two integers, you can simply use type casting as described above.

## Chapter 4. Classes and Objects

When we develop problem solutions using the *object-oriented paradigm*, we design a set of *classes* and *objects* to represent the "things" in our system. Each of those "things" will have *state* (characteristics, like height and weight for a person), *behavior* (stuff we can have those things do, like feed themselves, measure themselves, wash themselves, etc.), and *identity* (ways to tell one thing from another). This chapter introduces you to the key ideas behind classes and objects, and we'll start working with classes and objects as well. For the next few chapters, we'll provide the classes for you to use and, in some cases, modify. But don't worry – you'll be designing your own classes before you know it!

This whole object-oriented thing is particularly well suited to game development. Think about it – basically, games are about modeling virtual worlds populated by lots of entities and then simulating how all those entities behave following the rules of the game world. It doesn't matter if we're talking about Paladins, magic swords, a TVR Tuscan you're about to slam into a wall, or any number of "things" in the game world. They all have state, behavior, and identity. Let's get started.

From here until Chapter 12, we'll spend lots of time talking about how to use classes and objects and even how to implement pieces of classes, but we won't actually discuss how to design and implement our own classes in detail until Chapter 12. Some beginning programmers are frustrated with that approach, but Computer Science educators have been debating for decades whether learning to use classes and objects before trying to design and implement them leads to better or worse learning. Since there's no clear winner in that debate, we follow the "learn to use them first" philosophy in this book. If you don't like that approach and want to jump ahead to Chapter 12, that's the beauty of having complete control over how you read the book!

### 4.1. Introduction to Classes and Objects

The main idea behind the object-oriented paradigm is that our problem solutions consist of a set of collaborating objects, which tend to reflect all the objects we see around us in the real world. Things like furniture, cars, bank accounts, students, and bicycles are all objects. Even though most well adjusted carbon-based life forms understand that game worlds aren't actually "real" (ignore senators and members of Mothers Against Having Fun for the moment), the same exact ideas apply to virtual worlds as well. So when we talk about an object in our design, what is it? As mentioned above, it's something that has state, behavior and identity. Let's look at each of those things individually.

The state of an object indicates what the object "looks like" at any given time. For example, if we had an object for a student in a particular course, its state might include all their grades up to this point, their average, and their standing in the course. If we had a (virtual or real) Lotus Elise, its state might include its horsepower, speed, its current location, how many wheels (if any <grin>) are in contact with the road at the moment, and so on. For an object for a book, its state might include how many pages are in the book, the contents of each page, whether the book is open or closed, and what page it's opened to (if it's open). A playing card object's state would probably have the rank of the card (Ace, for example), the suit of the card, and whether it's face up or not. So the state of the object indicates information about characteristics of the object. We'll call the things we use to keep track of an object's state that object's *fields*.

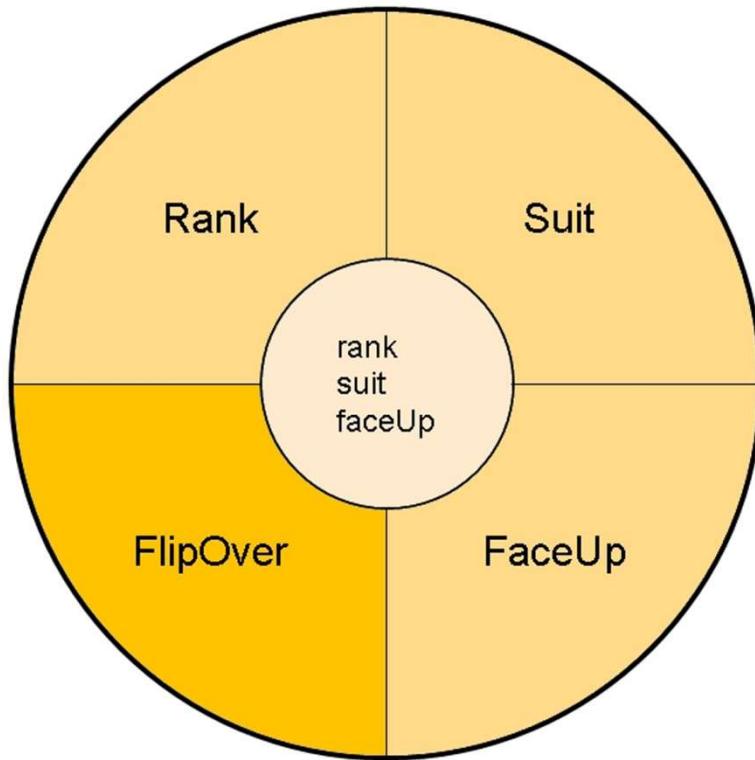
Although there are some cases where keeping an object's state hidden makes sense, there are also lots of times where something outside the object needs to know its state. For the Lotus Elise, for example, we

need to know its location (and other information) to check if it's colliding with something in our game. Because it's so common for us to need to access an object's state, C# provides an easy way to do that – through *properties*. For the playing card example, we're probably going to need to know the rank and suit of the card and whether or not it's face up; we'll get access to that state information through properties.

The second part of an object is its behavior. We don't mean whether the object gets punished a lot or not (ha ha); we mean the actions the object can take. What can we ask it to do, or what can we do to it? For the book, for example, we want to be able to open or close the book, to change the page in the book, and to look at the current page. For the card object, we'll want to be able to flip the card over, so we'd need the card object to do that for us. We'll call the things we use to specify an object's behavior that object's *methods*.

The final part of an object is its identity. We can have lots of cards, right? How do we tell one card from another? Well, each card has a unique identity; that's how we tell them apart. We'll discuss this more in a few pages.

So really, what does an object look like? It has some fields, which keep track of important characteristics of the object (its state); it has some properties that provide access to the object's state; and it has some methods, which let us have the object complete some action. Visually, you can think of our card object as looking like this, with the fields in the middle (lightest color) and the properties (medium color) and methods (darkest color) forming a ring around them:



**Figure 4.1. A Card Object**

One of the most important ideas here is that we don't get to see “inside” the object to use it – but we don't have to either! All we do is use the object's properties and methods to get it to do what we need it

to, without worrying about what fields are inside the object or even how the methods work. To actually use a method, we call or invoke the desired method.

These ideas lead us to a couple important object-oriented terms. *Encapsulation* is the process of combining related fields and behaviors into a single object, essentially placing them all in a single "capsule." *Information hiding* means that the fields of an object are "hidden behind the wall of properties and methods" for the object, like in the picture above. In addition, the implementation details of the properties and methods are also hidden. In other words, an object tells us what its current state is and what it can do through its properties and methods without telling us how it does it using its internal fields and the code in those properties and methods.

Information hiding is a very powerful technique because we can use the object without having to know how it works internally; in other words, we can just view the object as a black box. Why is this helpful? Because if we end up changing the internals of the objects later on, we don't break all the code that uses them.

At the beginning of this section, we talked about classes and objects. We've already figured out objects, but what's this class thing? Well, it turns out that we might have lots of objects that look exactly the same (there are lots of playing cards in the world, right?). In those cases, we build a template or blueprint for what all the objects will look like. And what's that template called? You guessed it – the class. The class describes the fields, properties, and methods for objects in the class, then whenever we need another one of those objects we simply use the class as the template. In object-oriented lingo, we create a new object (instance) from the class using *instantiation*.

We're clearly going to need some kind of notation if we want to show the structure of the classes and objects we design for our problem solutions. We'll use something called the Unified Modeling Language, or UML. There have been lots of object-oriented notations proposed, so we have plenty of choices, but UML is easy to understand, easy to draw, and is really the most commonly used notation.

Let's take a look at a UML representation of our playing card class<sup>12</sup>:

---

<sup>12</sup> You can use lots of different tools to generate UML diagrams. For the diagrams in this book, we used a copy of Visual Studio Community 2015.



**Figure 4.2. UML for Card Class**

In the top section, we have the name of the class (we decided to call it `Card` for obvious reasons). We'll capitalize the first letter of our class names and start with a lower case letter for the names of our objects so that it's easy to tell from the name of something whether we're talking about a class or an object.

The second section in our diagram lists all the fields of the class. We'll have to keep track of the rank of the card, the suit of the card, and whether the card is face up or not, so we list all of these as variables. Note that we start each field name with a lower case letter. Each variable is followed by a data type (either a value type or a reference type – a class). When we actually declare our variables in C# the data type comes before the variable name, but in UML the order is reversed. Although we recognize that this could be a little confusing at first, we want to stick with the basic rules of UML even if we ignore lots of advanced UML features.

The third section in our diagram lists all the properties for the class<sup>13</sup>. The properties for our card class let us get the rank of the card, get the suit of the card, and determine whether the card is face up or not. Note that we start each property name with an upper case letter.

Finally, the last section in our diagram lists the methods for the class. In this case, we only have one method that flips the card over.

By the way, in C# the fields, properties, and methods of a class are called the class *members*.

**Important Unity Note:** There are lots of classes in Unity that actually do expose the fields of objects rather than using properties to control access to those fields. We'll obviously access those fields when

---

<sup>13</sup>UML actually calls properties and methods operations, but we'll use our C#-specific terminology instead.

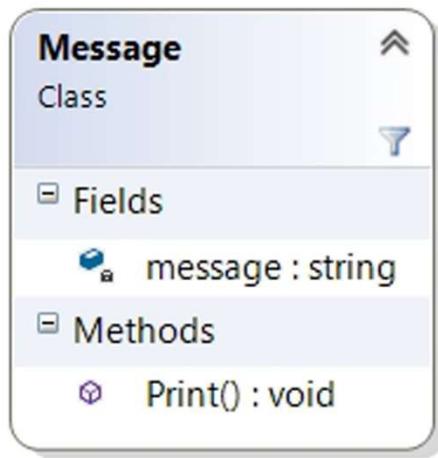
we need them without feeling any guilt whatsoever, but when we design our own Unity scripts (which are classes) we'll use properties whenever we can.

It's also important to realize that the Unity documentation uses the term *functions* rather than methods, but you can think of them as being the same thing – implementation of an object's behavior.

## 4.2. Your First C# Program Revisited

Remember how we wrote an application class that simply printed our message out to the screen? Well, there's definitely a more object-oriented approach to solving that problem! Let's use a `Message` class for the message; then we can create an object of that class and have the object print itself. That does a number of things for us. It gets us used to thinking about things in an object-oriented way, and it also lets us start working with objects. Sounds good, right? You should note that we wrote the `Message` class for this example; it isn't one of the classes provided by C#.

The first thing we need is the UML for the `Message` class; that's provided in Figure 4.3.



**Figure 4.3. UML for Message Class**

You should notice that for this class, we don't have any properties at all. That's because the only time the message field is needed is when we're actually printing the message to the console. The `Print` method definitely needs access to the field, but as a member of the `Message` class the method automatically has access to all the other members in the class. That means that there's no need to "expose" the message field; instead, we'll keep that information completely hidden inside the class.

Because we're only interested in using the class, we're not worried about how the class works internally. Instead, we'll look at the documentation that's generated by Sandcastle for the members of the `Message` class; that's shown in Figure 4.4.

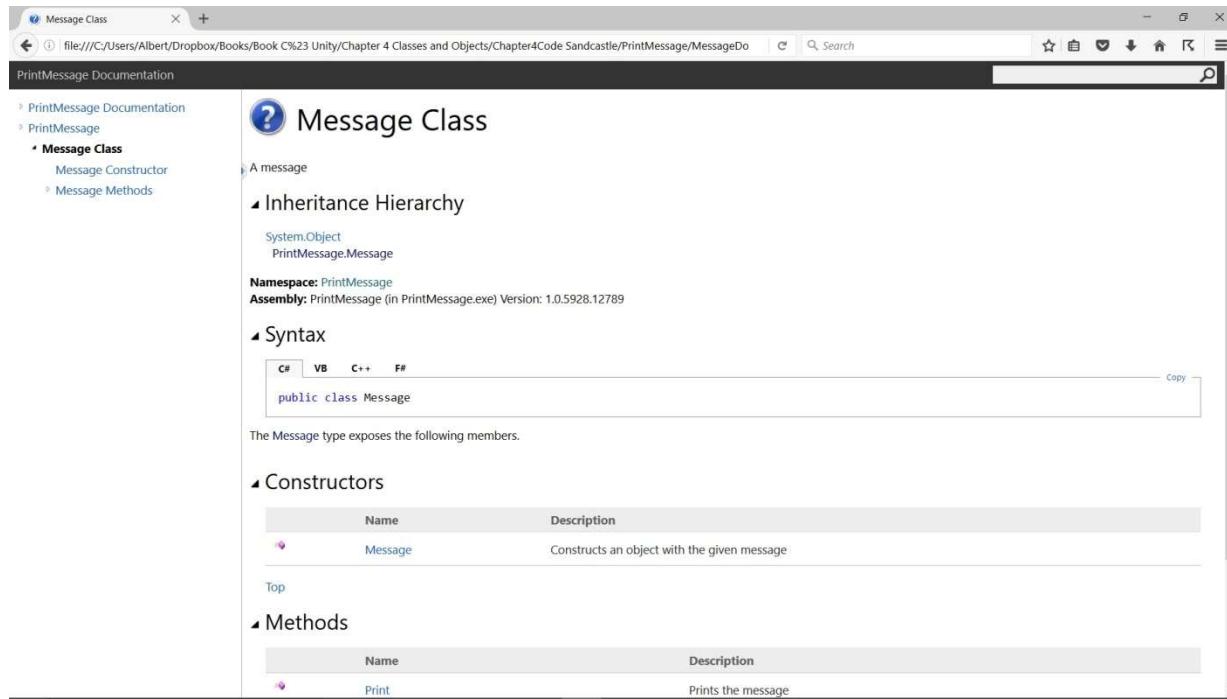


Figure 4.4. Message Class Documentation

Although we don't need to look at the code implementing the `Message` class to figure out how to use the class, we will need that code so we can actually compile our application class that uses the class. A Visual Studio project containing a template application class and the `Message` class is available from the web site for the book.

Based on the documentation, we'll be able to do two important things with the `Message` class: we'll be able to create new objects of that class using the *constructor*, and we'll be able to have those objects print their messages to the screen. First, let's talk about creating a message object. The general syntax for creating objects is provided below.

### SYNTAX: Creating an Object

```
ClassName objectName = new ClassName(arguments);
```

`ClassName`, the name of the class for the object

`objectName`, the name of the object being created

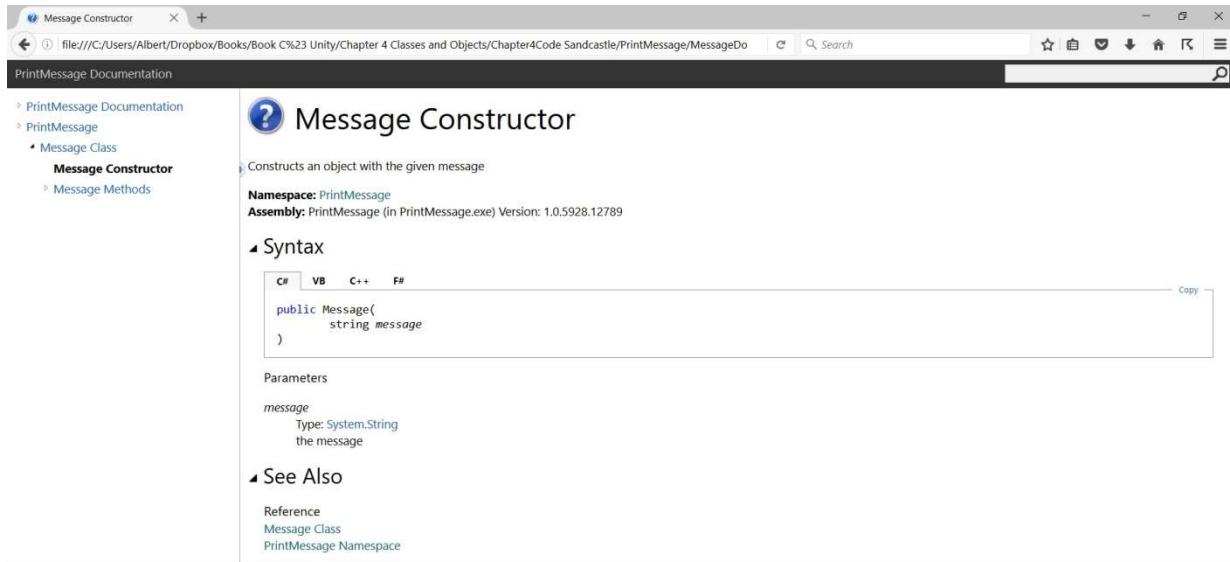
`arguments`, the arguments for the constructor (there may not be any)

That means that, to create a tiny message that says Hi!, we'd use:

```
Message hiMessage = new Message("Hi!");
```

When this statement gets executed as you run your program, `hiMessage` is created as a new object. The message contents, which are called `message` internal to the `Message` class, are set to the string `"Hi!"` when the object is built.

How do we know that the constructor requires one argument for the message string? By reading the documentation, of course. If we click on the [Message](#) constructor link in the documentation shown in Figure 4.4, we get the documentation shown below.



**Figure 4.5. Message Constructor Documentation**

You can see at the top of the documentation that the constructor has a single `string` parameter. At the bottom of the screen shot, you can see that the parameter is the message that we want the object to hold.

It's important to note that all we really care about in the documentation above is that the parameter for the constructor is a `string`, so we'll have to pass a single `string` as an argument when we call the constructor. We don't care at all that the programmer who wrote the constructor decided to name the parameter `message` because that's an internal detail of the constructor that doesn't affect how we call it.

Now, creating a `message` object for our rain message is marginally more complicated because our message goes over several lines. Here's one way we could do it:

```
Message rainMessage = new Message("Hello, world\n" +
    "Chinese Democracy is done and it's November\n" +
    "Is it raining?");
```

Notice that we used `+` to concatenate the three parts of our message string together; we only did that because we couldn't fit the entire string on one line of code. The more interesting part of the message string we use is the `\n` after `world` and `November`. The `\n` is known as an *escape sequence*, because it consists of an *escape character*, which is the backslash (`\`) in C#, followed by another character. When we output a string containing an escape sequence, a special character is output instead of the escape sequence. Some of the more common escape sequences and their meanings are provided below.

\n

Newline. The cursor is moved to the next line on the screen

\t

Tab. The cursor moves to the next tab stop on the screen

\r

Carriage return. Moves the cursor back to the beginning of the current line

\\\

Backslash. Prints a backslash. We can't just include a single backslash, because a single backslash is interpreted as the escape character

\"

Double quote. Prints a double quote. We can't just include a double quote in our string, because a double quote is interpreted as the end of the string

So in our case, including the \n puts a newline character into our message, which will make the output appear on three separate lines.

Now that we have our rainMessage object, we can tell it to print itself whenever we want to by calling the Print method for the object. We do that using the following code:

```
rainMessage.Print();
```

We don't have to pass any arguments in to the method, because the string for the message we're going to print is already contained in the object itself; that's why we provided the message string to the constructor when we called it. To call this kind of method for an object, we start with the object's name; this tells C# which object it should use. We then put a period, followed by the method name. This tells C# which method in the given object to use. Finally, we put any arguments the method needs between open and close parentheses. Because the Print method doesn't need any arguments, we don't put anything in the parentheses (but we still need to include the parentheses!). We'll discuss calling the different kinds of methods in the following section.

There's an important point we need to make here. Although we showed one possible syntax for creating a message object, we could actually break that process into two steps. In the first step, we'd *declare* the variable using

```
Message rainMessage;
```

and then, sometime later on, we'd actually *create* the object using

```
rainMessage = new Message("Hello, world\n" +
    "Chinese Democracy is done and it's November\n" +
    "Is it raining?");
```

The object doesn't actually exist in memory (because it hasn't been created yet) until you've done the second step. Objects that have been declared as variables but haven't been created yet are initialized to a special value, null. We can't use any of the object's methods or do anything else particularly useful with

the object until we create it. If we don't create our object at the same time we declare it, we might try something like:

```
Message rainMessage;
rainMessage.Print();
```

Luckily, if you try this the compiler will complain to you that `rainMessage` was never initialized. Even if this kind of thing gets by the compiler, your program will "blow up" if you try to call a method for a null object. In general, it's a good idea to create objects using the syntax we provided in the syntax description above. We'll definitely find there are many times when we want or need to use the other approach, but for now we'll create objects at the same time we declare them.

So now we have all the tools we need to rewrite our application class using our new `Message` class. Basically, our application class will now create a message object with our rain message, then tell that object to print itself. The application class code is in Figure 4.6 (if you download the project from the web site, you can just add the comments and a few lines of code to match the code below).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrintMessage
{
    /// <summary>
    /// Prints a message to the console
    /// </summary>
    class Program
    {
        /// <summary>
        /// Prints the message
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            Message rainMessage = new Message("Hello, world\n" +
                "Chinese Democracy is done and it's November\n" +
                "Is it raining?");
            rainMessage.Print();
            Console.WriteLine();
        }
    }
}
```

**Figure 4.6. Application Class**

So now we have a true object-oriented program to print our message. But you could certainly be thinking to yourself that we've just replaced a tiny application class with an application class that's about the same size AND the C# code for a new class. And you're right – for this simple example, we didn't really save space. The point, though, is that you've gotten your first close look at using a different class from your application class.

### 4.3. Calling Methods

Let's take a closer look at how we make methods run as the program executes. We do this by *calling* (also called *invoking*) the method when we want it to run. This is nothing new; we've been calling methods since Chapter 1! We just wanted to talk in a little greater detail how this actually works.

There are 4 possible kinds of method calls; the syntax for all of them is provided below.

---

#### SYNTAX: Calling Methods

##### Method with No Return Value, No Parameters

```
objectName.MethodName();
```

*objectName*, the name of the object

*MethodName*, the name of the method we're calling

##### Method with Return Value, No Parameters

```
variableName = objectName.MethodName();
```

*variableName*, the name of the variable to hold the returned value

*objectName*, the name of the object

*MethodName*, the name of the method we're calling

##### Method with No Return Value, With Parameters

```
objectName.MethodName(argument, argument, ...);
```

*objectName*, the name of the object

*MethodName*, the name of the method we're calling

*argument*, one argument for each parameter

##### Method with Return Value, With Parameters

```
variableName = objectName.MethodName(argument, argument, ...);
```

*variableName*, the name of the variable to hold the returned value

*objectName*, the name of the object

*MethodName*, the name of the method we're calling

*argument*, one argument for each parameter

---

Remember the distinction we make between parameters and arguments. Parameters are the things included in the method header; at this point, we learn about a method's parameters by reading the documentation for the method. Arguments are the things we include in a method call to "match up with" the parameters for the method we're calling.

Let's look at a method call that doesn't return a value and doesn't have any arguments. Here's an example for telling a deck of cards to shuffle itself:

```
deck.Shuffle();
```

All we need to do is provide the object name followed by the method name with open and close parentheses. We already used a method call like this when we called the `Print` method for our `rainMessage`.

As we've said, calling a method makes the code in that method run. Say we're executing a program and we reach the method call above. What happens? The program actually goes to that method, executes the code in the method body, then returns to the next line of code in our original program. Of course, methods can call other methods, so we can go through a number of method calls (and code executions) before returning to our original code.

What about calling a method that returns a value but still doesn't have any parameters? Here's one example for getting the top card from a deck of cards:

```
card = deck.TakeTopCard();
```

So when we call a method that returns a value, we need to put a variable name (to hold the value the method returns), the `=` sign, and the name of the method followed by open and close parentheses.

It actually turns out that we could call a method that returns a value in the following way as well:

```
deck.TakeTopCard();
```

Although this can be useful in some cases, the `TakeTopCard` method returns a `Card` object for a reason. The code above just disposes of that returned `card`, which probably isn't what we really want to do. In general, if you call a method that returns a value you should save and use that returned value in some way rather than just throwing it away.

The third kind of method call has arguments but the method we're calling doesn't return a value. An example for calling a method to cut a deck of cards at a particular location is as follows:

```
deck.Cut(26);
```

In this method call, we say we want the deck to be cut at location 26, so we provide 26 as the argument. It's only slightly more complicated calling methods that have parameters, because for each parameter in the method header, we need to provide an argument in the method call.

There are, of course, some rules we need to follow for arguments and parameters. If the method we're calling has one parameter, we can't call that method with no arguments, two arguments, etc. – we have to call that method with exactly one argument. In other words, the *number* of parameters in the method header and arguments in the method call have to match. For each argument in the method call, the *data type* of the argument has to match the data type of the parameter (actually, they only have to be compatible, but for our purposes we'll say they have to be the same). For example, the `location` parameter in the `Cut` method is declared as an `int`, so the argument in the method call also needs to be an `int`.

The last thing we need to worry about is the *order* of our parameters and arguments. If a method has multiple parameters, we need to provide them in the correct order. C# uses the order in which we've listed our arguments in the method call to match up the parameters and arguments. Because this can lead to hard-to-find errors for beginning (and experienced) programmers, be VERY careful to put your arguments in the same order as the parameters for the method you're calling.

One last comment about arguments. If the parameter is a value type, we can either use a variable for the argument or a literal for the argument. In other words, we could have also called the `Cut` method using

```
int cutLocation = 26;
deck.Cut(cutLocation);
```

Okay, the last method call we need to worry about is one where the method call requires arguments (because the method has parameters) and the method returns a value. For example:

```
defLocation = bandName.IndexOf("Def");
```

We provide the string we want to look for in the `bandName` as an argument (`"Def"`) and store the result in the `defLocation` variable. We'll learn lots more about strings in Chapter 5.

## 4.4. Reference Types in Memory

In the previous chapter, we discussed value types and how they're stored in memory. We deferred our reference type discussion until this chapter, but now you know enough about classes and objects to explore reference types in more detail.

Reference types are called that because variables that are reference types refer to an object that's been created in memory. The data type for an object is a class (these are also called *reference data types* in C#).

What do we mean when we say that variables refer to an object in memory? Recall that for the value types discussed previously, the memory location for the variable holds the value of the variable. In contrast, the value of a reference type variable is the memory address of the actual object in memory. Perhaps a picture will help; in Figure 4.7, the `age` variable is an `int` and the `rainMessage` variable is a `Message`. The `age` variable holds its actual value 17, while the `rainMessage` value holds the memory address of (reference to) the actual `Message` object. Of course, both memory locations actually hold 1s and 0s, but hopefully you get the idea. You'll notice that the `Message` object starts at memory location 44 but takes up more than one memory location. Objects typically take up multiple memory locations, so the variables that refer to them always refer to the start of the memory block (set of locations) that holds the object.

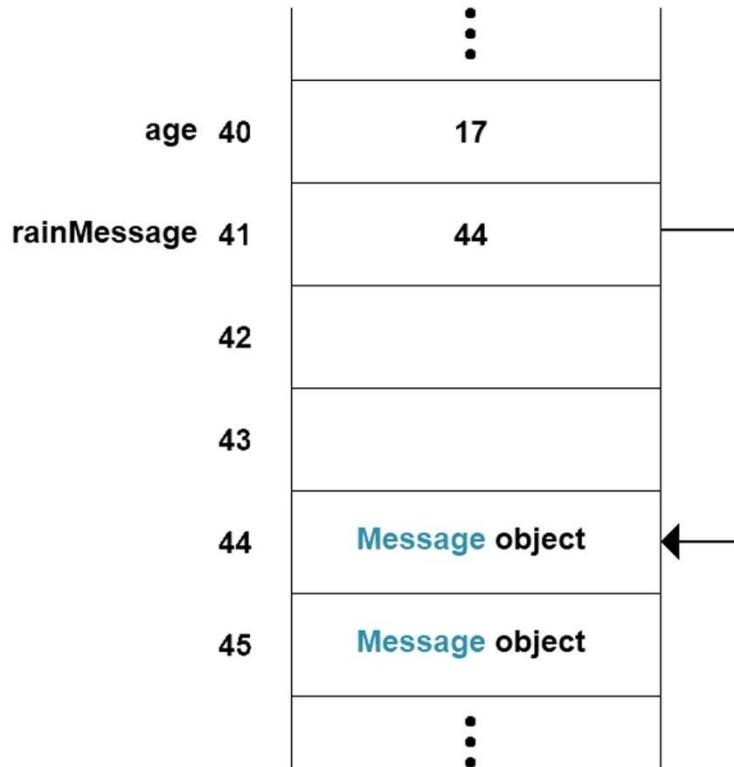


Figure 4.7. Value and Reference Type Example

Let's say we have a variable that's declared as

```
Message rainMessage;
```

What actually gets stored in the memory location set aside for the `rainMessage` variable? It's the special value called `null` that we discussed above. Why does it work this way? Because the variable is supposed to refer to a `Message` object in memory<sup>14</sup>, but we haven't used the `Message` constructor to create the actual object in memory yet. Rather than having an actual value in the variable, which might lead us to mistakenly access memory that doesn't actually hold a `Message` object yet, we get the special `null` value instead.

When it's time to create the `Message` object, we use the constructor as in the following example:

```
rainMessage = new Message("Hello, world\n" +
    "Chinese Democracy is done and it's November\n" +
    "Is it raining?");
```

This is the point at which the computer actually sets aside a block of memory for the object, initializes the object using the code in the constructor, and sets the value of the variable to the memory address of the start of the object's memory block.

---

<sup>14</sup> You could also say point to the `Message` object in memory if you're used to that idea from other programming languages.

## 4.5. The Circle Problem Revisited

Let's go back through the entire problem-solving process for the circle problem from the previous chapter, generating a more object-oriented solution to the problem. As a reminder, here's the problem description:

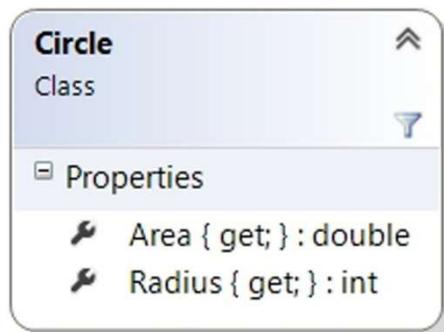
Calculate the area of the circles with integer radii from 0 to 5, then print out the radius and area for each circle.

### *Understand the Problem*

The problem hasn't changed, so you should still understand it.

### *Design a Solution*

We know what you're thinking – wouldn't it be great if we had a `Circle` class to use? Then we could create circle objects with particular radii, and even have the circle objects calculate their own areas and provide both their radius and area to us when we needed it. Great idea! Let's do it that way. Figure 4.8 provides the UML for a `Circle` class that should do what we need. Note that we didn't include any fields in the diagram (yet); that's because you're going to help us figure this part out, and we don't want to give away the punch line!



**Figure 4.8. UML for the Circle Class**

We'll need your help completing the `Circle` class code before we can actually complete our problem solution, but we'll get to that in the Write the Code step. For now, we know that the `Circle` class will provide a constructor (we'll always explicitly provide a constructor in this book) and the `Radius` and `Area` properties. Our application class will simply use those capabilities to solve the problem, so we're ready to move on to our next step.

### *Write Test Cases*

Our test case is exactly the same as it was in the previous chapter. Remember, functional test cases test the behavior of the code; since the required behavior hasn't changed, the test case doesn't change either. This is an important point. Once we have solid test cases, we can change the actual code that implements our problem solution, then run the test cases to make sure we haven't broken anything.

## Test Case 1: Checking Radius and Area Info

Step 1. Input: None.

Expected Result:

```
Radius: 0, Area: 0.00
Radius: 1, Area: 3.14
Radius: 2, Area: 12.57
Radius: 3, Area: 28.27
Radius: 4, Area: 50.27
Radius: 5, Area: 78.54
```

*Write the Code*

Remember, we need your help completing the code for the `Circle` class, so let's get started on that.

The first thing we need to decide is what our fields should be. We already know that the radius will be an integer, because that was in our problem specification. So one of our fields should be declared as

```
int radius;
```

Just as in the previous chapter, the area will be a `float`, so we'll have another field declared as

```
float area;
```

Now that we have our fields, the next thing we need to do is write our constructor. Let's say the *stub* (a small piece of code that compiles but doesn't really do anything) for our constructor looks like the following:

```
/// <summary>
/// Constructor
/// </summary>
/// <param name="radius">the radius of the circle</param>
public Circle(int radius)
{
}
```

The constructor has a single parameter called `radius` because the code calling the constructor needs to pass in the radius so the constructor can create a `Circle` object with the correct radius. At this point, we have a field for the radius (called `radius`) and the call to the constructor will pass in a value for radius (also called `radius`). It may seem confusing to have the parameter name be called the same name as the field, but calling the radius `radius` makes sense whenever we're talking about the radius, right? So our first attempt to set the field equal to the parameter might look like

```
radius = radius;
```

Unfortunately, we need to be more explicit about which of these `radius` variables is the field and which is the parameter. Essentially, the parameter named `radius` "hides" the field named `radius` so we can't get to it. That's okay, though, because C# helps us with the `this` reference (`this` is one of the C# keywords).

Our problem here is that we need to reference the field called `radius` for the current circle object. In C#, each object has access to its fields and methods through the use of the `this` reference. That lets us fix our code that sets the `radius` field by using

```
this.radius = radius;
```

It's now clear to both us and the compiler that the `radius` on the left is the field and the `radius` on the right is the parameter.

Because the radius for our circle object will never change, we should calculate the area of the circle in the constructor as well. That way, we can access the area (through the `Area` property) as many times as we like without having to recalculate the area every time. This is almost always a good approach to take in programming, but it's even more important in game development. Doing calculations costs us CPU time we could use for some other task (like figuring out the shortest path for the Orc to take to attack the Elf), so doing each calculation as few times as possible (once in this case) is best.

In the previous chapter, we calculated the area as

```
area = (float)Math.PI * radius * radius;
```

There's actually another way to square `radius` that can be helpful in some situations. What if we had to raise it to the 5<sup>th</sup> power instead? We sure wouldn't expect to type `radius * radius * radius * radius * radius`, would we? Well, the `Math` class comes to our aid again, because there's a method in the class called `Pow` (for power). This method raises the first number to the second number, so we'd use it like

```
area = (float)(Math.PI * Math.Pow(radius, 2));
```

Both the arguments for the `Pow` method need to be `double`. Although our `radius` variable is an `int` and the literal 2 is also an `int`, C# does an implicit type conversion to a `double` for us for both those arguments. Because both `PI` and the result returned by the `Pow` method are `double`, we need to do a type cast to put the area into our `float` `area` variable. We could have cast each of the terms in the multiplication to a `float` separately, of course, but in this case it's more reasonable to just get the final result before doing the type cast.

By the way, the `Math` class has lots of cool stuff (well, you know, cool math stuff anyway); you should check out the `Math` class if you're looking for ways to calculate common math functions.

Notice that we didn't need to use the `this` reference for the `area` in this code. Why not? Because there's no parameter called `area` "hiding" our `area` field, so we don't need to use the `this` reference to get to the field.

One more comment before we move on. You might have found it a little strange the way we used the `Math` class, particularly the `Pow` method. We never actually created a math object from the `Math` class, did we? Don't we need to create objects and then call the methods for them rather than calling methods directly on the class?

The answer is "Not always"! Methods that are defined in a class as being *static* methods are actually called using the class name rather than an object name. All of the `Math` methods are static methods, so you'd never actually create a `math` object (in fact, the compiler won't let you).

What about `Math.PI`? That's not even a method in the `Math` class, it's a constant – a field – in the `Math` class. We can access `PI` because the class says it's public (we'll talk a lot more about this when we're designing our own classes from scratch), and we can access `Math.PI` directly without a `math` object because constants are automatically static.

Have you ever accessed static methods directly before? Well ... yes! When we use `Console.WriteLine`, we're using the static `WriteLine` method, so we don't have to create a `console` object.

We're (finally) done talking about the constructor. We'll make both the `Radius` and the `Area` properties read-only because we want consumers of the `Circle` class to be able to access those properties, but not change them. We'll cover properties in more detail later on.

Here's a brief side note to help us easily discuss interactions between classes. Classes are designed to be used by entities external to the class; for ease of reference, we'll call those external entities *consumers* of the class. The members that a class provides to consumers of the class are said to be *exposed* because the consumers can "see" them.

Now we have the complete code for the `Circle` class; it's provided in Figure 4.9.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CirclesApplication
{
    /// <summary>
    /// A circle
    /// </summary>
    public class Circle
    {
        int radius;
        float area;

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="radius">the radius of the circle</param>
        public Circle(int radius)
        {
            this.radius = radius;
            area = (float)(Math.PI * Math.Pow(radius, 2));
        }

        /// <summary>
        /// Gets the radius of the circle
        /// </summary>
        public int Radius
```

```

    {
        get { return radius; }
    }

    /// <summary>
    /// Gets the area of the circle
    /// </summary>
    public double Area
    {
        get { return area; }
    }
}
}

```

**Figure 4.9. Circle.cs**

There are actually lots of details about building classes that we've glossed over in this discussion; we'll cover designing and implementing classes in much more detail in Chapters 12 and 13. For now, we just wanted to spend a little more time talking about variables and computations, so if you understand that part of the discussion you're in good shape.

Now we need to write our application class. This will be pretty straightforward, since all we need to do is create circle objects with the appropriate radii then print the information about each circle. Before we can write the code, though, we need to know the syntax for getting (also sometimes called reading) the `Radius` and `Area` properties. That syntax is provided below.

---

#### SYNTAX: Get (Read) a Property

```
variableName = objectName.PropertyName;
```

`variableName`, the name of the variable to hold the returned value

`objectName`, the name of the object

`PropertyName`, the name of the property we're getting

Note: We can also simply get the property without placing the returned value into a variable. For example, in the code below we just include our property accesses in our calls to `Console.WriteLine`

---

The code in Figure 4.10 shows our application class.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CirclesApplication
{
    /// <summary>
    /// Prints the radius and area for a variety of circle radii
    /// </summary>

```

```

class Program
{
    /// <summary>
    /// Outputs radius and area for circles with radii 0 through 5
    /// </summary>
    /// <param name="args">command-line arguments</param>
    static void Main(string[] args)
    {
        // create circle objects
        Circle circle0 = new Circle(0);
        Circle circle1 = new Circle(1);
        Circle circle2 = new Circle(2);
        Circle circle3 = new Circle(3);
        Circle circle4 = new Circle(4);
        Circle circle5 = new Circle(5);

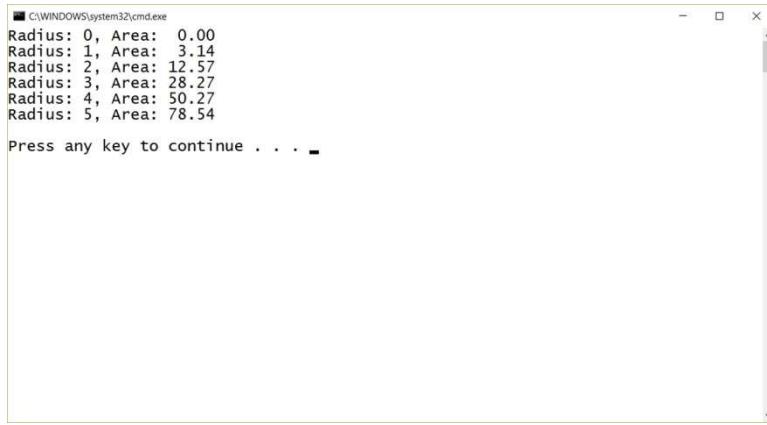
        // print object info
        Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
            circle0.Radius, circle0.Area);
        Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
            circle1.Radius, circle1.Area);
        Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
            circle2.Radius, circle2.Area);
        Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
            circle3.Radius, circle3.Area);
        Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
            circle4.Radius, circle4.Area);
        Console.WriteLine("Radius: {0}, Area: {1,5:N2}",
            circle5.Radius, circle5.Area);

        Console.WriteLine();
    }
}

```

**Figure 4.10. Program.cs***Test the Code*

So now we need to run our test cases to make sure the program works. The actual results from running the code are provided in Figure 4.11.



**Figure 4.11. Test Case 1: Checking Radius and Area Info Results**

This is just as we expected. We changed the code so we have an object-oriented solution to our problem, but from the user's perspective it's identical to our previous solution.

## 4.6. The Circle Problem in Unity Revisited

Let's solve essentially the same problem we solved in the previous section in Unity. Here's the problem description:

Display circles with integer radii from 1 to 5 in the game window, with each circle displaying its own radius and area in the Console window.

You should know that originally we wanted to have each circle display its information as text just below itself in the game, but that was far too complicated for this early in the book. We will solve that problem, though, when we get to the chapter that talks about in-game text output in Unity.

You should also know that almost every script we write in Unity is actually a class. Although there are a few exceptions in larger games you might write, this "rule" holds true for most of the examples in this book. We'll use the term script in the Unity domain to be consistent with Unity terminology, but be sure to remember that all the things we say about classes also apply to (almost all of) our Unity scripts.

### *Understand the Problem*

We've changed a number of things in the problem description, so we should make sure we know what the requirements are. First, we removed the circle with a radius of 0 because a circle with a 0 radius will be pretty hard to see! Second, the problem description says to display the circles "in the game window", but it doesn't say where, so we'll place those circles as we see fit.

### *Design a Solution*

Because each circle will be responsible for displaying itself and printing its information to the Console window once it's been placed in the scene at run time, it makes sense to write a new `Circle` script and implement the required functionality in the `Start` method in that script.

## *Write Test Cases*

Unfortunately, once we have graphical (rather than only textual) output, it becomes harder to exactly specify our expected results. For example, we don't know precisely where each of the circles will be placed in the scene, so we won't indicate in our expected results where each circle will appear. We do know, however, that there should be 5 circles with radii from 1 to 5, and we also know the radius and area that should be displayed in the Console window for each circle. Here's our test case:

### **Test Case 1: Checking Radius and Area Info**

Step 1. Input: None.

Expected Result: Five different circles, with radii 1, 2, 3, 4, and 5. Radius and area displayed in the Console window with the following values (not necessarily in the given order):

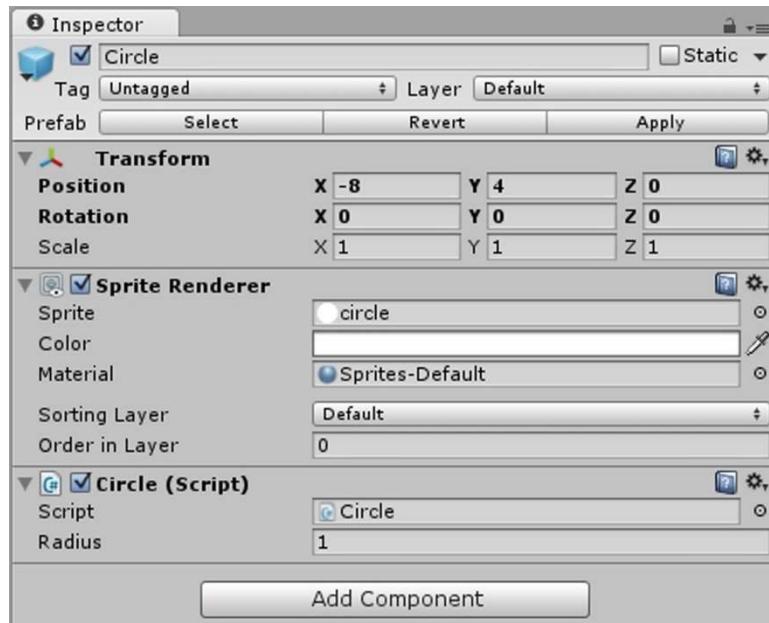
```
Radius: 1, Area: 3.141593
Radius: 2, Area: 12.56637
Radius: 3, Area: 28.27433
Radius: 4, Area: 50.26548
Radius: 5, Area: 78.53982
```

We had to add the caution that the information may not be displayed in the Console window in the given order because each Circle game object will print its information when it's added to the scene at runtime, and we don't know what that order will be.

## *Write the Code*

Our `Circle` script will be significantly different from the `Circle` class we wrote in the previous section because we need to make lots of changes to make things work properly in Unity. We'll include the script below, and interleave our comments throughout.

Before we get to the code, though, let's look at our Circle game object in the Unity editor. We'll go over a lot of the basics for 2D in Unity in a couple chapters, but for this small example we're just using a 2D Sprite as our Circle game object. As you can see from the Inspector in Figure 4.12, our Circle game object has a Transform component, a Sprite Renderer component, and the `Circle` script we added as a component. We'll see as we work through the `Circle` script how we can use the Transform component to scale the sprite to the appropriate radius.



**Figure 4.12. Circle Object in Inspector**

Okay, let's work our way through the `Circle` script, which we attached to the Circle game object:

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

/// <summary>
/// A circle
/// </summary>
public class Circle : MonoBehaviour
{
    // use [SerializeField] so we can change in the Inspector
    [SerializeField]
    int radius;
```

As the comment above states, by putting `[SerializeField]` above the `radius` field, we make it visible in the Inspector as you can see at the bottom of Figure 4.12. Doing it this way is helpful because we can just change the `radius` field in the Inspector when we place a circle game object in the scene. We'll learn more about attributes (like `SerializeField`) later in the book.

You might wonder why we didn't use a property here instead. The problem is that properties don't appear in the Inspector, while public fields and fields marked with `[SerializeField]` do. Because we really wanted access to the `radius` field in the Inspector, we couldn't use a property to provide write access to it.

```
float area;
```

The `area` field holds the calculated area for the circle; this field isn't marked with `[SerializeField]` because we don't need (and shouldn't have) access to it in the Inspector.

```

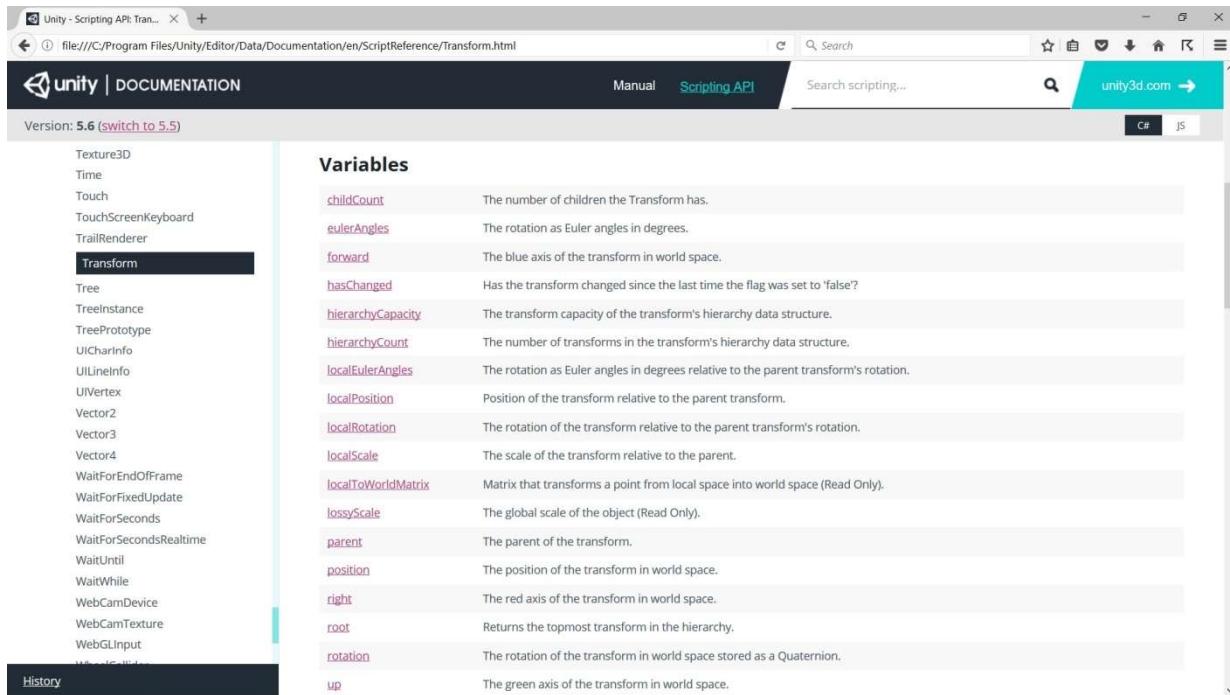
/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // calculate area
    // note that we're using UnityEngine.Mathf instead of System.Math
    area = Mathf.PI * Mathf.Pow(radius, 2);
}

```

The `UnityEngine` namespace contains a `Mathf` class<sup>15</sup> that essentially provides all the functionality of the `Math` class in the `System` namespace, but using `floats` instead of `doubles`. Because `float` is the default floating point data type in Unity, we use the `Mathf` class for our area calculation.

Next, we want the displayed circle to be scaled based on its radius. In general game development, a transform is used to read and set (write) the position, rotation, and scale of an object in the game world; that's what Unity uses transforms for as well. That tells us that we'll probably need a reference to the Transform component for our Circle game object. To find out what to do with that component, though, we'll need to read some documentation.

When we look at the documentation for the `Transform` class in the Unity Scripting Reference, we can scroll down through the Variables area looking for descriptions that talk about scale; that's what we did to get Figure 4.13.



**Figure 4.13. Transform Documentation**

It turns out that we can modify the `localScale` field for the transform that's attached to the Circle game object to scale the sprite.

<sup>15</sup> `Mathf` is actually a struct, not a class, but that distinction isn't important to us here.

```
// scale circle sprite based on radius
Vector3 scale = transform.localScale;
scale.x *= radius;
scale.y *= radius;
transform.localScale = scale;
```

To access the Transform component for a game object, we can simply reference the `transform` field (we'll learn how to access other kinds of components later in this chapter). That field is a `Transform` object, so once we have a reference to that object we can access any of the fields, properties, and methods provided by the `Transform` class. Adding `.localScale` after `transform` lets us access the `localScale` field of the `Transform` object. You should note that we access an object's fields using exactly the same syntax we use to access an object's properties.

The first line of code above accesses the `localScale` field of the game object's `transform` field and puts it into the `scale` variable; we had to follow the `localScale` link in the documentation in Figure 4.13. to learn that the `localScale` field is a `Vector3` object, so that's the data type we need for the `scale` variable.

Each `Vector3` object has `x`, `y`, and `z` fields for the vector. The second line above takes the `x` component of the current scale and multiplies it by the radius; for example, if the radius is 2 that line will double the width scale. The third line above does the same thing for the height. You should, of course, realize that we don't have to modify the `z` component of the vector because we're working in 2D in this Unity project.

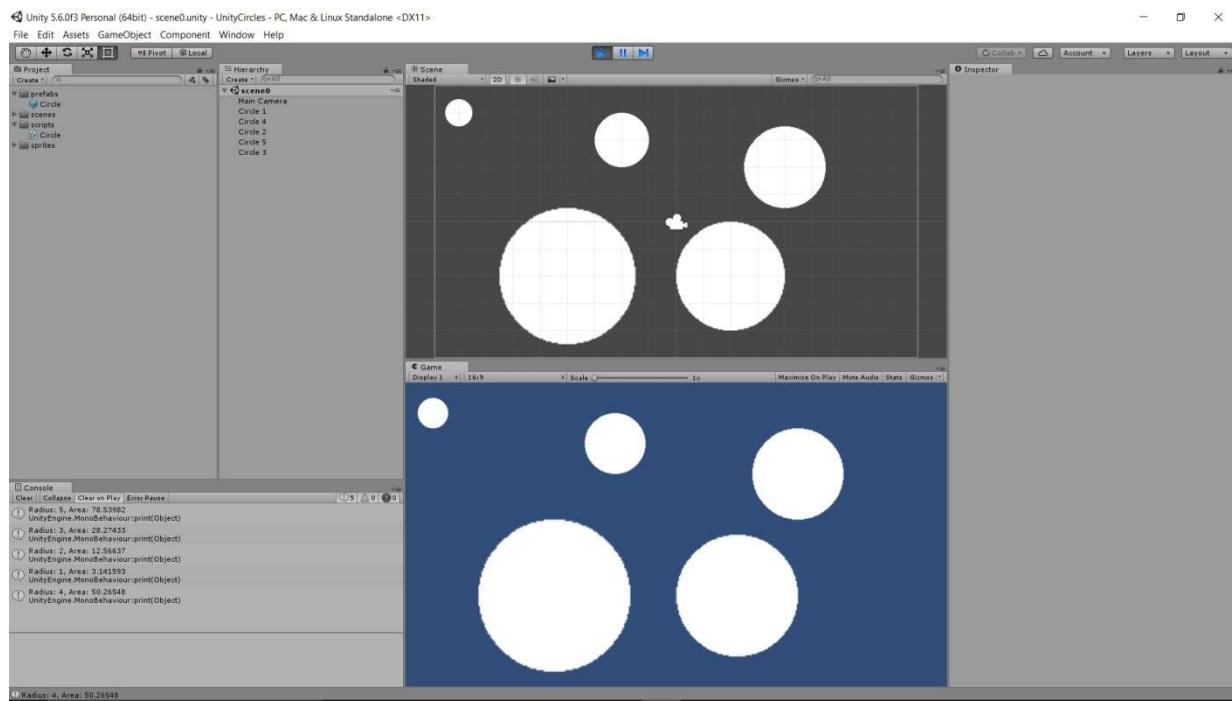
At this point, the `scale` variable holds the appropriate `Vector3` for scaling the circle, but we still need to actually scale the circle. The fourth line above sets the `localScale` field for the `transform` field to the scale we just calculated; changing that `localScale` field is what actually scales the circle when we run the game.

```
// print radius and area
print("Radius: " + radius + ", Area: " + area);
}
```

Finally, a line of code that seems pretty clear! This is just using the `print` method to output text to the Console window.

### *Test the Code*

The actual results from running our Unity game are provided in Figure 4.14. As you can see, our actual results match our expected results, though as we noted above, the order isn't the same as the order we listed in the test case.



**Figure 4.14. Test Case 1: Checking Radius and Area Info Results**

## 4.7. Putting It All Together in a Console App

Let's solve a problem using the `Card` class we described at the beginning of this chapter. Here's the problem description:

Create three cards with ranks and suits of your choosing, then print out the ranks and suits of those cards in reverse order.

### *Understand the Problem*

Our only real question is what does "reverse order" mean? For this problem, let's assume that means we print the last card we created first, then the second card we created, and finally the first card. Since the problem description says we can pick the ranks and suits, let's use the Ace of Spades, the Jack of Diamonds, and the Queen of Hearts.

### *Design a Solution*

We can easily solve this problem using an application class that creates the three cards, then prints the pertinent information in reverse order. We'll need more details about the `Card` class before we can actually write the code, but we'll worry about that when we reach Step 4 in our process.

### *Write Test Cases*

Since this program won't have any user input, all we have to do is run it to make sure it prints out the required card info.

### **Test Case 1: Checking Card Info**

Step 1. Input: None.

Expected Result:

Queen of Hearts  
Jack of Diamonds  
Ace of Spades

We have made an assumption here – that the `Card` class has already been tested, so we don't have to test it again here. Unless you actually wrote the `Card` class yourself, this is generally a reasonable assumption. If you did write the `Card` class, though, you'd need to develop unit test cases for the `Card` class in addition to the functional test case above. Unit testing details are briefly discussed throughout the book.

#### *Write the Code*

Because we want you to get used to looking at C# code that implements classes, we've included the code for the `Card` class in Figure 4.15. Remember, though, that you can write your application class just by using the `Card` class documentation; we're only providing the `Card` class code in case you want to look at the implementation.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ShowCards
{
    /// <summary>
    /// A playing card
    /// </summary>
    class Card
    {
        #region Fields
        string rank;
        string suit;
        bool faceUp;

        #endregion

        #region Constructors

        /// <summary>
        /// Constructs a card with the given rank and suit
        /// </summary>
        /// <param name="rank">the rank</param>
        /// <param name="suit">the suit</param>
        public Card(string rank, string suit)
        {
            this.rank = rank;
            this.suit = suit;
            faceUp = false;
        }

        #endregion
    }
}
```

```

#region Properties

    /// <summary>
    /// Gets the card rank
    /// </summary>
    public string Rank
    {
        get { return rank; }
    }

    /// <summary>
    /// Gets the card suit
    /// </summary>
    public string Suit
    {
        get { return suit; }
    }

    /// <summary>
    /// Gets whether or not the card is face up
    /// </summary>
    public bool FaceUp
    {
        get { return faceUp; }
    }

#endregion

#region Public methods

    /// <summary>
    /// Flips the card over
    /// </summary>
    public void FlipOver()
    {
        faceUp = !faceUp;
    }

#endregion
}

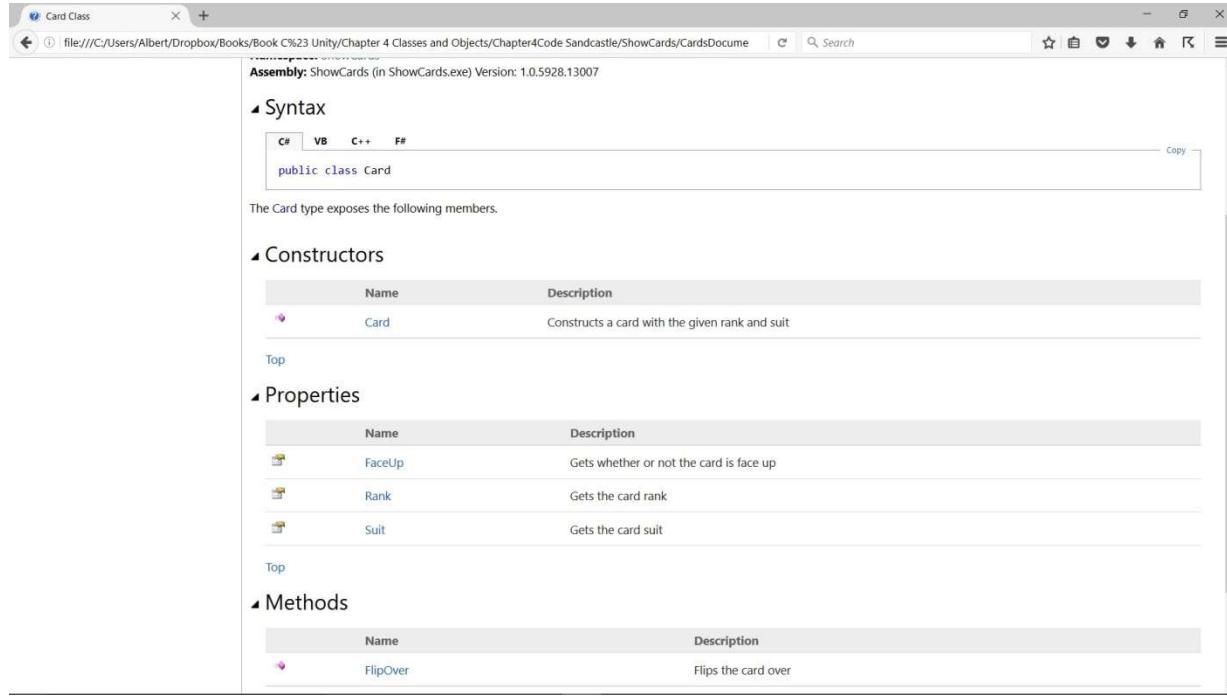
```

**Figure 4.15. Card.cs**

You may have noticed that there's `#region` and `#endregion` stuff in the code that we haven't talked about yet. You can think of regions as being analogous to comments, because they don't affect the actual compiled code and how it runs. Why do we include them, then? Really, just for convenience when we're working with the code. If we put a block of code between `#region` and `#endregion`, we can collapse and expand (hide and show) the region in Visual Studio. You should check that out in the code itself, because it's a really helpful thing to do.

Remember that we don't really need to look at the source code to use a class – if we did, it would be impossible to use all the code that's provided in the C# language and in Unity! All we really need is to

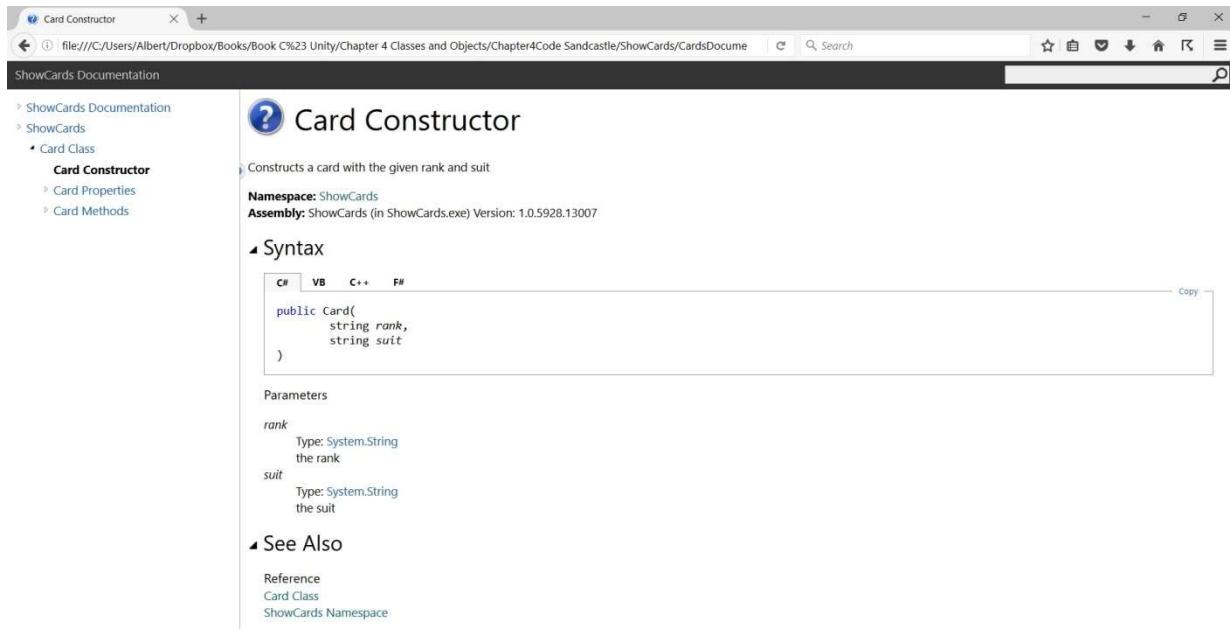
know how to use the code, not how it's actually implemented (remember the information hiding thing). Documentation to the rescue – check out Figure 4.16 (we scrolled down a little).



**Figure 4.16. Card Class Documentation**

When we look at the `Card` class documentation, we see that we'll need to use the constructor to create our card objects, the `Rank` property, and the `Suit` property. Let's start by figuring out how to create the three cards first, then move on to using the properties.

Unfortunately, at this point we don't know what arguments need to be passed into the `Card` constructor, so we need to look at the documentation a little more to figure this out. If we click on the link for the constructor in Figure 4.16, we get the details shown in Figure 4.17.



**Figure 4.17. Card Constructor Documentation**

The documentation shows that we need to pass in `string` arguments for the rank and suit to the constructor, so our code to create the three cards should be

```
Card firstCard = new Card("Ace", "Spades");
Card secondCard = new Card("Jack", "Diamonds");
Card thirdCard = new Card("Queen", "Hearts");
```

We could access the first card's properties using

```
string firstCardProperties = firstCard.Rank + " of " + firstCard.Suit;
```

This would be a bit awkward, because we don't really need a variable to store the card properties, we just need to print them out. Instead, we can do it this way:

```
Console.WriteLine(firstCard.Rank + " of " + firstCard.Suit);
```

Now we can use our knowledge of the `Card` class to write our application class. Figure 4.18 provides the completed code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ShowCards
{
```

```

///<summary>
/// Creates and shows playing cards
///</summary>
class Program
{
    ///<summary>
    /// Creates and shows the cards
    ///</summary>
    ///<param name="args">comand-line arguments</param>
    static void Main(string[] args)
    {
        // create the cards
        Card firstCard = new Card("Ace", "Spades");
        Card secondCard = new Card("Jack", "Diamonds");
        Card thirdCard = new Card("Queen", "Hearts");

        // print the cards in reverse order
        Console.WriteLine(thirdCard.Rank + " of " + thirdCard.Suit);
        Console.WriteLine(secondCard.Rank + " of " + secondCard.Suit);
        Console.WriteLine(firstCard.Rank + " of " + firstCard.Suit);

        Console.WriteLine();
    }
}
}

```

**Figure 4.18. Application Class**

Notice that we're reading the same properties, `Rank` and `Suit`, for each of our three card objects. We can do that because the `Card` class tells us that any object of the class will have those properties (and others). C# knows which object we should get the property from because we put the object name before the property name; that's why each object having its own identity is important. As you can see, reading properties looks a lot like calling a method; we just don't need to include parentheses after the property name.

Also, we've embedded the property accesses within a call to `Console.WriteLine`. Since we're only printing the rank and suit for each card, there's no need to put those results into variables first before printing them.

#### *Test the Code*

Now we simply run our program to make sure we get the results we expected. Figure 4.19. shows a screen snap of the output when we run Test Case 1. It worked!



**Figure 4.19. Test Case 1: Checking Card Info Output**

## 4.8. Putting It All Together in a Unity Script

Let's solve a problem in Unity similar to the one in the previous section. Here's the problem description:

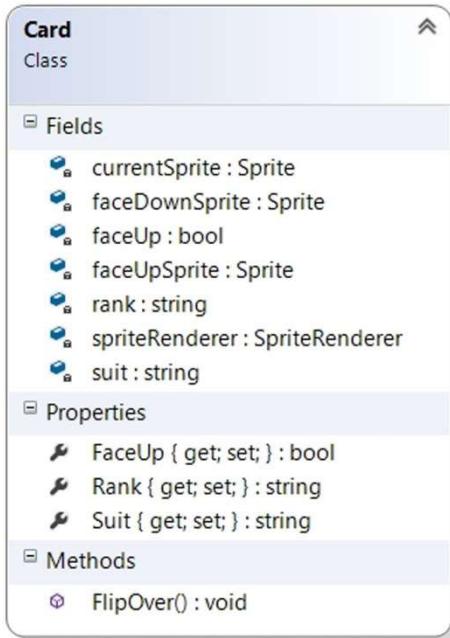
Create three cards with ranks and suits of your choosing, then display the three cards, face up, on the screen.

### *Understand the Problem*

This seems pretty straightforward, so we can move along to the next step.

### *Design a Solution*

We could just display sprites for three cards and declare victory, but let's actually develop a `Card` script that includes access to strings for the rank and suit of the card as well as the ability to flip the card over. Even though we don't need that capability for this specific problem, that approach is more in the spirit of having a Card game object with state, behavior, and identity.



**Figure 4.20. UML for Card Script**

Notice that the UML for the `Card` script has four more fields than our original `Card` class: `faceUpSprite`, `faceDownSprite`, `currentSprite`, and `spriteRenderer`. We'll use these fields to store Sprites for the Card game object and to help us display the card properly when it's face up and when it's face down. We don't provide properties for these fields because a consumer of the class shouldn't have any need to access them; they're just for the class itself. We'll mark most of the fields with `[SerializeField]` when we Write the Code so we can populate them in the Inspector.

Within the Unity editor, we create an empty game object that we attach our `Card` script to. We also add a Sprite Renderer component to the game object so it can actually display the appropriate sprite. We'll call our new object a Card game object throughout the rest of this section.

### *Write Test Cases*

Since this program won't have any user input, all we have to do is run it to make sure it displays the three cards we set up in Unity. Just like in our Unity test case for our circles problem, we'll have to be a little vague about the details of the output, but we can certainly provide enough detail to make sure our code is running properly.

### **Test Case 1: Checking Card Display**

Step 1. Input: None.

Expected Result: Cards for the Queen of Hearts, Jack of Diamonds, and Ace of Spades

### *Write the Code*

For this problem solution, the interesting details are actually in the `Card` script, so that's what we'll focus on here. There are some steps we take in the Unity editor to set up the scene properly, but those are "Unity things" rather than "C# things". Don't worry, though, because in Chapter 6 we'll actually discuss some important Unity concepts for 2D games.

Okay, let's go through the `Card` script:

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

/// <summary>
/// A playing card
/// </summary>
public class Card : MonoBehaviour
{
    #region Fields

    // card sprites
    [SerializeField]
    Sprite faceUpSprite;
    [SerializeField]
    Sprite faceDownSprite;
```

Each Card object has a reference to the sprite it displays when it's face up and the sprite it displays when it's face down. We mark these fields with `[SerializeField]` so we can populate them in the Inspector, but how do we actually do that?

Figure 4.21 show the Project window for our project for our problem solution. As you can see, we have a sprites folder that contains the 3 face up sprites we need and the face down sprite that all cards will have. We created the sprites folder from within the Unity editor – not in our operating system, by right clicking in the Project window and creating the folder – and then just added our png files to that folder – this time in our operating system. When we do it this way, Unity automatically imports those sprites into our project. We can then just drag the appropriate sprites from the Project window onto the script fields in the Inspector to populate those fields.

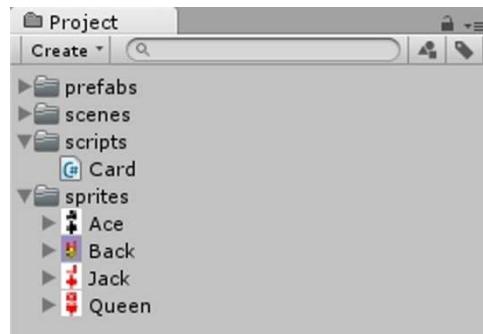


Figure 4.21. Project window

```
// other card state
[SerializeField]
string rank;
[SerializeField]
string suit;
[SerializeField]
bool faceUp;
```

We need to manually populate these fields in the Inspector so that the corresponding properties return the appropriate values if a consumer of the class reads them. It would be awesome if we could have the script automatically process the `faceUpSprite` field we populated above to figure out the `rank` and `suit` fields, but that's definitely a harder problem to solve than we want to tackle at this point!

Figure 4.22 shows the script fields in the Inspector for our Queen of Hearts card after we populate the fields in the Inspector. Because the `faceUp` field is a `bool`, so it can only be `true` or `false`, it shows up as a checkbox in the Inspector (checked is `true`, unchecked is `false`).



**Figure 4.22. Inspector**

```
// saved for efficiency
SpriteRenderer spriteRenderer;
```

Because we need access to the `Sprite Renderer` component of the `Card` game object whenever we flip the card over, we just find that component once in the `Start` method and save it in the `spriteRenderer` field so we don't have to look for it every time. That saves us a little CPU time whenever we flip the card over, and especially in games, saving CPU time is almost always a good idea!

```
#endregion

#region Properties

/// <summary>
/// Gets the card rank
/// </summary>
public string Rank
{
    get { return rank; }
}

/// <summary>
/// Gets the card suit
/// </summary>
public string Suit
{
    get { return suit; }
}

/// <summary>
/// Gets whether or not the card is face up
/// </summary>
public bool FaceUp
{
    get { return faceUp; }
}
```

```
#endregion
```

We'll discuss the details of how properties work later in the book, so for now, you should just realize that they provide access to the appropriate fields in the class (script).

```
#region Methods

/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // save sprite renderer in a field for efficiency
    spriteRenderer = GetComponent<SpriteRenderer>();

    // set appropriate sprite
    SetSprite();
}
```

Recall that the `Start` method is one of the methods we get in our default Unity scripts, and that it gets called once when the game object is added to the scene at run time. We didn't need the `Update` method, so we deleted it from our script.

The first line of code in the method finds the Sprite Renderer component of the Card game object the script is attached to and puts it in the `spriteRenderer` field. When we needed access to the Transform component of our Circle game object in the circles problem, all we had to do was access the `transform` field. That's because every game object we create gets a Transform component by default, so the Unity folks made it easy to access that component since Unity game developers need to do that so often.

To get access to other components, though, we call the `GetComponent` method. This method call looks different from the method calls we've seen before because it has `<SpriteRenderer>` before the open parenthesis. That's because `GetComponent` is actually something called a *generic method*, something that most programmers learn how to write in intermediate or advanced programming classes. The good news is that we don't have to write a generic method, we just have to call one. You can think of this as providing a special argument when you call the method; in this case, that special argument is the type (class name, `SpriteRenderer` in this case) of the component we want the `GetComponent` method to find.

The second line of code calls a `SetSprite` method that we wrote in the script; we'll get to that soon. We actually initially wrote the code to set the sprite properly here, but when we got to the `FlipOver` method (coming up next) we discovered that we needed those same lines of code in that method. Although we could have just copied and pasted the code, that's almost NEVER the correct way to reuse code in our programs. We'll discuss the details of how to write your own methods later in the book, but we wanted to be sure to show you the right way to do it here.

```
/// <summary>
/// Flips the card over
/// </summary>
public void FlipOver()
{
    faceUp = !faceUp;
```

```
// set appropriate sprite
SetSprite();
}
```

Remember that the not Boolean operator (`!`) changes a field to `false` if it's currently `true` and `true` if it's currently `false`. That's exactly what we need here to flip a card over, so that's what our first line of code uses to change the state of the card.

In our Unity solution, flipping the card over will also require that we change the sprite to match the current state of the card (face up or face down). That's why we also call the `SetSprite` method, which is next.

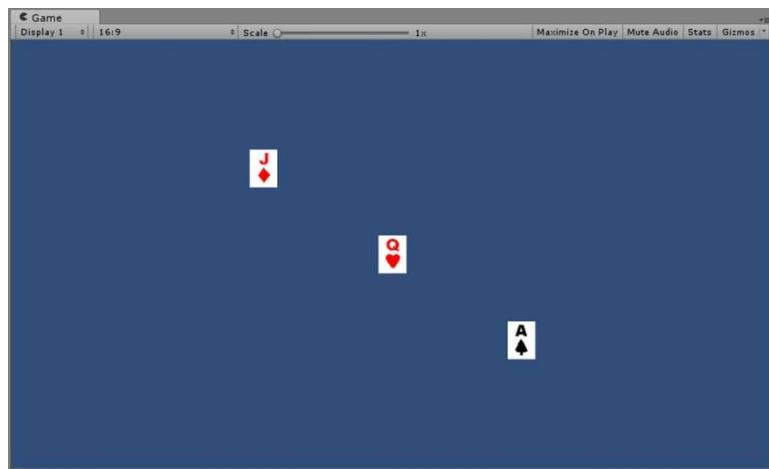
```
/// <summary>
/// Sets the sprite
/// </summary>
void SetSprite()
{
    // set appropriate sprite
    if (faceUp)
    {
        spriteRenderer.sprite = faceUpSprite;
    }
    else
    {
        spriteRenderer.sprite = faceDownSprite;
    }
}

#endregion
}
```

We haven't discussed the selection control structure yet, so we'll just say that the `SetSprite` method checks the value of the `faceUp` field to determine which sprite should be displayed. Because `SpriteRenderer` objects (remember, that's the data type for our `spriteRenderer` field) have a `sprite` field that determines what sprite gets rendered (drawn), we just set that field to the appropriate sprite (`faceUpSprite` or `faceDownSprite`) based on whether the `faceUp` field is `true` or `false`.

### *Test the Code*

The actual results from running our Unity game are provided in Figure 4.23; since we see the three cards we're supposed to see, the test case passes. The art leaves something to be desired (that's programmer art from the author), but at least they're the correct cards!



**Figure 4.23. Test Case 1: Checking Card Display**

## 4.9. Common Mistakes

### *Trying to Use an Object Before It's Created*

Despite our suggestion that you usually create your objects when you declare them, you might still end up forgetting to do that. The compiler should warn you about that if you do it, but if it doesn't, your program will "blow up" when it tries to use an object you didn't create. Please take our suggestion whenever it makes sense!

This doesn't apply to Unity scripts, because the object for a script (class) is created when the game object it's attached to is added to the scene at run time, but it definitely applies to console apps.

### *Adding Parentheses After the Property Name*

Remember, the syntax for reading properties is very similar to that for calling methods. What if we were trying to read the `Rank` property on the `thirdCard` object and we used the following (which erroneously includes parentheses):

```
thirdCard.Rank()
```

What happens? The compiler will give you an error message, telling you

```
Non-invocable member 'ShowCards.Card.Rank' cannot be used like a method.
```

In other words, you tried to invoke (a fancy word for call) the property as though it's a method, but it's not.

### *Forgetting the Parentheses After the Method Name*

The opposite problem – forgetting to include parentheses after a method name – might also happen to you. Consider the line of code below (which is missing the required parentheses):

```
Console.WriteLine;
```

What happens? The compiler will give you the following error message:

Only assignment, call, increment, decrement, await, and new object expressions can be used as a statement

Boy, that seems a lot more cryptic than the previous compiler error message we looked at. Rather than trying to really understand the details of that error message (which you'll understand much better by the end of the book), you should just realize that if you see this error message, you should add the required parentheses after the method name.

# Chapter 5. Strings

In previous chapters, we've talked about both value and reference types and how we can use a variety of .NET and Unity classes to help us solve problems in C#. In this chapter, we introduce two more classes that let us do lots of cool stuff with strings of characters. Those classes are the `String` class and the `StringBuilder` class. Let's take a look.

## 5.1. The String Class

A `string` is simply a sequence (or string) of characters: `ftatrwsy`, `Clash`, and `triathlon` are some examples. We might use a `string` to store a player's user name or an invoice number. Here are some examples:

```
string gamertag = "Dr. T";
string invoiceNumber = "AC-3412";
```

The more astute readers (all of you, we're sure) have noticed that even though `String` is a class, we're creating new `string` objects without explicitly using a constructor like:

```
String gamertag = new String("Dr. T");
String invoiceNumber = new String("AC-3412");
```

Because programmers use strings so regularly, C# lets us use the more compact syntax for assigning a value to a value type for `string`, even though `string` is actually a reference type. In fact, while there are 8 different overloaded constructors for the `String` class, none of them actually let us pass in a string literal as shown above!

You probably also noticed that we've been using `string` and `String` interchangeably. We'll typically use `String` when we're talking about the class and `string` when we're talking about an object of the class, but they're essentially the same thing. You should use whichever style you prefer (or your professor prefers).

Because `String` is a class, there are lots of cool properties and methods available to us. Let's take a look at a few of them; you can get the complete set of methods from the MSDN documentation for the `String` class.

For example, we can find out how long a `string` is by accessing the `Length` property. Say we want to store the length of a `string` object called `lastName` in an `int` variable called `lastNameLength`; we'd use the following:

```
string lastName = "Elliott";
int lastNameLength = lastName.Length;
```

which puts the number of characters in `lastName` (in this case, 7) into `lastNameLength`.

We can also try to find characters or strings within a `string` by using the `IndexOf` method. For example, say we wanted to find out where the character `,` appears in a `string` called `carPrice`. We could use something like this:

```
string carPrice = "16,000";
int commaLocation = carPrice.IndexOf(',');

```

The `IndexOf` method returns the location of the first comma in the string of characters in `carPrice`. There are a couple key points to consider, though. The characters in the `string` are numbered starting at 0, so if a character we're looking for is the first character in the string, `IndexOf` returns 0. For the example above, `IndexOf` returns 2. Also, what happens if the character doesn't appear in the `string` at all? The `IndexOf` method returns -1. That means we can easily tell if the character was found by comparing the location to -1; if the character was in the `string`, the location will be something other than -1.

We can also find the location of a string in another string using the `IndexOf` method. Say we're looking for the string "Leppard" in a `string` called `bandName`. We could use the following:

```
string bandName = "Def Leppard";
int lepLocation = bandName.IndexOf("Leppard");
```

The `IndexOf` method returns the location of the first character of the string we looked for, which in this case is 4.

This brings up an interesting point. Why can we call the `IndexOf` method with either a character or a string argument? Don't methods have parameters that are particular data types? How can `IndexOf` take either one? The answer is easier than you might think; remember our discussions about *overloading*? The `String` class provides multiple `IndexOf` methods; one of those overloads has a `char` parameter and another has a `string` parameter. Even though the methods have the same name, C# can figure out which method to use based on the data type of the argument provided in the method call. In fact, there are 9 different `IndexOf` methods for the `String` class!

Let's look at one more method before moving on to the next class. Say we want to extract a substring from a particular `string` and put it into another `string`. For example, let's extract the substring that represents how many thousands of dollars the car costs from `carPrice`; in other words, the characters before the comma. We can do that using the `Substring` method and the `commaLocation` we found previously:

```
string thousands = carPrice.Substring(0, commaLocation);
```

The first argument gives the beginning index for the substring; the character at the beginning index is included in the substring. The second argument gives the length of the substring we want to extract. Let's look at an example to see why using the comma location as the length of the substring works in this case. Say our `carPrice` variable has a value of `16,000`; in this case, `commaLocation` would have a value of 2. Because we start counting at 0, this is also exactly the number of characters from the start of the string up to but not including the comma, which is just the substring we want. While this may seem a little confusing at first, doing it this way makes lots of the typical string processing programming tasks easier to program.

We actually made a number of assumptions here. First of all, we assumed a comma appeared in `carPrice` somewhere (so the `IndexOf` method didn't return -1). We'll learn how to test conditions like this in the Selection chapter. Also, we assumed that the car costs less than 1 million dollars. If it didn't, the first comma would appear after the millions of dollars, and we wouldn't actually be extracting

thousands of dollars as we thought. We'll just live with this assumption, because it sure seems reasonable!<sup>16</sup> Finally, we assumed that the car price is stored as a `string`. It would certainly be reasonable to store the car price as a `float` or `int` instead (without the commas, of course), especially if we needed to use the car price in math calculations. It does provide a nice string example, though.

There's actually another version of the `Substring` method that will give us a substring from the beginning index all the way up to (and including) the last character in the original string. So if we wanted to extract the characters after the comma in the `carPrice`, we could use:

```
string afterComma = carPrice.Substring(commaLocation + 1);
```

Why did we add 1 to `commaLocation` for the beginning index of the substring? Because we didn't want the comma included in `afterComma`, so we "skipped over it" when we took the substring.

As we said, there are lots of useful methods provided by the `String` class. Instead of trying to learn them all now, we'll just go to the MSDN documentation to look for particular methods as we need them.

## 5.2. The `StringBuilder` Class

The `String` class is very useful in C#, and we're going to find we use it a lot. But there is one important restriction on a `string`; it's *immutable*. In other words, once we create a `string` object, we can't change that object. We can certainly make changes and put them in a different `string` object (that's exactly what the `Substring` method does), but sometimes what we really want to do is change this object instead of creating a new one.

To get a mutable string of characters, all we need to do is use the `StringBuilder` class instead of a `string`. Let's look at the differences between them.

First of all, we have to explicitly use the class constructor when we create a `StringBuilder` object:

```
StringBuilder lastName = new StringBuilder("Romijn");
```

Now that we have a `StringBuilder` object we can use the methods provided by the `StringBuilder` class. Let's look at the `Append` method, which simply adds whatever we provide as an argument to the end of the string builder. If you look at the documentation for the `Append` method, you'll see that there are 19 overloads; this overloading thing is apparently pretty useful! The overloading lets us append any value type (and other stuff) to a string builder; numbers will be converted to strings before being appended.

Let's say that we know the person named `lastName` is going to get married, and that she plans to hyphenate her last name. While we wait to find out who she's marrying, we'll add the hyphen to `lastName`:

```
lastName.Append(' - ');
```

---

<sup>16</sup> Though we could of course write code that would handle this as well with some more advanced programming techniques. You'll want to do that if you happen to be shopping for a Bugatti Veyron ...

At this point, `lastName` consists of the sequence of characters `Romijn-`. Finally, this person meets someone whose biggest claim to fame is starring in a television series with the Olson twins, so we can complete our modifications of `lastName`:

```
lastName.Append("Stamos");
```

Now, what if we know we're done changing `lastName` and we want to put it into a `string` variable called `finalLastName` rather than a `StringBuilder`? We use the `ToString` method to convert the `StringBuilder` object to a `string`:

```
string finalLastName = lastName.ToString();
```

We could then use the methods provided by the `String` class that we've already discussed.

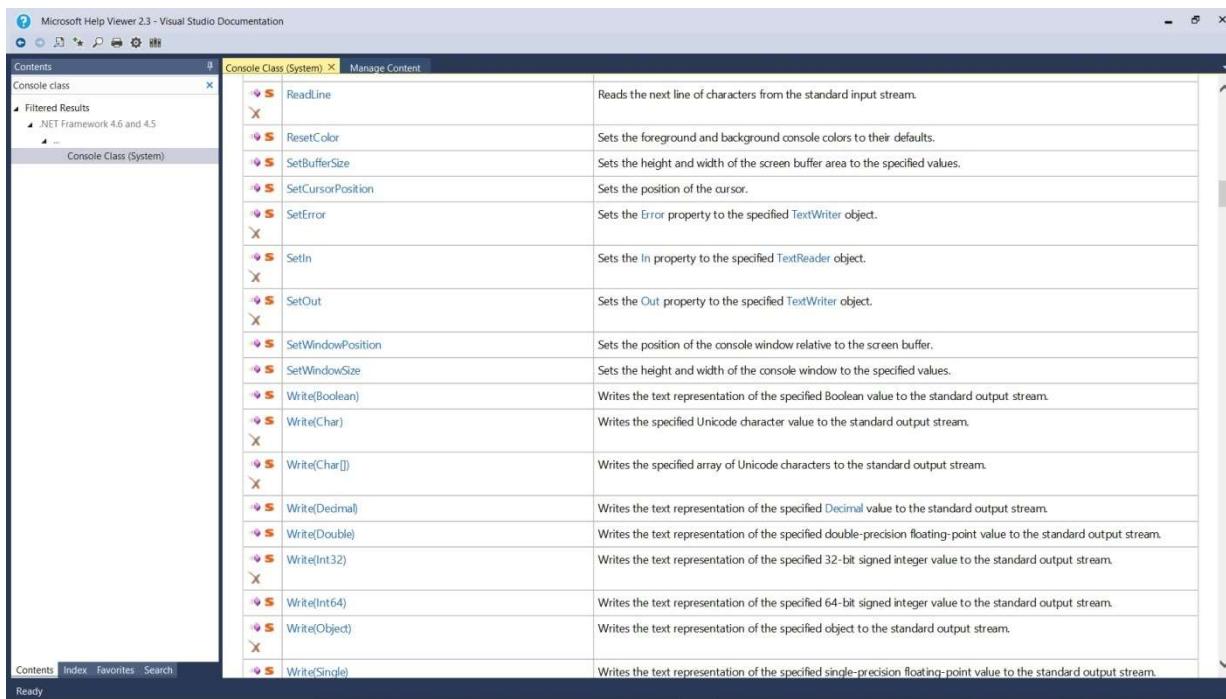
By the way, all data types in C# – even the value types – automatically have a `ToString` method. We'll talk about how that happens later in the book when we discuss inheritance, but knowing that now may come in handy as we move forward.

The `StringBuilder` class also provides other useful methods, including methods that let us remove substrings, insert substrings at particular locations, and even replace substrings with other substrings. You should check out the documentation for the `StringBuilder` class as needed to figure out how to use those methods.

### 5.3. Getting Input

We started talking about how to use `Console.WriteLine` to output strings way back in Chapter 1, but we haven't actually talked about how we get input. We'll discuss Unity input in a few chapters, so we'll focus on input in console applications here.

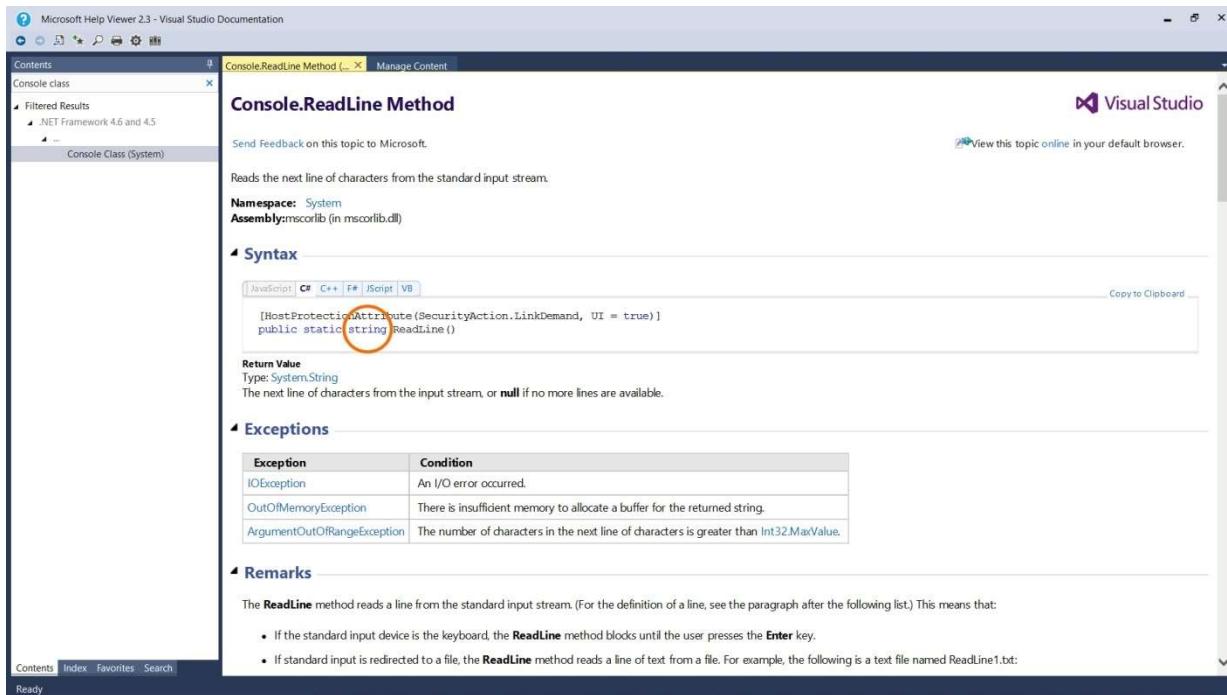
As you can see from the documentation in Figure 5.1., the `Console` class provides lots of useful properties and methods. Let's see how to use the `ReadLine` method to get keyboard input (we scrolled down to the `ReadLine` method in the Methods area of the help documentation to generate Figure 5.1.).



**Figure 5.1. Console Class Documentation**

The documentation says that the `ReadLine` method "Reads the next line of characters from the standard input stream." That may not mean much to you until you realize that the standard input stream is just the keyboard.

We can also discover – again, from the documentation (see Figure 5.2.) – that the method returns a `string`. To go to the documentation shown in Figure 5.2., we clicked the `ReadLine` link shown at the top of Figure 5.1. We also circled the method return type in the figure, though it's of course not circled in the actual documentation. Basically, when we include a call to the `Console.ReadLine` method in our code, the computer waits for the user to press the <Enter> key (probably typing some other keys first), at which point the method returns the corresponding `string`.



**Figure 5.2. ReadLine Method Documentation**

Here's some code to read in a user name:

```
Console.WriteLine("Enter your user name: ");
string userName = Console.ReadLine();
```

Notice that we prompt the user for their input before actually reading it in. If we didn't include the prompt, the program would just reach the second line above and hang, waiting for the user to type something. Most users wouldn't simply know intuitively if the software that they're running stops that they should enter their user name! You always want to prompt the user for input.

You may have also noticed that we used `Console.Write` rather than `Console.WriteLine` to display the prompt. That's because it looks much nicer to print the prompt and have the user enter the requested information next to the prompt rather than on the following line. After the user types in their user name and presses <Enter>, the `userName` variable will contain the string of characters they entered (not including the <Enter>). That's exactly what we needed.

Doing input this way works perfectly well for strings, but what if we need to read in some other type like an `int` instead? For example, what if we already know that the player is a Paladin but we want to find out their level? We could certainly read in their level as a `string` and store it as a `string`, but that doesn't really make sense because it's an integer. There must be a better way.

Of course there is! We can use the `Int32` struct to help us here (remember, structs are like classes). The `Int32` struct represents a 32-bit signed integer just like the `int` data type; in fact, `int` is actually just an alias – another name for the same thing – for the `Int32` struct. We've already seen `Int32` and `int` used interchangeably in the documentation we've been reading as well. You should know that C# provides a struct for each of the C# value types; you'll find the existence of those structs and their methods (like `Parse`) to be particularly useful when processing user input.

Wait a minute, what's this `Parse` method? From the `Int32` documentation, the `Parse` method "converts the string representation of a number to its 32-bit signed integer equivalent." That sounds like exactly what we need! Here's how we can use it:

```
Console.WriteLine("Enter your level: ");
int level = int.Parse(Console.ReadLine());
```

Remember, we provide information to the methods we call through the arguments we pass to those methods. We need to pass a `string` to the `Parse` method, and we get that string by calling the `Console.ReadLine` method. It's perfectly valid and reasonable to call one method and provide its results as an argument to another method, which is precisely what we're doing here. When we execute the second line of code above, `level` will be assigned the `int` value of the `string` entered by the user.

For now, that will do exactly what we need. You should realize, however, that users don't always enter valid input even if we ask nicely. For example, say we have a "playful" user who enters the following string for their level:

```
123 &*^^^ 445456 GHFHF
```

There are lots of things wrong with this input. First of all, the entire string can't be represented as an integer because it contains non-numeric characters; we'll learn about how to handle that when we discuss exceptions. Just as importantly, everyone knows there's no such thing as a level 123 Paladin! We'll learn how to handle invalid input data when we learn about iteration. For now, though, we'll just have to trust the user to enter valid data.

## 5.4. Putting It All Together

Let's solve a problem that requires that we use some of the `string` stuff we've just learned. Here's the problem description:

Read in the names (first name, middle initial, and last name) of three different people from the user. Print the three names in the following format:

```
lastName, firstName middleInitial.
```

Also, identify the last names that are hyphenated.

### *Understand the Problem*

Do we understand the problem? The problem description doesn't say how we have to store the names, so that will be a design decision on our part. What about "identifying" the last names that are hyphenated? How will we do that? After printing the 3 names in the above format (we HAVE to meet that requirement), let's print the three last names again with an indicator of whether or not they're hyphenated, like:

```
Public, Hyphenated: false
Flagudorski, Hyphenated: false
Romijn-Stamos, Hyphenated: true
```

## Design a Solution

Let's use a `Name` class that will store the name information for us and will also provide properties that let us get the parts of the name and whether or not the last name is hyphenated. Although we're probably going to eventually need to be able to change the last name when people get married or change their name for some other reason, we won't add that capability in our solution to this problem.

Using the `Name` class, we can create name objects containing the names we want to use and have those objects do their own string processing to extract the last name, first name, and middle initial, and to tell whether or not the last name is hyphenated. Figure 5.3 provides a class diagram for a `Name` class that should do what we need.



**Figure 5.3. Class Diagram for the Name Class**

The features of the `Name` class make our application class design easy. All we'll have to do is create three `Name` objects as we read in the names from the user and print the names and whether or not they're hyphenated.

## Write Test Cases

Our only real decision here is which names we should use to test our program. We're going to want to include at least one name where the last name is hyphenated and one where the last name isn't. Other than that, it really doesn't matter which names we use. When we run the program, we'll make sure the names are printed in the correct format and that hyphenated and non-hyphenated last names are identified correctly.

### Test Case 1: Checking Name Info

Step 1. Input:

John Q. Public  
 Oscar Z. Plagudorski  
 Rebecca L. Romijn-Stamos

Expected Result:

Public, John Q.  
 Plagudorski, Oscar Z.  
 Romijn-Stamos, Rebecca L.  
 Public, Hyphenated: false  
 Plagudorski, Hyphenated: false  
 Romijn-Stamos, Hyphenated: true

Although we're going to discuss details of the actual implementation of the `Name` class, we'll be treating that as a complete, fully-tested class you can use as a consumer. Otherwise, we'd need to write unit test cases for that class, and we still have a while to go before we learn how to do that.

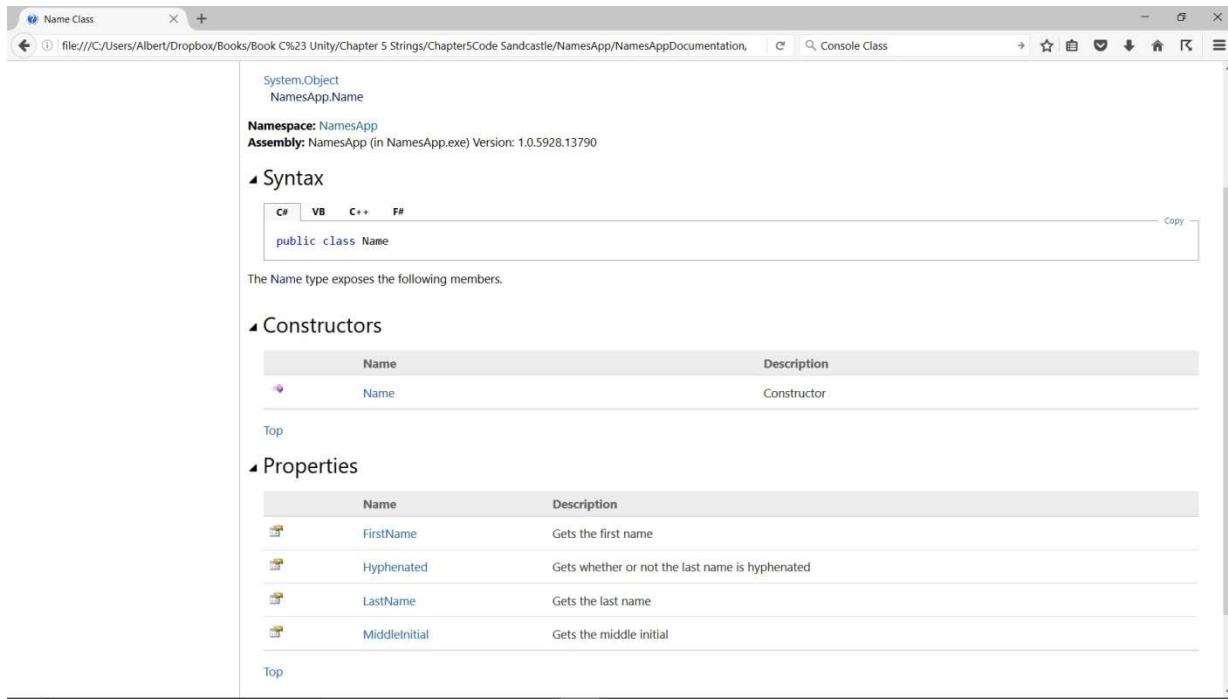
#### *Write the Code*

First, we should look at the source code for the `Name` class. Remember that we don't have to look at the source code to be a consumer of the class – all we really need is the documentation for the class – but there are some interesting aspects of the implementation that we want to discuss.

The UML diagram for the `Name` class tells us that we'll have three fields to hold the first name, middle initial, and last name, so we declare those fields (they're just variables) first:

```
string firstName;
char middleInitial;
string lastName;
```

We'll look at the implementation of the constructor and `Hyphenated` property momentarily, but before we do, take a look at the `Name` members documentation provided in Figure 5.4. Why did we just switch from UML to the documentation? First of all, consumers of classes should always have access to MSDN-style documentation for those classes, but they usually won't have access to UML for the classes. That means we should refer to documentation as much as possible rather than counting on UML. Why did we use UML for the fields then? Because documentation typically won't provide information about the fields internal to the class – consumers of the class don't need to, and shouldn't, know what the internal fields are (remember information hiding?) – so we needed the UML to show us which fields to include in our implementation of the class.



**Figure 5.4. Name Members Documentation**

So the general pattern for this is that we use UML as part of our design process, write and test the code until it works correctly, then generate the documentation for consumers of the class to use.

Let's take a look at the constructor. It makes sense for us to simply have the consumer of the class provide the full name and have the `Name` class actually do the work of extracting the first name, middle initial, and last name from the full name, so that's the way we do it here. At this point, we're going to have to assume that the user enters a valid full name (with no extra spaces or single-name names like Bono) that follows the format:

```
firstName middleInitial. lastName
```

Don't worry, as we move through the book we'll learn techniques we can use to handle special cases, but given what we know at this point we need full names to be in the above format.

You should immediately be able to see that the `IndexOf` method is a great choice for extracting the first name from the full name. All we have to do is find the first space in the full name; once we have that, the first name is just a substring of the full name up to the first space. Here's the code:

```
int firstSpaceLocation = fullName.IndexOf(' ');
firstName = fullName.Substring(0, firstSpaceLocation);
```

We could have actually done this in one line of code instead, without using the `firstSpaceLocation` variable (see if you can figure out how), but since we're going to need to know where the first space is to extract the middle initial also, we might as well just look for the first space once rather than twice.

Extracting the middle initial is straightforward as well, though we will need to learn something new about working with strings to do it. You might think that we should just use `Substring` again as shown in the following code:

```
middleInitial = fullName.Substring(firstSpaceLocation + 1, 1);
```

but this won't actually work. The problem is that the `Substring` method returns a `string` and the `middleInitial` variable is a `char` (as it should be). We could try to do some additional manipulation of the `string` returned by the `Substring` method, but there's actually an easier way.

Remember that a `string` is used to represent a sequence of characters. The good news is that we can access the single character following the first space in the `string` by using the following:

```
middleInitial = fullName[firstSpaceLocation + 1];
```

The value we provide between the square brackets is called the *index*, which is really just the position of the character we want to retrieve from the `string`. Obviously, we can access any character locations in the `string` we want, not just the one following the first space!

Extracting the last name also requires that we learn one more `string` method. We know that the last name is the sequence of characters following the last space in the full name. Extracting that substring will be just like extracting the dollars from the car price in our example above, but how do we find the location of the last space? It seems like we need a method similar to the `IndexOf` method, which finds the location of the first occurrence of the specified character. Luckily, `string` also has a `LastIndexOf` method, which finds the last occurrence of the specified character; this is exactly what we need! Here's the code we can use to extract the last name:

```
lastName = fullName.Substring(fullName.LastIndexOf(' ') + 1);
```

Notice that we add 1 to the results of the `LastIndexOf` method to make sure the last space isn't included as part of the last name (yes, that was a bug we fixed while testing the `Name` class).

Putting all the above code together, we have the following constructor:

```
/// <summary>
/// Constructor
/// </summary>
/// <param name="fullName">the full name</param>
public Name(string fullName)
{
    // extract parts from full name
    int firstSpaceLocation = fullName.IndexOf(' ');
    firstName = fullName.Substring(0, firstSpaceLocation);
    middleInitial = fullName[firstSpaceLocation + 1];
    lastName = fullName.Substring(fullName.LastIndexOf(' ') + 1);
}
```

We're not going to discuss the source code for three of the properties since we'll cover those once you start designing your own classes, but we will cover the `Hyphenated` property because it's a little more interesting.

The `Hyphenated` property returns a `bool`, and we know from the `Name` documentation that the property should return `true` if the last name is hyphenated and `false` if it's not. There's even an easy way to figure this out; all we need to do is try to find a hyphen in the last name. If we find one we should return `true`, and if we don't we should return `false`. How do we look for a particular character in the `string` for the last name? With the `IndexOf` method.

Remember, the `IndexOf` method returns the location of a particular character in the `string` or `-1` if the character doesn't appear in the `string`. We can therefore tell whether or not a hyphen is in the last name by comparing the value returned by the `IndexOf` method to `-1`. For example,

```
lastName.IndexOf(' - ') != -1
```

is called a *Boolean expression*. Boolean expressions are simply expressions that evaluate to `true` or `false`. The expression above evaluates to `true` if the value returned by the `IndexOf` method is not `-1` (remember, `!=` means "not equal to"). The expression only evaluates to `false` if the value returned by the `IndexOf` method is exactly `-1`, which means there isn't a hyphen in the last name. That means we can use

```
return lastName.IndexOf(' - ') != -1;
```

to do exactly what we want in our `Hyphenated` property.

You should also realize that we could have declared a `bool` field and set its value in the constructor as we did for the first name, middle initial, and last name fields. We decided to show you how we determine whether or not the last name is hyphenated "as needed" in this case instead.

The complete code for the `Name` class is provided in Figure 5.5.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace NamesApp
{
    /// <summary>
    /// Stores information about a name
    /// </summary>
    public class Name
    {
        #region Fields

        string firstName;
        char middleInitial;
        string lastName;

        #endregion

        #region Constructors
```

```

/// <summary>
/// Constructor
/// </summary>
/// <param name="fullName">the full name</param>
public Name(string fullName)
{
    // extract parts from full name
    int firstSpaceLocation = fullName.IndexOf(' ');
    firstName = fullName.Substring(0, firstSpaceLocation);
    middleInitial = fullName[firstSpaceLocation + 1];
    lastName = fullName.Substring(fullName.LastIndexOf(' ') + 1);
}

#endregion

#region Properties

/// <summary>
/// Gets the first name
/// </summary>
public string FirstName
{
    get { return firstName; }
}

/// <summary>
/// Gets the middle initial
/// </summary>
public char MiddleInitial
{
    get { return middleInitial; }
}

/// <summary>
/// Gets the last name
/// </summary>
public string LastName
{
    get { return lastName; }
}

/// <summary>
/// Gets whether or not the last name is hyphenated
/// </summary>
public bool Hyphenated
{
    get { return lastName.IndexOf('-') != -1; }
}

#endregion
}

```

**Figure 5.5. Name.cs**

Don't worry about all the syntax for the properties and the method; we'll cover that when we're designing our own classes from scratch. At this point, you should be able to understand the string processing parts of the code above.

Now we need to write our application class. Remember, we already know from our design step that the application class will create three `Name` objects and print the names and whether or not they're hyphenated. That makes it pretty easy to generate the application class code in Figure 5.6.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace NamesApp
{
    /// <summary>
    /// Prints out information about three names
    /// </summary>
    class Program
    {
        /// <summary>
        /// Prints name info
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            Name name0;
            Name name1;
            Name name2;

            // read names and create name objects
            Console.Write("Enter Name 1 (firstname mi. lastname): ");
            name0 = new Name(Console.ReadLine());
            Console.Write("Enter Name 2 (firstname mi. lastname): ");
            name1 = new Name(Console.ReadLine());
            Console.Write("Enter Name 3 (firstname mi. lastname): ");
            name2 = new Name(Console.ReadLine());
            Console.WriteLine();

            // display name info
            Console.WriteLine("{0}, {1} {2}.", name0.LastName,
                name0.FirstName, name0.MiddleInitial);
            Console.WriteLine("{0}, {1} {2}.", name1.LastName,
                name1.FirstName, name1.MiddleInitial);
            Console.WriteLine("{0}, {1} {2}.", name2.LastName,
                name2.FirstName, name2.MiddleInitial);

            // display hyphenated info
            Console.WriteLine("{0}, Hyphenated: {1}", name0.LastName,
                name0.Hyphenated);
            Console.WriteLine("{0}, Hyphenated: {1}", name1.LastName,
                name1.Hyphenated);
            Console.WriteLine("{0}, Hyphenated: {1}", name2.LastName,
                name2.Hyphenated);
        }
    }
}
```

```
        Console.WriteLine();  
    }  
}
```

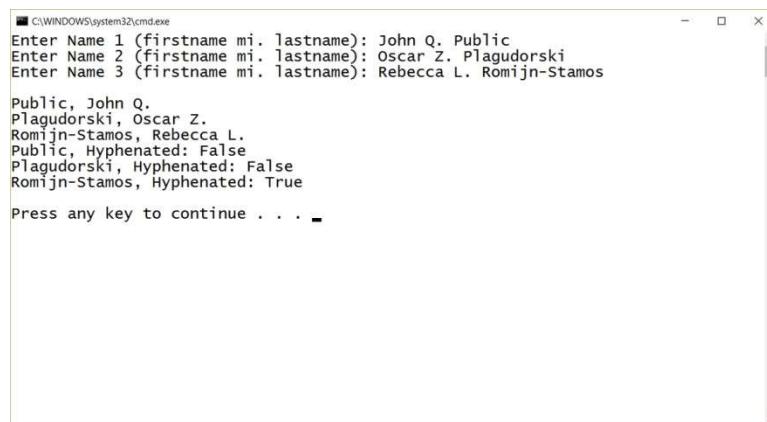
**Figure 5.6.** Program.cs

Before we move on to our testing, we should make a couple comments about the prompts we provide for user input. First, notice that we tell the user what format we expect them to use when they enter the names. There's no reason for us to assume the user knows what format we expect, so we need to explicitly tell them in the prompt.

Second, notice that our prompts ask for Name 1, Name 2, and Name 3 even though our variables are called `name0`, `name1`, and `name2`. That's because normal people (that's right, not game programmers, NORMAL people!) start counting at 1, not 0. The interface that the users interact with, both in games and in other software, should reflect how they think, not how programmers think.

### *Test the Code*

Finally, we run our test case. The actual results are provided in Figure 5.7.



**Figure 5.7. Test Case 1: Checking Name Info Results**

This is exactly what we expected.

## 5.5. Common Mistakes

## *Forgetting That Character Locations Are Zero-Based*

The character locations in a `string` are numbered starting with 0, not 1. If you forget this, you'll get unexpected results as you process your strings. You'll see and use zero-based indexing throughout your programming career, so now is the time to really learn it well!

## Chapter 6. Unity 2D Basics

In 2005, Unity Technologies released the Unity game engine. Although it was initially limited to OS X development, it's now a true cross-platform game engine, supporting a wide range of deployment platforms. The Personal Edition is free to individuals and companies making less than \$100,000 a year, and it's an extremely popular development platform for indie game developers.

As if that weren't enough, the most popular Unity scripting language for commercial game development is C#. So, we have a robust, free, popular game development tool that uses C#. That makes it the obvious choice for the game development examples that form the bulk of the material in this book.

As we said in the preface, this book isn't designed to teach you all the ins and outs of Unity; our focus is on the actual C# scripting you do to build Unity games. With that said, though, we've already used some of the important features provided in Unity in previous chapters, and we're going to use even more before we're done, so it makes sense to go over some of the core concepts in Unity in this chapter. We'll limit ourselves to 2D games in this book, but Unity most definitely supports building fantastic 3D games as well.

### 6.1. The Worst Game EVER

Just to drive you into a frenzy of game development madness, where all you want to do is drink Mountain Dew, eat Twinkies, and sling code until your fingers bleed<sup>17</sup>, we'll start by building the worst game EVER.

Okay, start up Unity, click New near the upper right corner, select 2D, pick a location and name for your project, then click the Create project button. When the Unity editor starts up, use the layout dropdown near the upper right to set the layout to the one you want to use. Right click in the Project window, create a new scenes folder, then save the current scene as scene0 in that folder.

When you click the play button in the middle near the top, you get the amazing blue screen of nothing happening in the Game view. If you want to see something really exciting, click the Maximize on Play option at the top right of the Game view and play the game again. Amazing, huh?

This truly is the worst game ever, but it is the starting point for all our 2D Unity games. By default, we get a Main Camera included in our scene, since without a camera we won't be able to see anything when we run our game!

So why is this the worst game ever? Because there's no player interaction with the game world, and that's what games are all about!

The bad news is that to have player interaction we need to use Unity's input capabilities, which are covered in Chapter 8, not here. The good news is that we can start learning about how Unity works in this chapter by providing some interesting graphical output in our game.

---

<sup>17</sup> While the author has never coded until his fingers bled, there is a vicious rumor that he once played the arcade version of Track and Field until one of his fingers bled. Sadly, the vicious rumor is in fact true.

## 6.2. A Better Not A Game (NAG)

Let's build a slightly better Not A Game (NAG) than our worst game ever. For this NAG (though we'll lie and call it a game throughout the section), we'll focus on drawing a few stationary teddy bears in the game.

Create a new Unity 2D project and save the current scene into a new scenes folder in the Project window. Right click in the Project window and create another new folder called sprites. Now go to your operating system and copy the teddy bear png files from the code for this chapter into the sprites folder (which you'll find under the Assets folder wherever you saved your new Unity project). When you copy the png files into the sprites folder, Unity automatically imports them as sprites in your project.

Let's take a look at the Import Settings for one of the sprites. When you click teddybear0 in the sprites folder in the Project window, the Inspector should look like Figure 6.1.



**Figure 6.1. teddybear0 Import Settings**

There are lots of Texture Types we can select when we import a .png (and other formats) image into Unity, but the default for a 2D game is as shown, and we have to use that setting to use the texture as a Sprite in our game.

The two possible sprite modes are Single and Multiple. Single means we're importing a single image and Multiple means multiple images are included in the png. The most common use of Multiple is if we're importing a sprite strip (also called a sprite sheet) for an animation. We'll see how to use that setting when we import an animation later in the book.

The Packing Tag is really for more advanced users, so we won't worry about it here.

Pixels per Unit tells how many pixels in the image correspond to one distance unit in world space in the game. The best approach to use here is to figure out the Pixels per Unit you want to use and to use that for EVERY art asset you generate for your game. The easiest thing to do is to just use the default 100 pixels per unit. One of the things you'll need to do in some of your games – we'll need to do it when we solve the more advanced circles problem with text displayed in-game – is convert from a coordinate in

the game world coordinate system (in Unity distance units) to a coordinate in the screen coordinate system (in pixels). Unity needs to know the "conversion factor" to use for each sprite, so that's why we need to provide this information here.

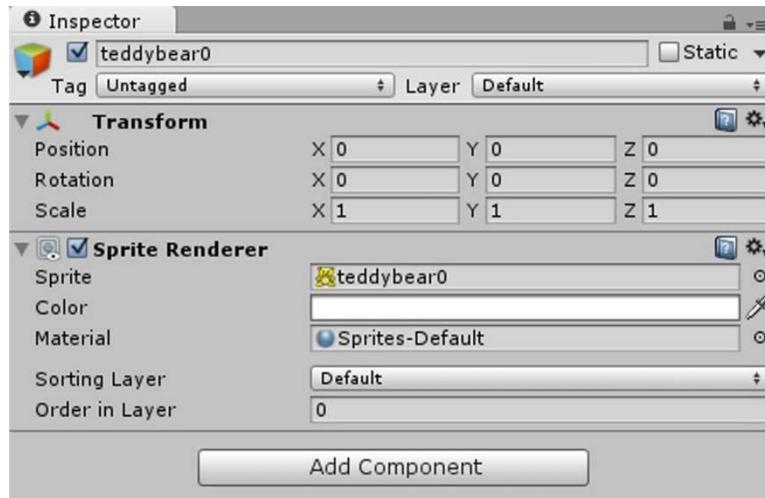
Pivot tells where the origin of the local coordinate system for the sprite is. In our game development experience, having the origin at the center of the sprite makes the most sense, especially if you ever need to rotate the sprite. Try doing the math to rotate a sprite properly when the origin is in the upper left corner and you'll see what we mean! You do have lots of options here, though, including setting a custom pivot for the sprite.

The rest of the items in Figure 6.1. are less commonly changed, so we won't discuss them here. You can go to the Unity Manual and search on 2D Textures for more information if you need it.

By the way, the dimensions for the textures you import into Unity should always be in powers of two (32 by 32, 64 by 128, and so on). They don't have to be square, but the width and the height should each be a power of two. Although the Unity documentation doesn't explicitly say why, this is a fairly common requirement to make efficient use of memory on the graphics card.

So we have a sprite imported into our project; what do we do with it? One of the most common things to do is just drag it into the scene. When you do, the sprite becomes a game object in the scene. That's actually what we're going to do next for our game.

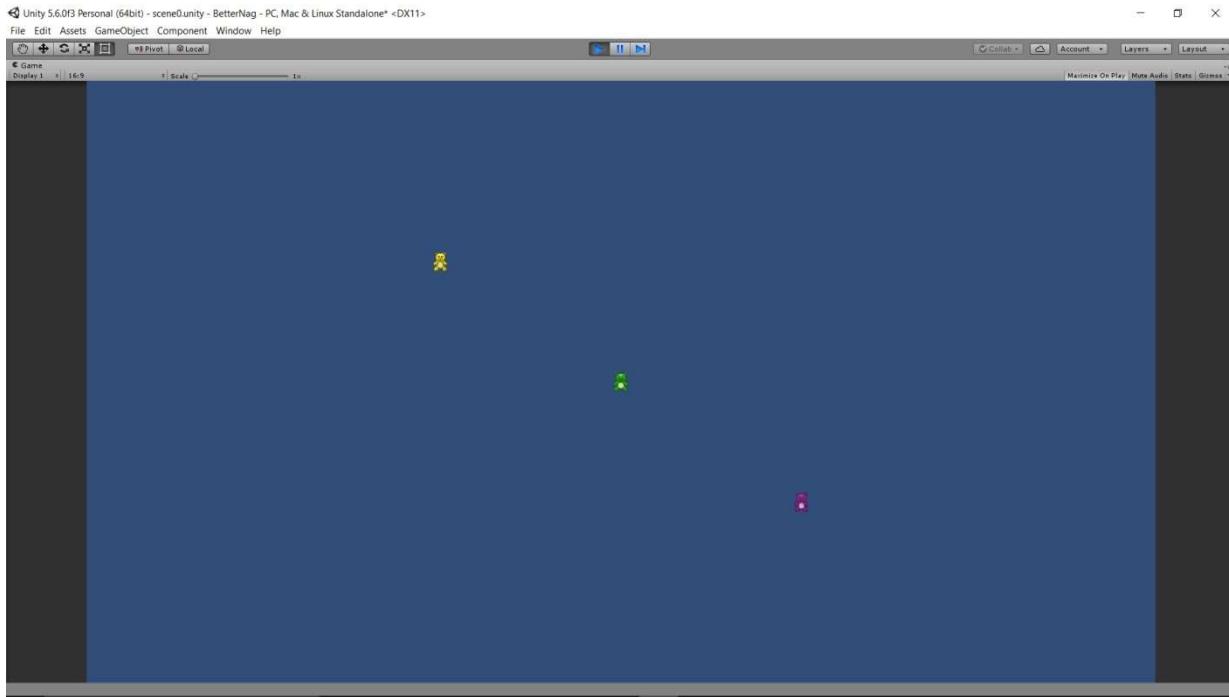
Drag and drop the teddybear0 sprite from the Project window into the Hierarchy window; the Inspector should look like Figure 6.2. As we mentioned previously, all Unity game objects get a Transform component by default. This makes great sense, of course, because every object in the game has a position, rotation, and scale. All Sprite game objects also get a Sprite Renderer by default, because we'll almost always want our sprites to be rendered (drawn) in the game world.



**Figure 6.2. Inspector for Teddy Bear 0 Game Object**

Change the X Position value in the Transform component to -3 and change the Y Position value in the Transform component to 2. As you can see from the new location of teddy bear 0 that the camera is pointed at the origin of the game world, with the X-Y coordinate system behaving as usual. You should note that the values we enter in the Transform component are in world coordinates (Unity distance units), not in pixels.

Now we create game objects for the other two teddy bear sprites by dragging the Sprites from the Project window into the Hierarchy window (or Scene view) and adjusting each of their Transform components to space the teddy bears in a reasonable way. When we did that and ran the game, we got the output shown in Figure 6.3.



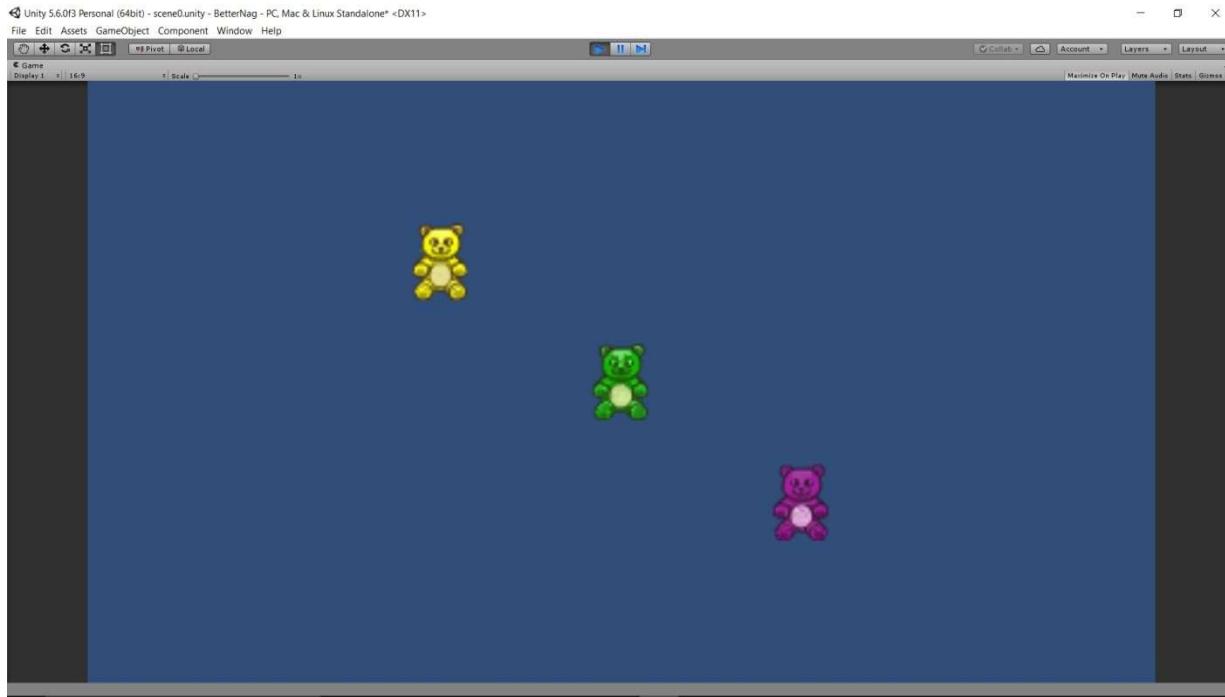
**Figure 6.3. Initial Teddy Bear Output**

Hmmm. Although it's nice to see our 3 teddy bears in the game, they seem to be pretty small. What should we do about this?

The best approach to take is to go back to your artist and have them create larger art assets for you. In fact, one of the first things you should do when you start working on the art for a game is to figure out the appropriate sizes for your art assets to make sure they're all big enough. When in doubt, you should always err on the side of art assets that are larger than you think you'll need. When we shrink art assets, we lose some detail, which is often fine, but when we expand art assets past their original size, it's usually not fine.

Let's say, though, that you're writing a book and the artist who made your art assets is no longer available (we know a friend this happened to <grin>)? You could find another artist, but let's say you're constrained to use the art assets you have. What can we do?

Well, we talked earlier about the Pixels per Unit setting in the Import Settings; couldn't we just change those to 25 (say) to make the teddy bears larger in the game? If you try that out, you'll get something like Figure 6.4.

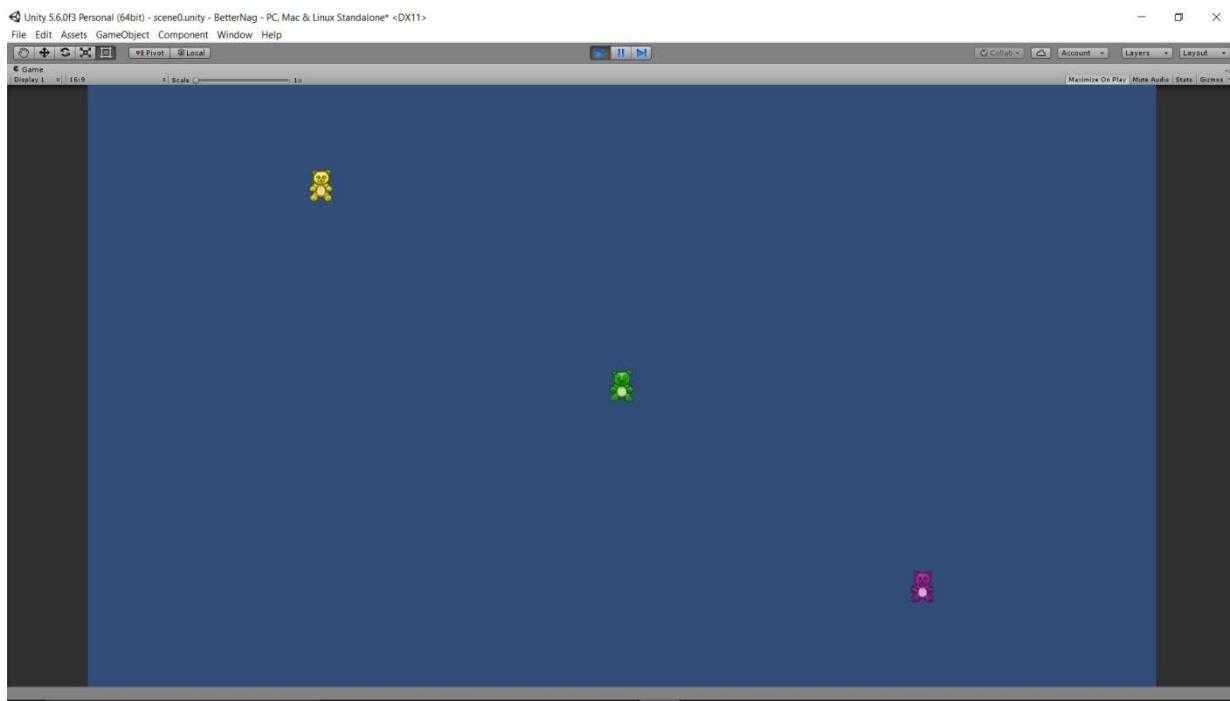


**Figure 6.4. Teddy Bear Output with Modified Pixels per Unit**

As you can see, we do end up with larger teddy bears, but they're kind of blurry. Changing the X Scale and Y Scale values in the Transform component also makes them large and blurry. We should also note that both these approaches also make the teddy bears larger in the game world, which may not be what we want.

Remember, though, that the game world is rendered based on the view provided by the Main Camera in our scene. How do you get a better picture of something that seems too small in your camera's preview screen? One thing you can do is zoom in on the objects to make them appear larger.

We can essentially do that in our Unity scene by changing the size of the camera. If you change the camera size from the default size (5) to 3 instead, you'll get something like Figure 6.5. The teddy bears are larger in the rendered display while still retaining their original crisp detail. You should realize this doesn't change the size of the game objects within the game world, it just makes them appear to be larger because we've basically zoomed in on them.



**Figure 6.5. Teddy Bear Output with Smaller Camera Size**

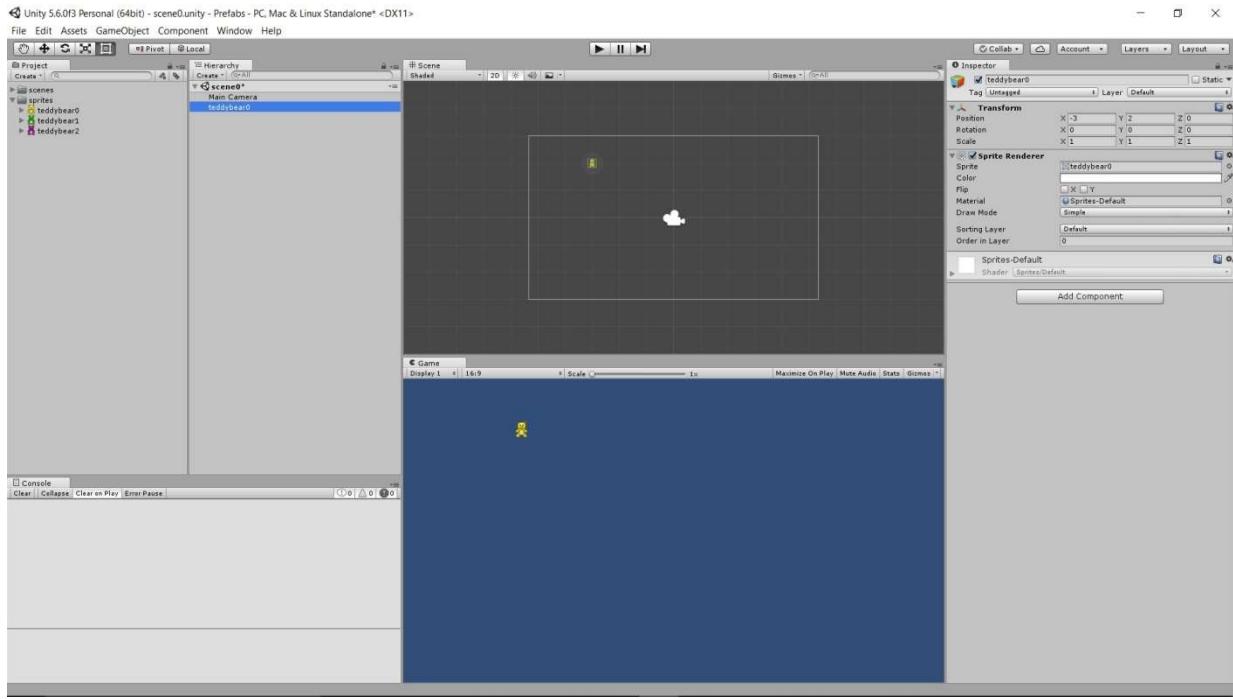
The moral of the story here is that the absolute best thing to do is to just get art assets that are the appropriate size for your game from your artist. If you can't do that, though, there is at least one reasonable way to handle art assets that are actually smaller than you need.

### 6.3. Prefabs

One of the things that we'll find to be really useful in Unity is something called *prefabs*. You can think of a prefab as a template for creating as many instances of a game object, including all its properties and components, as we need in the scene. This is useful when we're placing objects in the scene in the Unity editor, and it's also just what we need when we need to spawn new objects as the game runs.

In this section, we'll walk through the process of creating and using prefabs to refactor the game from the previous section to use prefabs instead. Although we won't really see much benefit from doing that for this particular game, it will give us practice with prefabs. We'll see more benefit from using prefabs in the following sections of this chapter (and throughout the rest of the book as well).

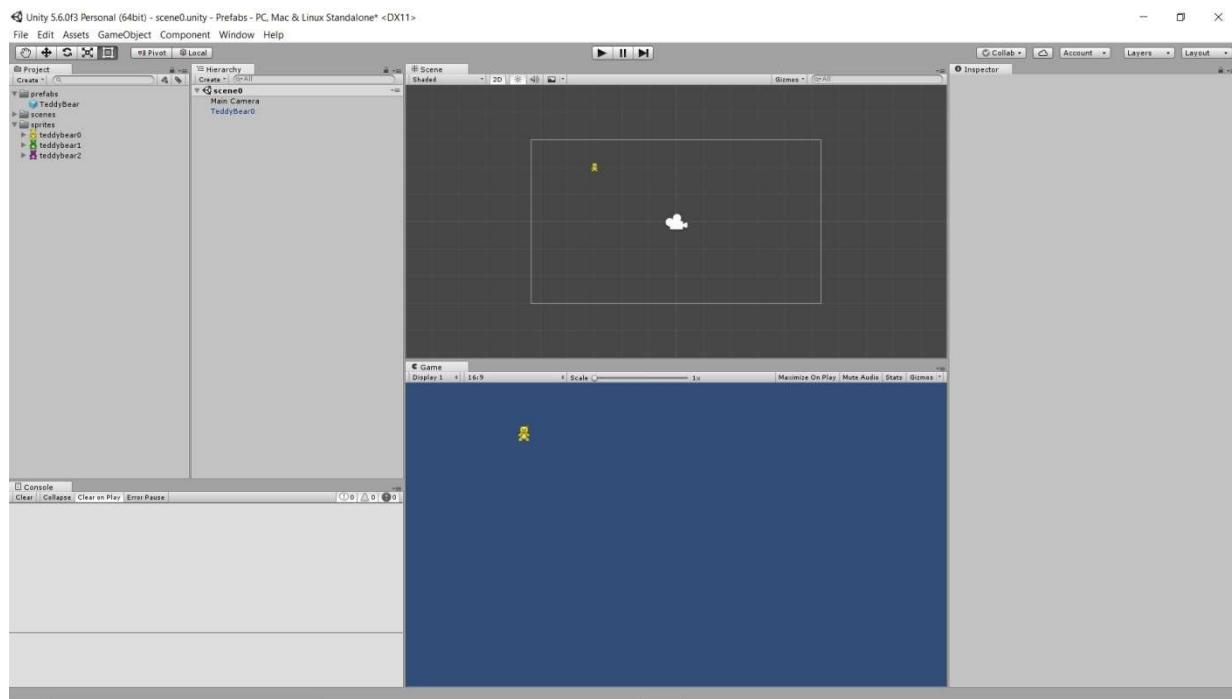
Our starting point for our solution in this section is our solution from the previous section with only the first teddy bear placed in the scene; see Figure 6.6.



**Figure 6.6. Prefab Starting Point**

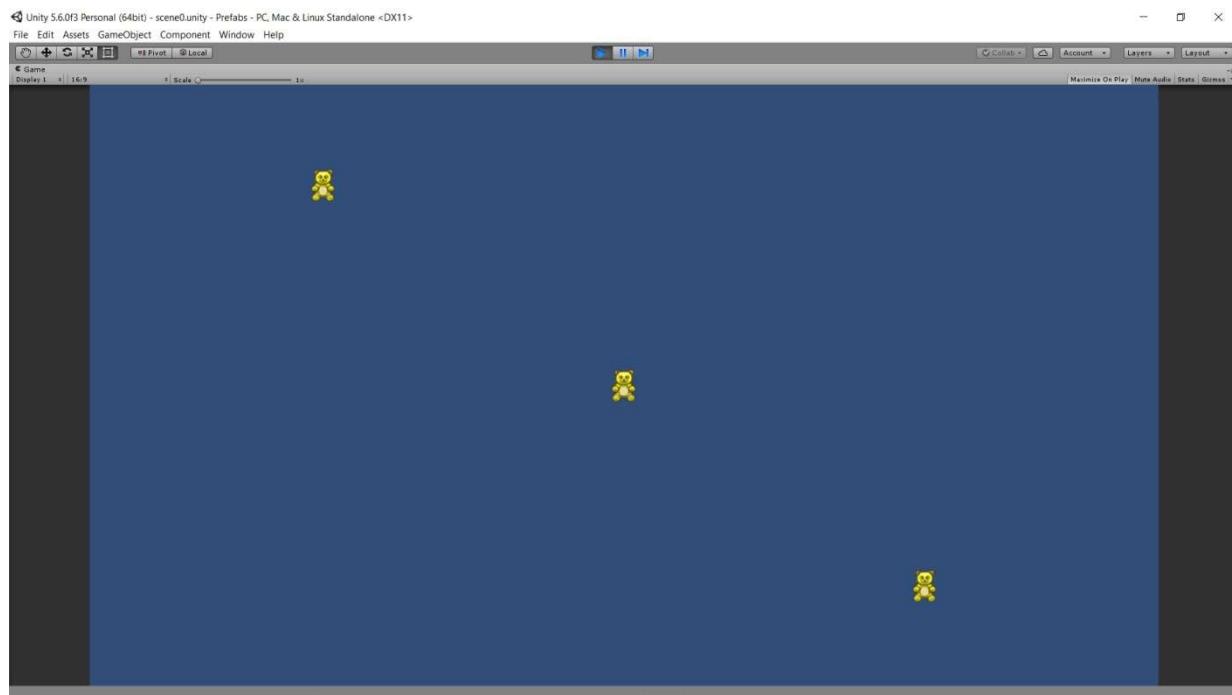
Here's where we do something new. Right click in the Project window and create a new folder called `prefabs`. Now drag `teddybear0` from the Hierarchy window into the new `prefabs` folder in the Project window.

As you can see, a couple things happen when we do this. First, we get a prefab called `teddybear0`, with a blue "prefab cube" next to it, in the Project window. Second, `teddybear0` in the Hierarchy window turns from black text to blue text to indicate that the `teddybear0` object in the scene is an instance of a prefab. We don't really want our prefab to be named `teddybear0`, because it's really our prefab for all our teddy bears, so rename it to `TeddyBear` instead. This actually changes the name of the instance in the Hierarchy window as well, so rename that instance to `TeddyBear0`. Your Unity editor should now look like Figure 6.7.



**Figure 6.7. TeddyBear Prefab Created**

Now, drag the TeddyBear prefab from the Project window into the Hierarchy window twice, name the new instances TeddyBear1 and TeddyBear2, and change their (X, Y) locations in the game world to (0, 0) and (3, -2) by changing the X and Y Position values in their Transform components. Run the game, and you'll get Figure 6.8.



**Figure 6.8. Three TeddyBear Instances**

Well, this close to what we want, but we did want to use different sprites for each of the teddy bears. Let's fix that now.

Select the TeddyBear1 instance in the Hierarchy or Scene view and look at the Inspector. You may not have thought about this explicitly when you were changing the Transform of this game object to be located at (0, 0) in the game world, but it's important that you realize that you were only editing this instance of the prefab, you weren't changing the prefab itself. That's why all the instances in the scene didn't jump to (0, 0), only the one you were editing did.

We can actually edit any of the available characteristics of our instances in the scene, not just their Transform component values. That's good for us here, because each of the instances also has a Sprite Renderer component. Remember how our `SetSprite` method in our `Card` script in Chapter 4 changed the `sprite` field of the `SpriteRenderer` component based on whether the card was face up or face down? Although we did that at run time from within our script rather than at design time in the Unity editor, it does tell us that changing the `sprite` field changes what sprite is actually displayed. That's just what we need here.

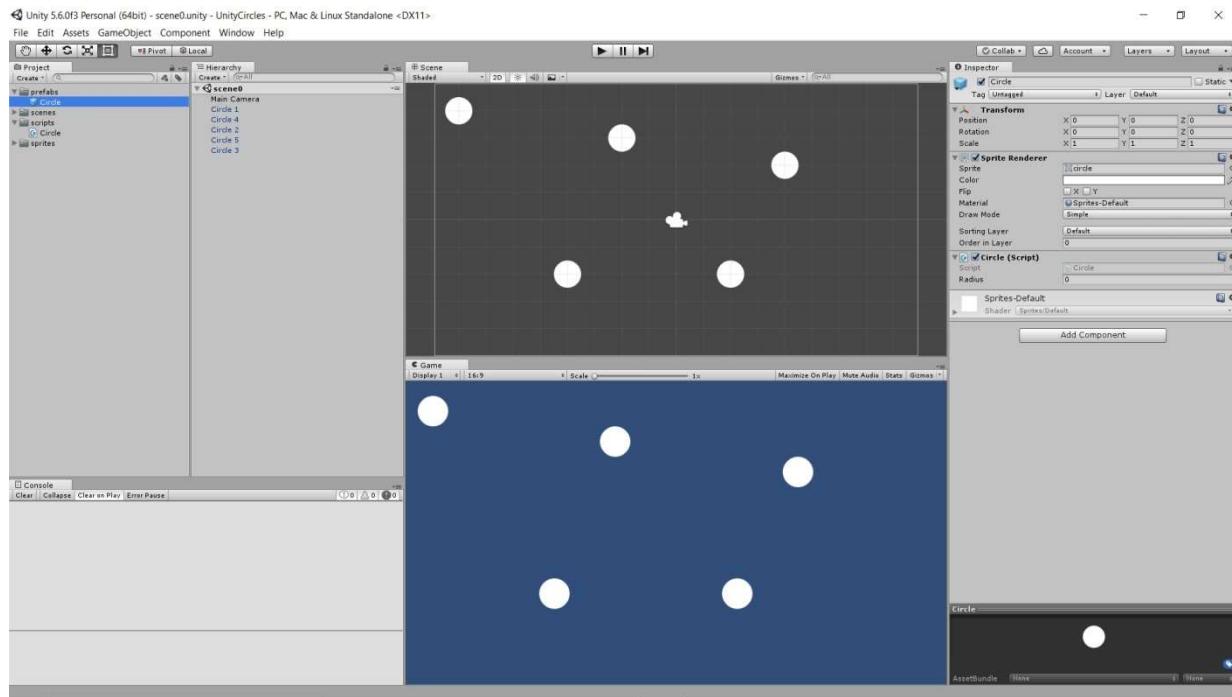
Drag the `teddybear1` Sprite from the `sprites` folder in the Project window onto the `Sprite` value in the `Sprite Renderer` component in the Inspector. Now select the `TeddyBear2` object in the Hierarchy window and drag the `teddybear2` Sprite from the `sprites` folder in the Project window onto the `Sprite` value in the `Sprite Renderer` component in the Inspector. As you can see in the both the Scene and Game views, this changes the sprites that are rendered for those two game objects.

This probably seemed like a lot of extra work to get to the same output as we had in the previous section, but prefabs will be a common and critical part of your Unity games, so changing our project to use prefabs was an important step in your Unity learning.

## 6.4. Unity Circles Revisited

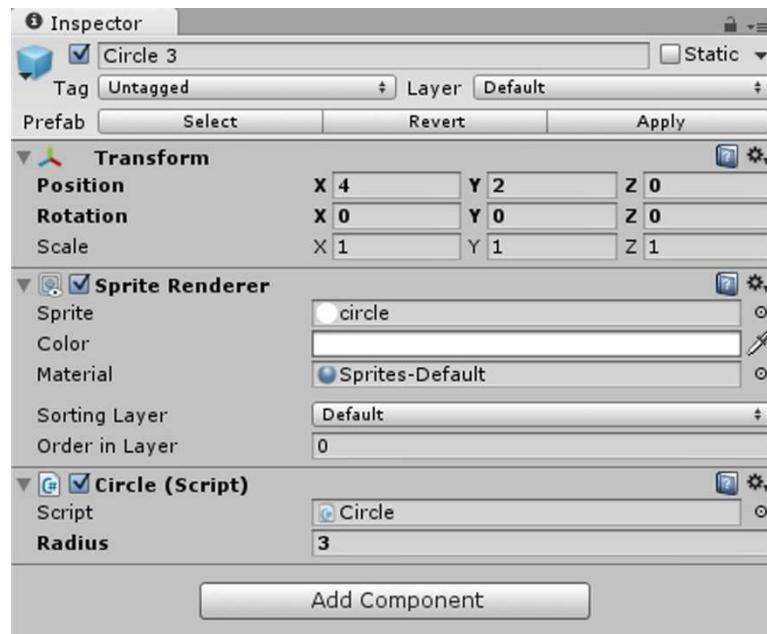
Now that we understand a little more about how Unity works, let's go back and look a little more closely at the structure of our Unity project for the Circles problem from Chapter 4. The Unity editor for this project, with the `Circle` prefab selected in the Project window, is shown in Figure 6.9.

As you can see in the Inspector, our `Circle` prefab has the expected `Transform` and `Sprite Renderer` components, and also has our `Circle` script attached as a component. For this problem, we didn't need to change the sprite that was displayed for each instance, because they all display a white circle, but they did need to be different radii. Even though all the circles are the same size in the Scene and Game views, the script actually changes the size of the game object at run time based on the value of the `radius` field in the script.



**Figure 6.9. Circle Prefab**

Select the Circle3 game object (which is an instance of the Circle prefab) in the Hierarchy window and look at the Inspector, shown in Figure 6.10.



**Figure 6.10. Circle 3 Game Object**

As you can see, we changed the X and Y Position values in the Transform component to move the Circle3 game object to a different location, but we also changes the Radius in the `circle` script component to make this game object have a radius of 3 at run time.

So what did using prefabs do for us in this case? At a lower level, using them meant we didn't have to manually attach the `Circle` script to each game object we placed in the scene because that script was already attached to the prefab. At a higher level, we followed Unity best practices because when we have multiple game objects in the scene that are all the same type of game object (circles with certain behavior, in this case), they should all be instances of the same prefab.

## 6.5. Putting It All Together

Let's make our NAG more interesting; here's the problem description:

Move three teddy bears around the screen until the player quits the game.

### *Understand the Problem*

This seems pretty straightforward, but you should have at least one question. What should we do when a teddy bear reaches the edge of the screen? There are lots of possibilities here – we could have the teddy bear bounce off the edge of the screen, warp to the other side of the screen, explode and then re-spawn on the screen, and so on. Since the problem description doesn't tell us which method is preferred, we'd like to bounce the bears off the edges of the screen.

You should also be asking what should happen when two teddy bears run into each other. There are lots of possibilities here as well, but let's have them bounce off each other when they collide.

Because we're going to use our problem description for our testing, we need to revise the problem description to capture these new details. Here's the revised problem description:

Move three teddy bears around the screen, bouncing them off the edges and each other, until the player quits the game.

By the way, you as the programmer will almost NEVER get to change the problem description at your own discretion! Instead, the typical process would be to discuss your recommended changes with whoever "owns" that problem description (typically, the person who gave you the problem to solve) and negotiate the appropriate changes to the description. Since the author owns the problem description above, though, we negotiated with ourselves<sup>18</sup> and agreed to the revised wording.

### *Design a Solution*

It actually turns out that we're going to be able to solve most of this problem using Unity's 2D physics system, with a pretty simple `TeddyBear` script that we use to start each teddy bear moving! Let's defer the details of how to do that to the Write the Code step so we can iteratively get it all to work.

### *Write Test Cases*

This program doesn't have any user input, so we'll just run it and make sure the teddy bears move around the screen and bounce when they're supposed to.

---

<sup>18</sup> Talking to yourself is a perfectly normal part of programming, so don't be alarmed if you find yourself doing so. It's also normal to talk to your computer in either a pleading or threatening tone depending on your current mood.

## Test Case 1: Checking Bear Behavior

Step 1. Input: None.

Expected Result: Bears move around screen and bounce off edges and each other

The problem description never addressed how the teddy bears should move, so we'll start each teddy bear moving in a random direction at a random speed when it gets added to the scene at run time. You might be wondering about testing the randomness when we run the program. Because we're going to make each teddy bear move in a random direction with a random speed, don't we have to make sure those are actually random?

The answer is no because the problem description didn't say how the bears were supposed to move. We decided as part of our problem-solving process that we'd have them move randomly, but we could just as easily have chosen some other way to make them move. Functional tests compare the program's behavior against the program description (often called the *requirements*), so our functional tests don't have to test behavior that's not specified in the problem description.

### Write the Code

We start by creating a new Unity 2D project, saving the scene into a scenes folder, importing our 3 teddy bear sprites into a sprites folder, creating a `TeddyBear` script in a scripts folder, and creating an empty prefabs folder. We also drag the teddybear0 Sprite from the Project window into the Hierarchy window.

We'll get our solution working in a number of smaller steps rather than trying to get everything working all at once. This is exactly the way we actually program in general, and it's also the way we actually develop games in practice. For this solution, we'll get our teddy bear moving first, then we'll get it bouncing off the sides of the screen, then we'll save it as a prefab, then we'll add the remaining two teddy bears and make sure the teddy bears bounce off each other properly.

Recall that we plan to use the built-in Unity 2D physics capabilities to implement most of the required functionality; that includes helping to get the teddy bear moving. To get our teddy bear to obey the laws of physics, we need to add a Rigidbody2D component to the teddy bear game object. Select teddybear0 in the Hierarchy window and click Add Component at the bottom of the Inspector. Click Physics 2D > Rigidbody 2D to attach the required component.

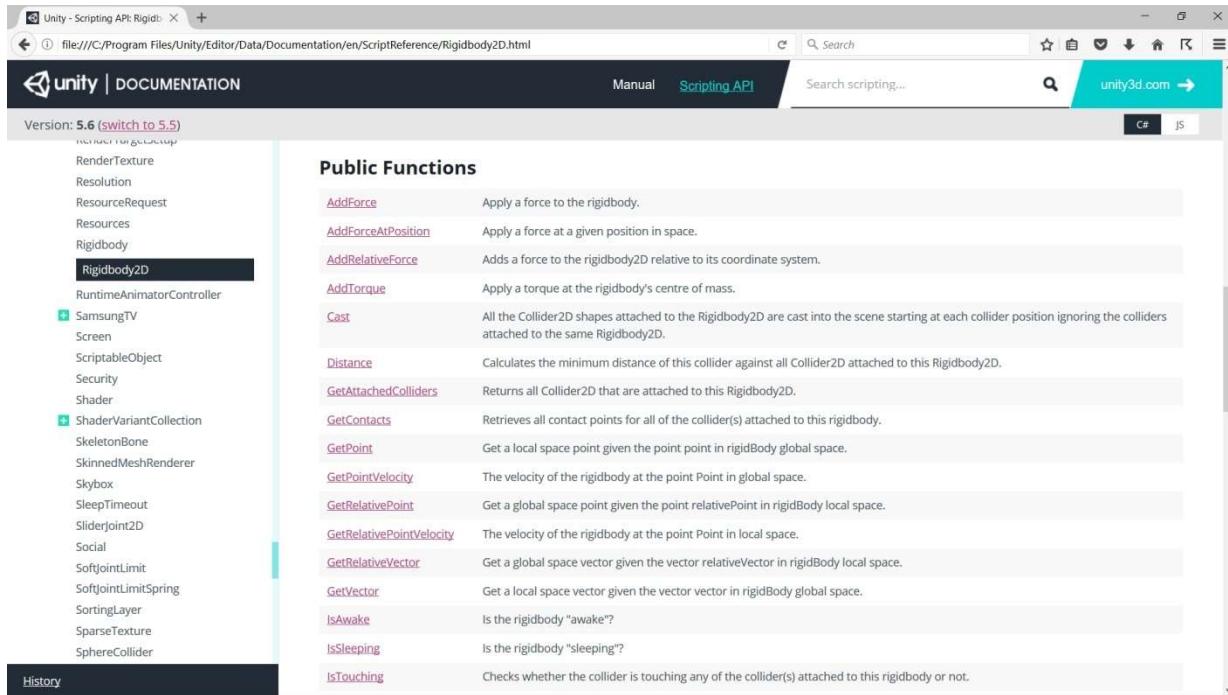
If you run the game at this point, you'll see the teddy bear fall down and off the screen! That's because it's now obeying the laws of physics, and the default for 2D games in Unity is that we have gravity in the negative y direction. That's really helpful in a lot of cases, but we don't have gravity in this game, so we need to turn it off. We can do that by selecting Edit > Project Settings > Physics 2D from the top menu bar in the Unity editor, then setting the Y component of gravity to 0 in the Inspector. If you run the game again, you'll see that the teddy bear no longer falls.

Okay, it's time to get the teddy bear moving when the game starts. You should realize by now that we'll do that in the `Start` method of the `TeddyBear` script, which we attach to the game object in the usual way. Go ahead and open up the `TeddyBear` script in Visual Studio and delete the `Update` method.

We're ready to get the teddy bear moving, but how do we do that in the `Start` method? You'll find throughout your game development career that thinking about the way the real world works will often

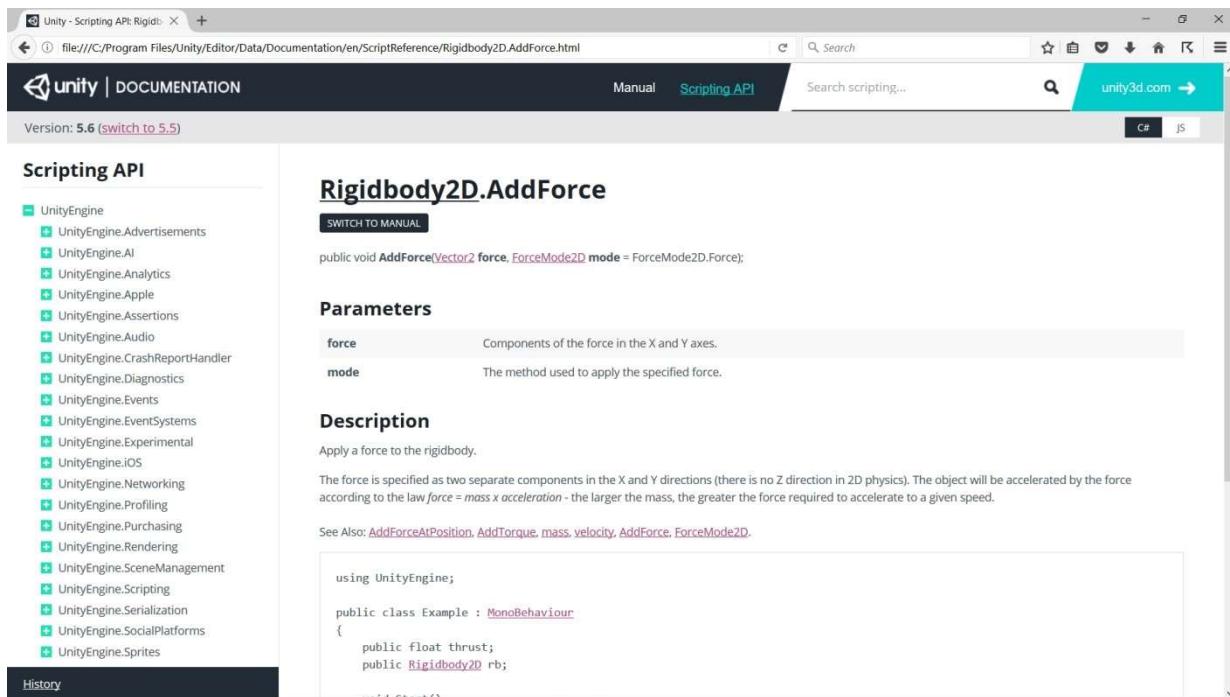
help you figure out how to make things work in your game world. So let's rephrase the question to "How do we get an object moving in the physical world"?

The answer, of course, is that we apply a force to the object, with a particular magnitude, in the direction we want it to move. We know we added a Rigidbody 2D component to our game object so it would do physics things, so maybe we can apply a force to that component. How do we find out? The Unity Scripting Reference, of course! In the Unity editor, select Help > Scripting Reference from the menu bar, search on Rigidbody 2D, then click the Rigidbody2D link in the search results. We know we need to access a behavior of the class, so scroll down to the Public Functions section (see Figure 6.11).



**Figure 6.11. Rigidbody2D Documentation**

Wow, it looks like the very first method (`AddForce`) is what we need (remember, we'll call them methods in C# even though the Unity documentation calls them functions)! Click the `AddForce` link to get Figure 6.12.



**Figure 6.12. AddForce Documentation**

Okay, so we'll retrieve a reference to the `Rigidbody2D` for the teddy bear and call the `AddForce` method with `Vector2` and `ForceMode2D` arguments. We'll talk about the `Vector2` now and the `ForceMode2D` soon.

So how do we generate the random direction and speed for the bear? The big idea is that we need to generate a random direction and a random magnitude for a force to apply in that direction. To get the random direction, we'll start by generating a random angle between 0 and  $2\pi$  radians. We can then use that angle to create a unit vector, which is simply a vector with a magnitude (length) of 1, by using cosine to generate the x component and sine to generate the y component of the vector. Finally, we can multiply the vector by a random magnitude to get our final force vector.

It turns out that Unity provides a very useful `Random` class we can use to get the random numbers we need. If you read the documentation for this class, you'll find it exposes a static `Range` method that returns a `float` between min and max values we provide as arguments to the method. We can use this method to generate our random angle, and we can also use this method to generate our random magnitude. Finally, actually applying the force will get the teddy bear moving in a random direction with a random speed, which is just what we want.

We now have all the pieces we need to implement our script; see below.

```

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

/// <summary>
/// A teddy bear
/// </summary>
public class TeddyBear : MonoBehaviour

```

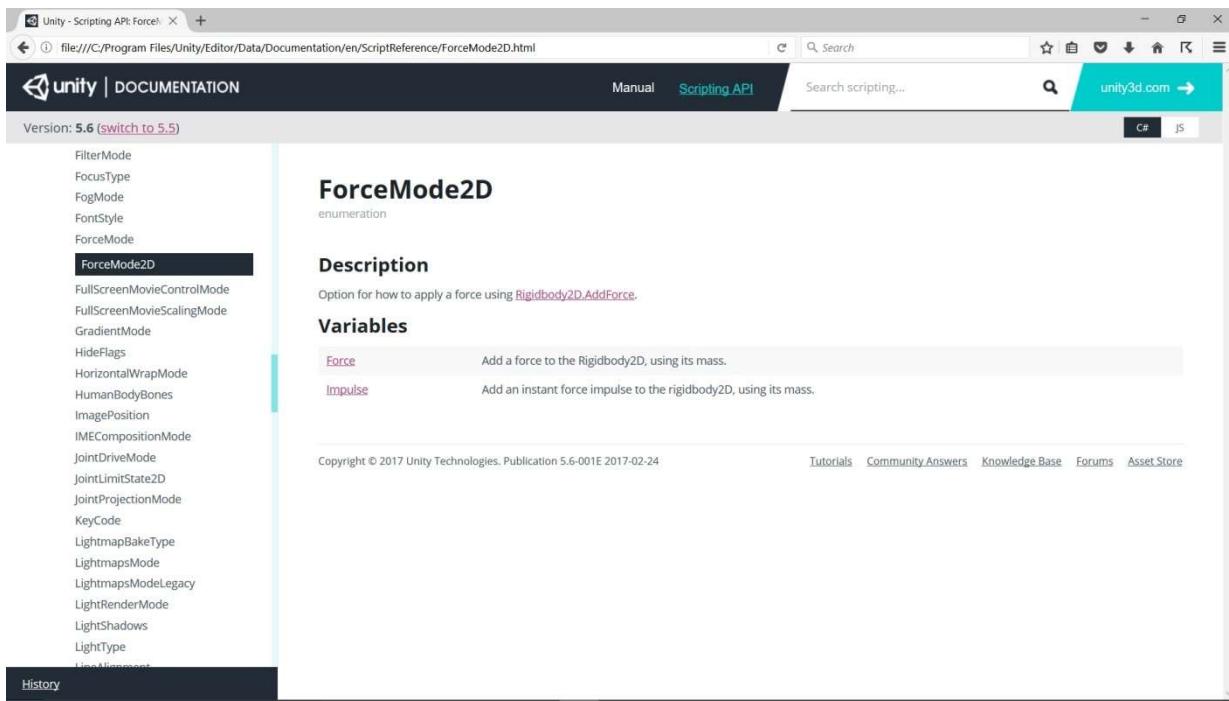
```

{
    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
    {
        // apply impulse force to get teddy bear moving
        const float MinImpulseForce = 3f;
        const float MaxImpulseForce = 5f;
        float angle = Random.Range(0, 2 * Mathf.PI);
        Vector2 direction = new Vector2(
            Mathf.Cos(angle), Mathf.Sin(angle));
        float magnitude = Random.Range(MinImpulseForce, MaxImpulseForce);
        GetComponent<Rigidbody2D>().AddForce(
            direction * magnitude,
            ForceMode2D.Impulse);
    }
}

```

Let's discuss the details of the last line of code in the `Start` method. The `GetComponent<Rigidbody2D>()` piece returns a reference to the `Rigidbody2D` component for the game object, then we call the `AddForce` method on that `Rigidbody2D` object. The first argument to the method call is the force vector we generate as discussed above. We're actually multiplying a `Vector2` by a `float` in that argument, but that's okay. When we multiply a vector (`direction`) by a scalar (`magnitude`), we just multiply each component of the vector by the scalar to get our new vector.

The second argument to the method call might look a little strange to you because it uses something called an *enumeration*. An enumeration essentially defines a new data type with a specific set of values. Take a look at the `ForceMode2D` documentation shown in Figure 6.13 (which you could have gotten to by clicking the `ForceMode2D` link in Figure 6.12). The documentation shows that variables and (more importantly in our current case) arguments that are declared to be of type `ForceMode2D` can have one of only two values: `ForceMode2D.Force` or `ForceMode2D.Impulse`. We always need to precede the value with the name of the enumeration. We decided to use `ForceMode2D.Impulse` in this case – if you want to, think of this as whacking the bear with something to get it moving!



**Figure 6.13. ForceMode2D Documentation**

If you run the game now, you'll see the teddy bear moves in a particular direction at a particular speed. You'll need to start and stop the game a few times to see that both the direction and the speed are random.

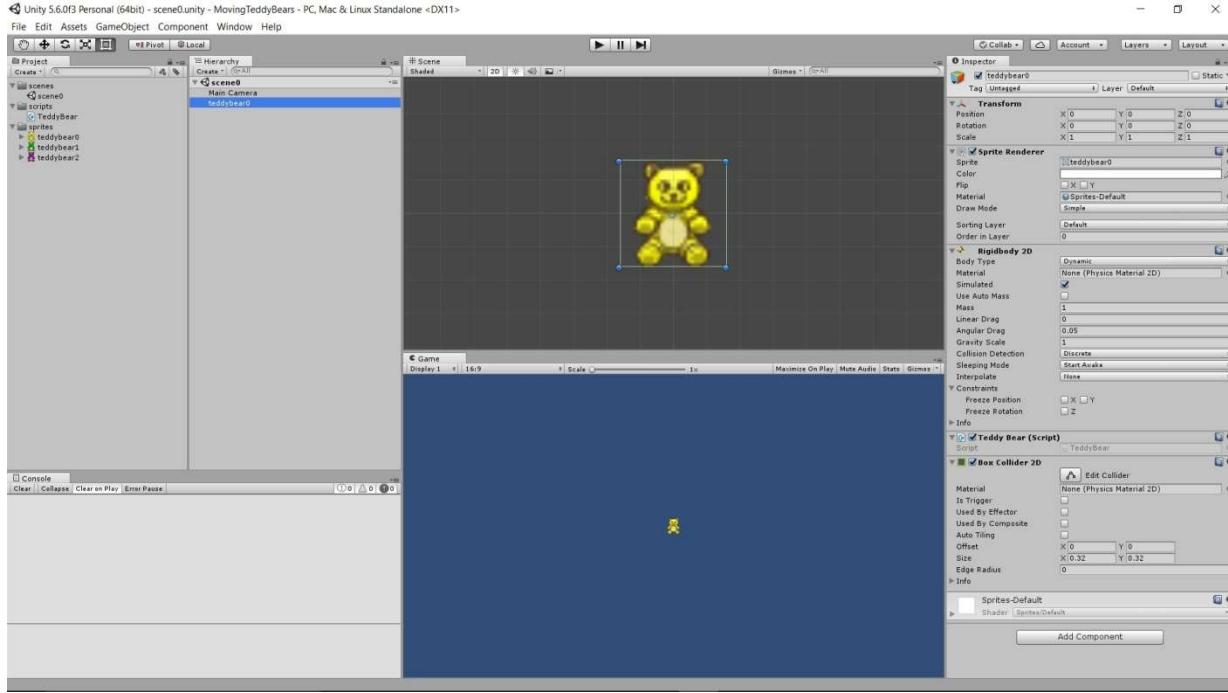
On some computers, the movement of the teddy bear might seem a little "jerky" in some cases. There are a number of ways to address this. If you look at the teddy bear's Rigidbody 2D component in the Inspector, you'll see there's a dropdown box next to an Interpolate label. Selecting Interpolate instead of None (the default) for that could help. The None selection takes less processing power – that's why it's the default – but it's certainly reasonable to select Interpolate instead if you need to.

It's also sometimes the case that the "jerky" motion shows up in the Unity editor but doesn't in the built version of the game. To check this, you can select File > Build Setting in the Unity editor then pick the options you want and click the Build button to actually build an executable of your game. Then all you have to do is double-click the executable in your operating system to run your built game.

Whew, that was our first small step! As you can see, the teddy bear doesn't bounce off the sides of the screen, so we'll add that next. To do that, we need to add a collider component to the teddy bear game object so it can run into stuff in the game world.

We actually have four choices for the kind of collider we want to add: Circle Collider 2D, Box Collider 2D, Edge Collider 2D, and Polygon Collider 2D. These are listed in order of efficiency for collision detection, so using a Circle Collider 2D on everything would yield a much better-performing game than using a Polygon Collider 2D on everything. Polygon Collider 2Ds are great for complex shapes, but we can also add multiple colliders to a game object, so combining a number of circle and box colliders to approximate an object's shape can be a more efficient approach (for CPU processing) without adversely affecting game play. For our teddy bear game object, we'll add a Box Collider 2D because that will work well enough for our bouncing around in this game.

Select the teddy bear game object, click the Add Component button in the Inspector, and select Physics 2D > Box Collider 2D; you should end up with something like Figure 6.14 (you might need to double-click the game object in the Hierarchy window to zoom in on it in the Scene view).



**Figure 6.14. Initial Box Collider 2D**

You can probably tell right away that the collider is going to be too wide to get nice collisions. Unity defaults the box collider to the size of our Sprite, but we added some transparency on the left and right of our original teddy bear image to make the width of the png a power of 2 (32 pixels, in this case).

To tighten up the collider on the left and right, click the button to the left of Edit Collider in the Box Collider 2D component. You can now drag the edges of the collider to where you want them. When you're done, click the Edit Collider button again. You can also edit the collider by changing the Offset and Size values for the component, but we usually find it easier to drag the collider edges.

Okay, now we have a collider attached to the teddy bear game object, but there are no other colliders in our game for this collider to collide with. Next, we'll add Edge Collider 2Ds at the top, bottom, left and right of the screen so the teddy bear game object bounces off the edge of the screen. The screen that we see as we play is determined by our Main Camera, so we'll add these 4 components to the Main Camera.

Double-click the Main Camera in the Hierarchy or Scene view. The box around the camera in the Scene view shows the edges of the screen when we run the game, so that's where we want to place our edge colliders. You may need to zoom in to get a better view of the box; to do that, use the scroll wheel on your mouse. You can pan around the Scene view by holding down Alt and the left mouse button, then moving the mouse around.

Okay, let's add the edge collider on the top edge of the box. Click the Add Component button in the Inspector and select Physics 2D > Edge Collider 2D. This adds the edge collider, but it's obviously in the wrong place. Click the button to the left of Edit Collider in the Edge Collider 2D component; you can

now drag each end point of the collider to where you want them. When you're done, click the Edit Collider button again. Add the edge colliders for the bottom, left, and right edges of the screen the same way.

Go ahead and run your game. You'll see that the teddy bear doesn't leave the screen anymore, so we're detecting the collision between the teddy bear's box collider and the edge collider at the edge of the screen, but the teddy bear just slides along the edge of the screen instead of bouncing off.

From the Unity manual entry for Physics Material 2D, "A Physics Material 2D is used to adjust the friction and bounce that occurs between 2D physics objects when they collide." Select the teddy bear game object in the Hierarchy or Scene view. Notice that the Material field for the Box Collider 2D component says "None (Physics Material 2D)". That's where we'll add the material we need, but we need to create one in our project first.

Right click in the Project window and add a new folder named materials. Right click the new folder and select Create > Physics2D Material; name the new material TeddyBearMaterial. With the new material selected, you can see in the Inspector that we can modify the Friction and Bounciness fields for the material. We'll set the Friction to 0 and the Bounciness to 1 (maximum bounciness, like a rubber ball).

Select the teddy bear game object in the Hierarchy or Scene view and drag the TeddyBearMaterial from the Project window onto the Material field for the Box Collider 2D component. When you run the game again, you can see that the teddy bear bounces around inside the screen the way we wanted it to.

Our next step is to save our current teddy bear game object as a prefab so we can easily create more copies of it in our scene. To do this, right click in the Project window and add a new folder named prefabs. Next, drag the teddy bear game object from the Hierarchy window onto the new prefabs folder in the Project window. Finally, rename the prefab TeddyBear and rename the teddy bear game object in the Hierarchy window TeddyBear0.

Change the X and Y Position values for TeddyBear0 to (-3, 2). Now, drag the TeddyBear prefab from the Project window into the Hierarchy window twice, name the new instances TeddyBear1 and TeddyBear2, and change their (X, Y) locations in the game world to (0, 0) and (3, -2) by changing the X and Y Position values in their Transform components.

Select the TeddyBear1 game object in the Hierarchy or Scene view and drag the teddybear1 Sprite from the sprites folder in the Project window onto the Sprite value in the Sprite Renderer component in the Inspector. Now select the TeddyBear2 object in the Hierarchy window and drag the teddybear2 Sprite from the sprites folder in the Project window onto the Sprite value in the Sprite Renderer component in the Inspector.

Run your game and watch until two teddy bears collide; of course, this can take a while, so you can add lots more teddy bears to the scene or modify the impulse force you apply in the `TeddyBear Start` method to make the collision happen more quickly. When two teddy bears collide, you'll see that they start rotating! Although that looks amusing, let's make sure all the teddy bears stay upright as they bounce around in the game.

Select the TeddyBear0 game object in the Hierarchy or Scene view and look at the RigidBody2D component in the Inspector. The component contains a Constraints area at the very bottom; expand that area by clicking the arrow next to it if necessary. As you can see, we can freeze the position of the

rigidbody on the X and Y axes (so the rigidbody always stays at a particular X location, for example) and we can freeze the rotation of the rigidbody around the Z axis. To make it so our teddy bears can't rotate, check the check box to the left of Z in the Freeze Rotation constraint.

Now we've made it so the TeddyBear0 game object can't rotate, but we've only modified this instance, not the prefab. That means that the TeddyBear1 and TeddyBear2 game objects can still rotate. What we want to do next is apply the changes we made to the TeddyBear0 game object back on the prefab so that all instances of the prefab share those changes. To do that, go to the Prefab area near the top of the Inspector and click the Apply button on the right. You can select any of the other game objects or the prefab itself to see that the Rigidbody 2D component for all of them now has rotation around the Z axis frozen.

Run your game again, and it should be working fine. We actually discovered that we could end up with teddy bears that would stick to the edge of the screen if we ran for long enough and they hit the edge just right. We created an additional Physics2D Material called ScreenEdgeMaterial with Friction 0 and Bounciness 1 and attached it to all the edge colliders on the Main Camera. This seemed to resolve the problem for us. Although the ScreenEdgeMaterial has the same Friction and Bounciness as the TeddyBearMaterial, we decided to keep them separate so we could tune each one independently if we discovered we needed to later.

#### *Test the Code*

Although we've been testing the code lots as we implemented our game, we'll still do our last problem-solving step. As a reminder, our Test Case is provided below.

#### **Test Case 1: Checking Bear Behavior**

Step 1. Input: None.

Expected Result: Bears move around screen and bounce off edges and each other

When we run the game, it behaves as expected.

And that's all there is to it! See Figure 6.15. for an image of the final Unity editor with the TeddyBear0 game object selected.

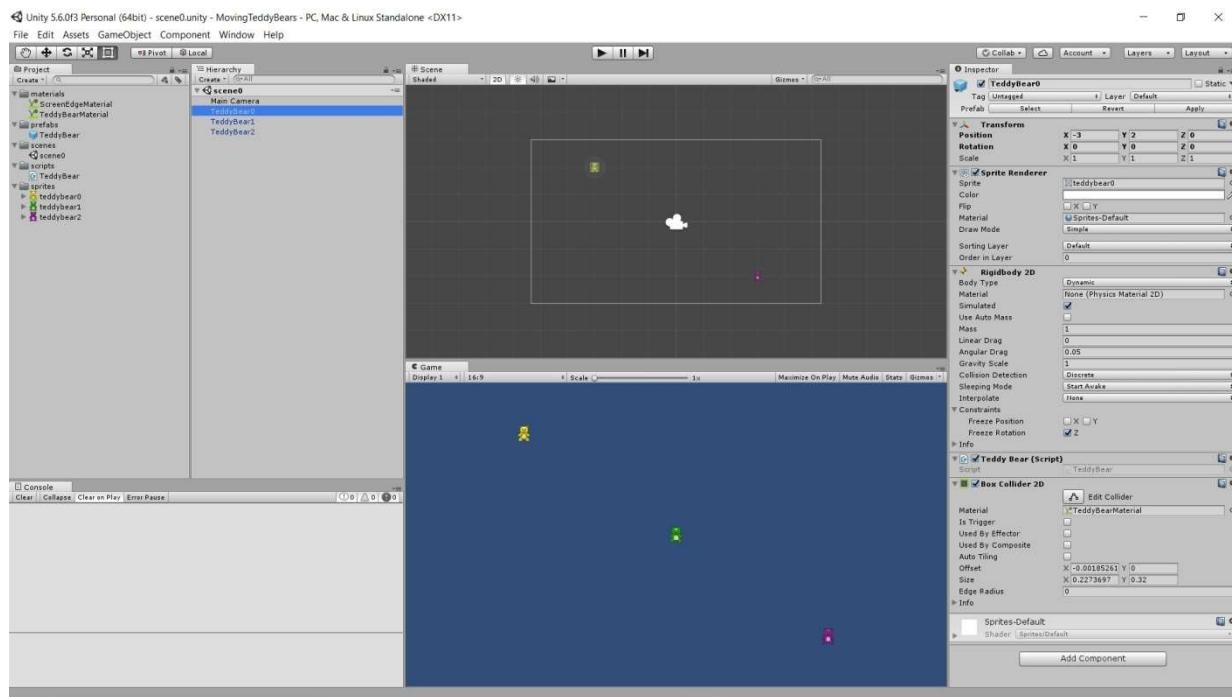


Figure 6.15. Final Moving Teddies Project

# Chapter 7. Selection

Well, we've solved lots of problems up to this point, but all of our programs have used the sequence control structure. Our programs haven't even had to choose between different steps to execute – they simply started at the beginning and continued until they were done.

While these kinds of problems helped us concentrate on understanding the basics of programming without lots of other distractions, it's time to consider some more complicated problems. Specifically, this chapter discusses some of the ways we can solve problems that require that we choose, or select, between different courses of action. Let's get to it.

## 7.1. Selection Control Structure

Suppose we need to make a decision in our problem solution. The sequence control structure won't be sufficient, because it doesn't let us select between different courses of action. Not surprisingly, the selection control structure is designed to let us do exactly that. Let's look at an example:

### *Example 7.1. Printing a Dean's List Message*

Problem Description: Write an algorithm that will print a Dean's List message if the student's Grade Point Average (GPA) is 3.0 or above.

Notice that we've stepped back to algorithms rather than code here. With a new control structure, we want to focus on the key ideas behind the control structure first before learning the C# syntax. Here's one solution:

```
If GPA greater than or equal to 3.0
    Print Dean's List message
```

Basically, our solution will print out the Dean's List message if the GPA is 3.0 or higher; otherwise, it won't do anything. Notice our use of indentation in the algorithm. We indent the second step to show that the second step is only accomplished if the GPA is 3.0 or higher. The associated CFG is shown in Figure 7.1.

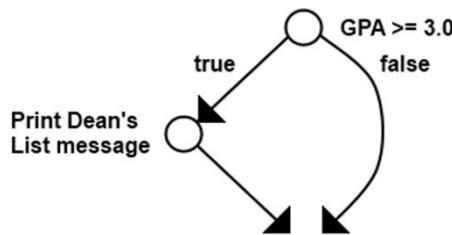


Figure 7.1. CFG for Example 7.1.

When we use a selection control structure, we end up with two or more *branches*. If the GPA is greater than or equal to 3.0, we take the branch on the left (the true next to the branch means that the statement "GPA  $\geq 3.0$ " is true) and print the Dean's List message. If the GPA is less than 3.0, we take the branch on the right. Because this branch doesn't have any nodes, the solution will continue to the next step without printing (or doing) anything on that branch.

Labeling the branches true and false may seem a little awkward to you right now (Yes and No might seem to be more intuitive labels, for example). We chose the above labels because "GPA  $\geq 3.0$ " is called a *Boolean expression*. Remember, a Boolean expression is an expression that evaluates to one of two values – `true` or `false`. For example, the Boolean expression GPA  $\geq 3.0$  evaluates to `true` if the value of GPA is greater than or equal to 3.0; otherwise, it evaluates to `false`.

Notice that both arrows at the bottom of the above CFG "end in thin air." That's because control structures are really building blocks we use to solve a particular problem; if our algorithm had more steps after the selection, we would simply plug in the appropriate CFG in the space after the selection portion of the CFG.

Now let's make our Dean's List example a bit more complicated:

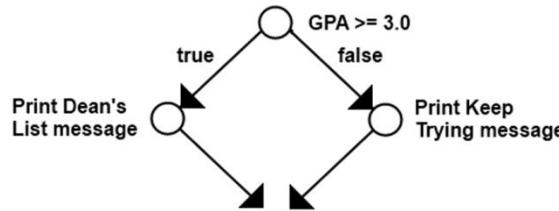
*Example 7.2. Printing a Dean's List or Keep Trying Message*

**Problem Description:** Write an algorithm that will print a Dean's List message if the student's GPA is 3.0 or above and will print a Keep Trying message if it's not.

**Algorithm:** Our solution builds on the one above:

```
If GPA greater than or equal to 3.0
    Print Dean's List message
Otherwise
    Print Keep Trying message
```

Basically, our solution will print out the Dean's List message if the GPA is 3.0 or higher; otherwise, it will print out the Keep Trying message. We again use indentation in the algorithm, this time to show that the second step is only accomplished if the GPA is 3.0 or higher and that the fourth step is only accomplished if the GPA is less than 3.0. The associated CFG is shown in Figure 7.2.



**Figure 7.2. CFG for Example 7.2.**

The branch on the left behaves exactly as it did in our previous solution, but this time the branch on the right (which we follow when GPA is less than 3.0) prints a Keep Trying message.

So now we have control structures that will simply do one step after another (sequence) and that will let us select between different branches (selection). All we need is one more control structure, and we'll have all the building blocks we need to solve ANY problem. Don't worry, that one (iteration) is coming soon.

## 7.2. Testing Selection Control Structures

Before we talk about how the selection control structure is implemented in C#, we need to talk about how we'll test the selection control structure – especially since we need to Write Test Cases before we Write the Code!

Let's take a look at how we'd test the structure we came up with in Example 7.2. Remember, when we use a selection control structure, we end up with two or more branches. Testing becomes more difficult when our CFGs contain branches. To thoroughly test such programs, we should do the following two things:

1. Test every branch at least once
2. Test boundary values in the Boolean expression

In this example, if the GPA is greater than or equal to 3.0, the program prints the Dean's List message; otherwise, the program prints the Keep Trying message. So how do we test every branch at least once? We run the program with a  $\text{GPA} \geq 3.0$  and make sure it prints the Dean's List message then we run the program again with a  $\text{GPA} < 3.0$  to make sure it prints the Keep Trying message.

But what do we mean when we talk about boundary values? We mean values that are right on the boundary of our decision criteria. If all we cared about was testing both branches, we could pick input values of 1.0 and 4.0 (for example) and both branches would be tested. Experience shows, however, that many of the mistakes programmers make that lead to defects (bugs) in software actually occur near the boundaries. In our example, the critical factor in the Boolean expression is whether or not the GPA is  $< 3.0$  or the GPA is  $\geq 3.0$ , so we should try values of 2.9 and 3.0 for the GPA to make sure the program works properly for both of them<sup>19</sup>. Of course, using these two values will end up testing both branches as well, so running the program twice (once with each of the values) should provide sufficient testing.

Here's a set of test cases for our program:

### Test Case 1

**Branches: false branch**

**Boundary Value: 2.9**

Step 1. Input: 2.9 for GPA

Expected Result:

Not on Dean's List this time. Keep Trying.

### Test Case 2

**Branches: true branch**

**Boundary Value: 3.0**

Step 1. Input: 3.0 for GPA

Expected Result:

You made Dean's List !!

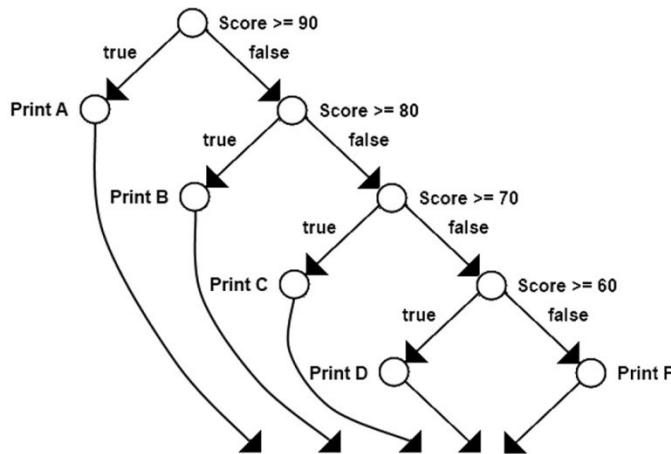
---

<sup>19</sup>Of course, there are plenty of floating point numbers between 2.9 and 3.0, so it can be argued that we're not EXACTLY at the boundary with these values. For the programs in this book, however, testing within 0.1 of floating point boundaries will be sufficient.

Because we need to run the program twice, we developed two separate test cases. Remember, we have to have one test case for each execution (or run) of the program. Of course, these one-step test cases are pretty simple, but as your programs get more complicated, your test cases will get more complicated as well.

This is the first time we've needed multiple test cases to test our program, and we're going to need a more convenient way to refer to them so we don't have to keep saying "set of test cases." One common way to refer to a set of test cases is as a *test plan*, so that's the terminology we'll use from now on. You may also hear people refer to the test plan as a *test suite*.

Now, suppose we have an even more complicated program – one that determines and prints a letter grade, given a test score (assuming there's no curve!). The CFG would probably look something like the one in Figure 7.3.



**Figure 7.3. CFG for Determining and Printing Letter Grade**

We now have lots of branches to test, and to test them all, we need to run the program with a score for an A, a score for a B, a score for a C, a score for a D, and a score for an F. But it's not quite that easy, since we also have to test the boundary conditions for each Boolean expression in the selection statements. We therefore should test the following scores during testing: 90, 89, 80, 79, 70, 69, 60, and 59. We have to run the program 8 times to make sure it's working properly! Clearly, adding a little extra complexity to our program can make testing the program much harder.

### 7.3. If Statements

Now that we understand the selection control structure and how we should test it, it's time to look at how we use that control structure in C#. One construct we can use to choose between different courses of action is the if statement. There are actually a number of different ways we can use this statement; the syntax for the simplest way is provided below:

## SYNTAX: If Statement

```
if (Boolean expression)
{
    executable statements
}
```

*Boolean expression*, some expression that evaluates to `true` or `false`  
*executable statements*, one or more executable statements

---

We need to put a Boolean expression between parentheses after the `if` in the first line. It's important to note that we could also place a `bool` variable or a call to a method that returns a `bool` in the space labeled *Boolean expression*. The real issue is that whatever appears between the parentheses needs to have a value of (evaluate to) either `true` or `false`.

It's also valid to omit the curly braces if you only want to execute a single executable statement if the Boolean expression evaluates to `true`. You'll actually see this approach commonly used by programmers. The problem, though, is that if you later decide to add another statement there, you also need to remember to add the curly braces; you'd be surprised how often programmers forget to do that! Unfortunately, this leads to problems that are very difficult to find. Since we can avoid such problems by simply always including the curly braces, that's the approach we'll use throughout the book (and we suggest you do also).

We use this form of the if statement when we simply need to decide whether or not to perform a particular action. For example, say you need to decide if a student is on the Dean's List; in other words, solve the problem in Example 7.1 using C#. Remember the algorithm from that example:

```
If GPA greater than or equal to 3.0
    Print Dean's List message
```

and turning this into code, we get

```
// check for dean's list
if (gpa >= 3.0)
{
    Console.WriteLine("You made Dean's List !!");
```

We actually snuck something else new into our discussion (`>=`), although we did include it in the list of operators in Chapter 3. There are a number of *relational operators* in C# that let us compare two things (try to figure out the relationship between them). Those operators are

`==`, equal to  
`!=`, not equal to  
`<`, less than  
`<=`, less than or equal to

>, greater than  
>=, greater than or equal to

The C# operators (including relational operators) have an *order of precedence*. From your math courses, you know that in a math equation we evaluate multiplications before additions because multiplication has a higher order of precedence. Here are all the operators in C# that we'll use in this book, listed from highest to lowest precedence (operators on the same line have the same precedence):

```
++, --
+, - (unary operators), !
*, /, %
+, -
<, <=, >, >=
==, !=
&&
||
```

So how does the if statement actually use the Boolean expression we write? When the program gets to the `if`, it evaluates the Boolean expression. If the Boolean expression evaluates to `true`, the *if block* (the statements between the open curly brace and the close curly brace) are executed; otherwise, the program just skips to the line in the program after the close curly brace. Do you see how we use the Boolean expression to decide (or select) which code to execute?

The next form of the if statement we examine takes us one step further by letting us choose between two alternatives. The syntax is as follows:

---

#### SYNTAX: If Statement with Else Block

```
if (Boolean expression)
{
    executable statements
}
else
{
    executable statements
}
```

*Boolean expression*, some expression that evaluates to `true` or `false`  
*executable statements*, one or more executable statements

---

As for the previous form of the if statement, the curly braces are optional if we have only a single executable statement, but we'll always include them in our examples.

Let's extend our example to print a sympathetic message if the student hasn't made Dean's List (Example 7.2). Our algorithm becomes

```
If GPA greater than or equal to 3.0
    Print Dean's List message
Otherwise
    Print Keep Trying message
```

and the resulting code:

```
// check for dean's list or keep trying
if (gpa >= 3.0)
{
    Console.WriteLine("You made Dean's List !!");
}
else
{
    Console.WriteLine("Not on Dean's List this time. Keep trying.");
}
```

The program still evaluates the Boolean expression when it gets to the `if`. If it evaluates to `true`, the program executes the if block, then skips to the line in the program after the final close curly brace. If the Boolean expression evaluates to `false`, the program executes the `else block` (the statements between the open curly brace after the `else` and the final close curly brace). Since a Boolean expression can only evaluate to `true` or `false`, either the if block or the else block is always executed in this form of the if statement.

What if we have multiple (more than two) alternatives we'd like to select from? We use the if statement shown below.

---

#### SYNTAX: If Statement with Else If and Else Blocks

```
if (Boolean expression)
{
    executable statements
}
else if (Boolean expression)
{
    executable statements
}
else if (Boolean expression)
{
    executable statements
}
else
{
    executable statements
}
```

*Boolean expression*, some expression that evaluates to `true` or `false`  
*executable statements*, one or more executable statements

---

Let's expand our algorithm just a bit more to print the sympathetic message if the student's GPA is greater than or equal to 3.0 (and less than 3.0), and inform them that they're on academic probation if their GPA is less than 2.0. Here's the algorithm:

```
If GPA greater than or equal to 3.0
    Print Dean's List message
Otherwise, if GPA is greater than or equal to 2.0
    Print Keep Trying message
Otherwise
    Print Academic Probation Message
```

and the code for our algorithm is

```
// check for dean's list, keep trying, or academic probation
if (gpa >= 3.0)
{
    Console.WriteLine("You made Dean's List !!");
}
else if (gpa >= 2.0)
{
    Console.WriteLine("Not on Dean's List this time. Keep trying.");
}
else
{
    Console.WriteLine("You're on Academic Probation.");
}
```

The program evaluates the first Boolean expression when it gets to the `if`; if it evaluates to `true`, the program executes the first if block, then skips to the line in the program after the final close curly brace. If the first Boolean expression evaluates to `false`, the program goes to the `else if` and evaluates the Boolean expression there; if it evaluates to `true`, the program executes the else if block, then skips to the line in the program after the final close curly brace. Finally, if none of the preceding alternatives have been selected, the program executes the `else` block.

The easiest way to remember how this works is to remember that the program goes "from the top down" – it will execute the statements in the FIRST alternative for which the Boolean expression is `true` (or the `else if` if it gets all the way there), then skip to the program line after the final close curly brace. Our if statements can have as many `else if` portions as you need (each of which has its own Boolean expression, of course), and the `else` portion is optional.

Given the rule that the statements are executed in the first alternative for which the Boolean expression evaluates to `true`, can you see why we didn't need

```
else if ((gpa >= 2.0) && (gpa < 3.0))
```

in the example above? If `gpa` is greater than or equal to 3.0, the first alternative in the `if` statement would have been executed, so the only way we can even get to this `else if` is if the `gpa` is less than 3.0. Including the check for `gpa` less than 3.0 in the `else if` doesn't change the way the code works (because the `gpa < 3.0` part always evaluates to `true` if we get here), but it does make the code more complicated than it needs to be and it also makes it take longer to run. You should keep your Boolean expressions as simple as possible to keep the code clean and fast.

We should point out that the statements contained inside an if statement can also be if statements; we call these *nested* if statements. We'll see an example of nested if statements later on.

So now you know how to add selection to your programs, using the various forms of the if statement. Cool.

## 7.4. Switch Statements

In some situations, a switch statement provides a convenient way to select between different courses of action. Here's the syntax:

---

**SYNTAX:** Switch Statement

```
switch (variableName)
{
    case value:
        executable statements
        break;
    case value:
        executable statements
        break;

    . . .

    default:
        executable statements
        break;
}
```

*variableName*, the name of a variable

*value*, a possible value of the variable

*executable statements*, one or more executable statements

---

We said switch statements are appropriate in some cases because the data type of the variable used for the alternative selection is restricted. Specifically, it can only be a `string`, `char`, integer, or enumeration<sup>20</sup>. The variable's data type is restricted so that we can explicitly list possible values of the variable after each `case`.

This can be a little confusing, so let's look at an example. Say we wanted to print an appropriate message after the user enters one of 4 characters: R, B, J, or C. Based on the character entered, the program should print "Rock and Roll Rules" (for an R), "Blues Rules" (for a B), "Jazz Rules" (for a J), or "Classical Rules" (for a C). If the user didn't enter one of those characters, we should print "You must not like music." The algorithm looks like this:

```
If character is R
    Print Rock and Roll Rules
```

---

<sup>20</sup>In fact, we really don't have to use a variable because C# lets us use an expression to select on if we want. Using a variable to select on is much more common for us, though, so that's what we'll use.

```

Otherwise, if character is B
    Print Blues Rules
Otherwise, if character is J
    Print Jazz Rules
Otherwise, if character is C
    Print Classical Rules
Otherwise
    Print You must not like music

```

and the code (using a switch statement, assuming `musicType` is a `char` variable holding the type of music) is

```

// print message based on music type
switch (musicType)
{
    case 'R':
        Console.WriteLine("Rock and Roll rules");
        break;
    case 'B':
        Console.WriteLine("Blues rules");
        break;
    case 'J':
        Console.WriteLine("Jazz rules");
        break;
    case 'C':
        Console.WriteLine("Classical rules");
        break;
    default:
        Console.WriteLine("You must not like music");
        break;
}

```

In a switch statement, the values we list for each alternative have to be different from the values we list for other alternatives. For example, we couldn't have

```
case 'R':
```

at the beginning of our switch statement, then have

```
case 'R':
```

later on as well. It wouldn't make sense to do this (we should just do everything we need to in the first alternative), so C# won't let you do it.

You should also have noticed that at the end of each alternative we put a

```
break;
```

That tells the computer that we're done with this alternative, and we should leave the switch statement (go to the statement following the closing curly brace for the switch statement). If we forget this `break` after an alternative, the compiler will give us an error saying that we're not allowed to "fall through" from one case label (alternative) to another.

For the last case in our example, we don't list explicit values; we simply say `default`. Using the `default` at the end of the switch statement (the only place you're allowed to put it, by the way), is a shorthand way of saying "all the other possible values not listed above." For the example above, that means that if the `musicType` variable has a value that's not R, J, B, or C, the code will print "You must not like music." To make sure your switch statement covers all possible values of the variable on which you're selecting, you should always include a `default` case label in your switch statement<sup>21</sup>.

One more thing before we move on. For the switch statement above, if the user enters an r rather than an R, it will print "You must not like music" because the user didn't enter a capital R. There's an easy way to fix this, though; we can list multiple cases above an alternative, and the alternative will be executed if any of those cases are true. To change the R alternative to handle both r and R (the other alternatives would be similar), we change it to

```
case 'R':
case 'r':
    Console.WriteLine("Rock and Roll rules");
    break;
```

and we'll now print "Rock and Roll Rules" if `musicType` is either r or R. Of course, we could also make sure that `musicType` is capitalized before we reach the switch statement. That's probably the more common approach, but we wanted to show you how to include multiple cases in a single alternative.

So when the computer gets to the switch statement, it figures out the current value of the variable, executes the alternative with that value listed after the `case`, then skips to the line in the program after the close curly brace. Switch statements therefore give us another way to select the code we want to execute in our programs.

## 7.5. Timers

One of the components we regularly include in our games are timers. Timers are great because we can use them to make something happen (or make something stop happening) when the timer goes off. Timers are great in this chapter because our implementation uses a number of if statements to make sure the timer works properly. Although we usually only follow our five problem solving steps in the Putting It All Together section of each chapter, we'll follow that process in this section also.

Here's our problem description:

Implement and test a Unity `Timer` class that can be used to run for a specified period of time. The consumer of the class should be able to start the timer and determine whether or not the timer is currently running and whether or not it has finished running.

### *Understand the Problem*

The requirements for the `Timer` class are fairly simple, though you might wonder how the consumer of the class tells how long the timer will run. That will be a design decision that we get to make, so we'll defer that decision until we get to the Design a Solution step.

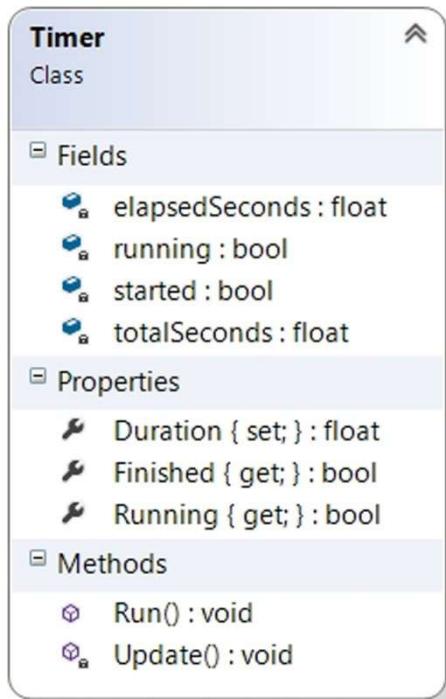
---

<sup>21</sup> As always, there are exceptions to this "rule", especially when we use enumerations. It's a generally good rule of thumb, though.

You can certainly imagine a `Timer` class that has much more robust functionality. For example, we could have a class that lets the consumer of the class pause the timer, reset the timer, and so on. In this section, we'll simply meet the requirements listed in the problem description. If we discover later on that we need more class functionality, we can always expand the `Timer` class then.

### *Design a Solution*

This is an interesting problem because we're just building a single class rather than a full problem solution. We'll of course have a basic Unity project that tests our `Timer` class, but we can focus all our efforts on the `Timer` class itself. We're still not really trying to get you to design your own classes yet, which is why the problem description says to implement and test; we've already done the design. We'll just provide the UML for the `Timer` class in Figure 7.4. and discuss the implementation details later.



**Figure 7.4. Timer Class UML**

You might be wondering about the method that we called `Run`, which a consumer of the class will call to start the timer. Why didn't we call this method `Start` instead? Because the `Start` method in Unity scripts already has a different function, to do the initialization when the game object the script is attached to is added to the scene. It would at least potentially be confusing to have a `Start` method in the `Timer` class that does something different; that's why we called the method `Run` instead.

You also might be wondering why we included the `Duration` property rather than writing a constructor that has a parameter for the timer duration. As you know, we've been using the `Start` method in our Unity scripts to do initialization things; in our `Timer` script, we'll be using the `Update` method instead. We've been getting both those methods "for free" because, by default, our Unity scripts are actually child classes of the Unity `MonoBehaviour` class. We'll spend an entire chapter later in the book discussing how inheritance (including child classes) works, so we won't cover that here. The important point is that we can't use `new` to create instances of the `MonoBehaviour` class or its child classes, so

writing a constructor with a duration parameter won't work for us. We'll cover how we actually do this in the Write the Code step, but the `Duration` property is required for the approach we've taken.

### *Write Test Cases*

To test the `Timer` class, we'll simply create a 3 second `Timer` object, start it, and print a message to the Console window when the timer finishes. We'll then restart the timer again and repeat this process, making sure the timer runs for approximately 3 seconds each time.

Note that our expected results for how long the timer actually executes is approximately 3 seconds rather than exactly 3 seconds. We'll discuss why this may not be exactly 3 seconds when we get to the Write the Code step.

### **Test Case 1**

#### **Checking Timer Functionality**

Step 1. Input: None.

Expected Result:

Timer runs for approximately 3 seconds each time

### *Write the Code*

We'll start by creating a Unity project, creating a scenes folder and saving `scene0`, creating a scripts folder and creating a new `Timer` script (remember, a script is a class in Unity), and opening the new script in Visual Studio.

Here's the code for the `Timer` class:

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

/// <summary>
/// A timer
/// </summary>
public class Timer : MonoBehaviour
{
    #region Fields

    // timer duration
    float totalSeconds = 0;
```

Initializing the `totalSeconds` field, which holds the timer duration, to 0 gives us an easy way to confirm that the timer has been given a valid duration when the consumer of the class tries to start the timer.

```
// timer execution
float elapsedSeconds = 0;
bool running = false;
```

We use the `elapsedSeconds` field to keep track of how long the timer has been running since it was started and we use the `running` field to tell whether or not the timer is currently running.

```
// support for Finished property
bool started = false;
```

We use the `started` field to keep track of whether or not the timer has ever been started. This field might seem unnecessary to you, but we actually need it so we can tell when the timer has finished running. We can't just check to see if the `running` field is `false` for this, because the `running` field starts as `false`. We only want to say the timer has finished running after it's been started, ran for its duration, then stopped running.

```
#endregion

#region Properties

/// <summary>
/// Sets the duration of the timer
/// The duration can only be set if the timer isn't currently running
/// </summary>
/// <value>duration</value>
public float Duration
{
    set
    {
        if (!running)
        {
            totalSeconds = value;
        }
    }
}
```

This property lets the consumer of the class set the duration of the timer; recall our discussion in the Design a Solution step about why we need to do this. We'll look at the details of how properties work later in the book, but we will tell you that the code above sets the `totalSeconds` field to the `value` the consumer of the class provides when they access this property.

Note that we only let the consumer of the class set the timer duration if the timer isn't already running (which should make sense to you if you think about it). This is a case where having a property to control access to the object's state really helps us, because we can do some error checking to make sure changing the state is appropriate before we do so.

```
/// <summary>
/// Gets whether or not the timer has finished running
/// This property returns false if the timer has never been started
/// </summary>
/// <value>true if finished; otherwise, false.</value>
public bool Finished
{
    get { return started && !running; }
}
```

This property returns `true` if the timer is finished and `false` otherwise. The value that's returned is simply the result of evaluating

```
started && !running
```

so let's make sure you understand that Boolean expression. Remember that `and (&&)` only evaluates to `true` if both the left and right operands are `true`. The `started` field will only be `true` if, at some point in the past, we started the timer (remember our discussion about this above); that field starts as `false`, and we'll see where it gets set to `true` soon. The `!running` operand will only be `true` if the `running` field is currently `false`.

You should now be able to see that the Boolean expression above will only be `true`, and therefore the `Finished` property will be `true`, if the timer was started but is no longer running.

```
/// <summary>
/// Gets whether or not the timer is currently running
/// </summary>
/// <value>true if running; otherwise, false.</value>
public bool Running
{
    get { return running; }
}
```

This is the easiest property of all! It simply returns the value of the `running` field, which will be `true` if the timer is currently running and `false` otherwise.

```
#endregion

#region Methods

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // update timer and check for finished
    if (running)
    {
        elapsedSeconds += Time.deltaTime;
        if (elapsedSeconds >= totalSeconds)
        {
            running = false;
        }
    }
}
```

The outer if statement checks to see if the timer is currently running, because if it isn't there's no need to keep track of its elapsed time. That means that we only go into the body of this if statement if `running` is `true`.

In the next line of code, we access `Time.deltaTime`, which tells us how many seconds the previous frame in the game took to complete. Because the `Update` method is called every frame in Unity, that value is exactly how much more time has elapsed on the timer since the last time we updated the elapsed time.

This line of code also introduces some new syntax. The code

```
elapsedSeconds += Time.deltaTime;
```

is a convenient shorthand for

```
elapsedSeconds = elapsedSeconds + Time.deltaTime;
```

In fact, we can use `-=`, `*=`, and `/=` in the same way.

Notice that this is our first example of nested if statements, where we have one if statement inside another. The inner if statement compares the elapsed seconds for the timer to the timer duration. If the timer has been running for exactly, or more than, the timer's duration, we stop the timer by setting the `running` field to `false`.

Why didn't we use `==` here to check for an exact match rather than `>=?` Imagine that on the previous call to the `Update` method we set `elapsedSeconds` to 2.999; that means that the timer isn't quite done yet. Also imagine that we're running at 120 frames per second, so on this call to `Update` the value of `Time.deltaTime` is 0.0083, so the new value for `elapsedSeconds` is 3.0073. This is certainly not exactly 3, but it just as certainly indicates that the timer should stop running.

This is also why the expected results for our test case says that the timer runs for approximately 3 seconds rather than exactly 3 seconds.

```
/// <summary>
/// Runs the timer
/// Because a timer of 0 duration doesn't really make sense,
/// the timer only runs if the total seconds is larger than 0
/// This also makes sure the consumer of the class has actually
/// set the duration to something higher than 0
/// </summary>
public void Run()
{
    // only run with valid duration
    if (totalSeconds > 0)
    {
        started = true;
        running = true;
    }
}

#endregion
```

The `Run` method only starts the timer if the timer has a valid duration (greater than 0); awesome, another if statement! If the timer does have a valid duration, we set the `started` field to `true` so we can implement the `Finished` property properly and we set the `running` field to `true` so the elapsed time for the timer gets updated properly in the `Update` method.

We're finally ready to test our new timer.

### Test the Code

We actually need to add an additional script to test the `Timer` class. Here's the code for the `TimerTest` class:

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

/// <summary>
/// A script to test the timer class
/// </summary>
public class TimerTest : MonoBehaviour
{
    // test object
    Timer timer;

    // time measurement
    float startTime;
```

We want to print out how long the timer ran for our test, so we use the `startTime` field to keep track of the time at which we started the timer.

```
/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // create and run timer
    timer = gameObject.AddComponent<Timer>();
```

Recall that in the Design a Solution step we said that we can't use `new` to create instances of the `MonoBehaviour` class or its child classes, and the `Timer` class is a child class of the `MonoBehaviour` class, so we can't call a `Timer` constructor to instantiate a `Timer` object.

Instead, we add a new `Timer` object as a component of the game object the `TimerTest` script is attached to (in this case, the Main Camera game object). Up to this point we've added components to game objects in the Unity editor, but we can use the `AddComponent` method to add components at run time as well. As you can see, this method is a generic method like the `GetComponent` method; just like when we used that method, we provide the data type (class name, `Timer` in the case above) for the component between the `<` and the `>`.

```
timer.Duration = 3;
timer.Run();
```

The above code sets the timer duration to 3 seconds by accessing the `Duration` property then starts the timer by calling the `Run` method.

```
// save start time
startTime = Time.time;
}
```

In the above line of code, we access `Time.time`, which tells us how many seconds have elapsed since the game started running. We save this in the `startTime` field so we can calculate how long the timer ran once it's finished.

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // check for timer just finished
    if (timer.Finished)
    {
        float elapsedTime = Time.time - startTime;
        print("Timer ran for " + elapsedTime + " seconds");

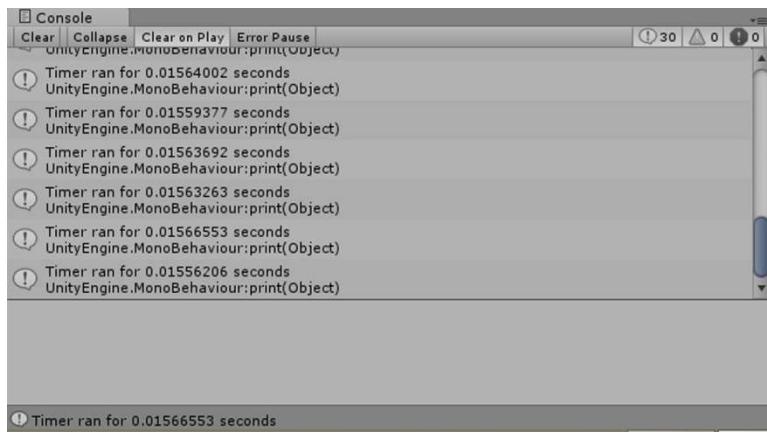
        // save start time and restart timer
        startTime = Time.time;
        timer.Run();
    }
}
```

The Boolean expression for the if statement above checks if the timer has finished because we only want to print the message once the timer is done.

The elapsed time calculation gets the new value of how many seconds have elapsed since the game started running and subtracts the `startTime` field to get how many seconds the timer actually ran. The remaining lines of code print the message to the Console window, save the new start time (since we're about to restart the timer) and start the timer again.

That might have seemed like a lot of work to go to just to test our `Timer` class, but it's not uncommon to write a non-trivial chunk of code to support our testing, especially when we're doing unit testing (as we are here) rather than functional testing.

When we attach the `TimerTest` class to the Main Camera object in the scene and run the game, we get the output shown in Figure 7.5.



**Figure 7.5. Test Case 1: Checking Timer Functionality**

Whoa, what's happening here? The first run of the timer took 3.020025 seconds (we weren't fast enough to catch that in our screen shot) as expected, but each of the next runs of the timer is much less than 3 seconds! In fact, for each of those runs the timer is finished after a single frame in the game. As an aside, you can see that there's some variation in how long each frame takes to run by observing the differences in the output seconds.

Okay, just as in normal programming it looks like we didn't get the code quite right, so we need to go back and do some debugging.

### *Write the Code Again*

So how do we figure out what's wrong? First, we need to think about what's going on, then we need to use the debugger to gather more information, and finally, we need to (hopefully!) fix the problem.

Here's what we know at this point:

1. The timer works fine the first time we run it
2. The timer only runs for one frame every time after that (even if we didn't realize it was only one frame, we certainly know it's running for much less time than it should)

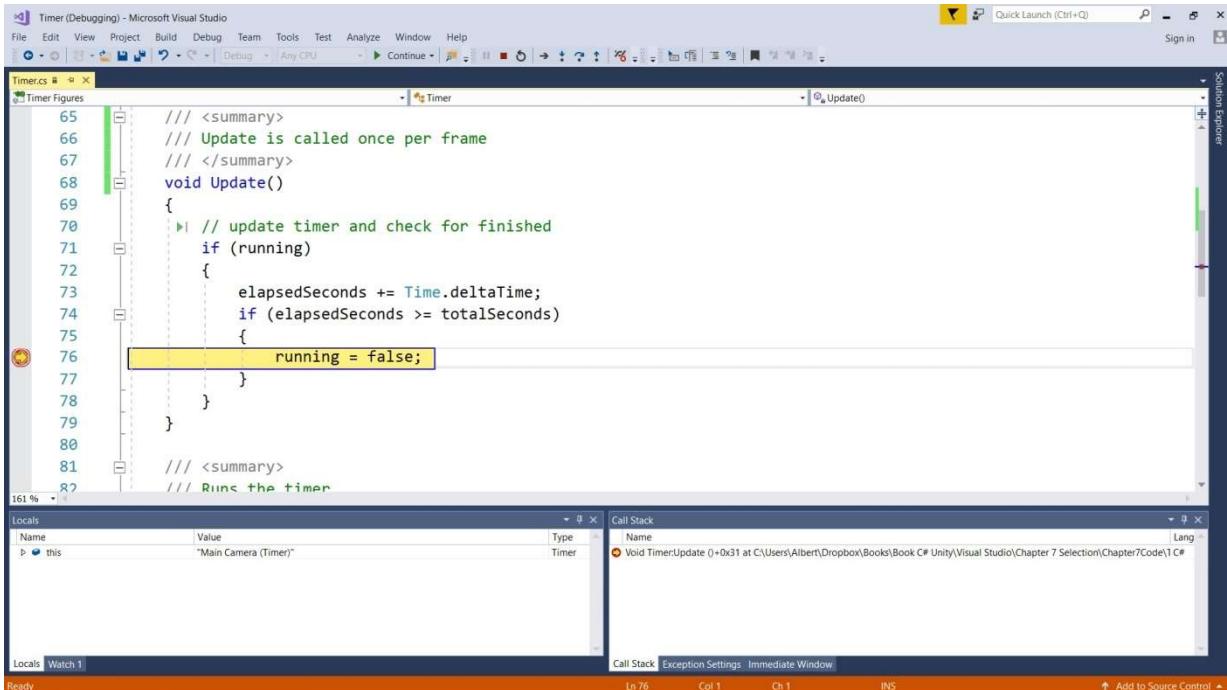
So to try to figure out the problem, we should pause the code after the timer finishes the first time (because we know it worked correctly up to that point), then look carefully at what happens when we run the timer the second time.

To do this, we'll use the debugger in Visual Studio. Go to Visual Studio and select Debug > Attach Unity Debugger from the menu bar at the top. In the resulting popup, double-click the current Unity project. You have to have the Unity editor running to do this, and it's easiest to pick the correct project if you only have one copy of Unity running. This step attaches Visual Studio to the Unity editor so we can debug in Visual Studio as we run the game in the Unity editor. You can also attach Visual Studio to the Unity editor by clicking Attach to Unity on the second row of options at the top.

Next, we'll set something called a *breakpoint* in Visual Studio. To do this, left-click the mouse in the light gray column just to the left of the line of code in the `Timer Update` method that sets the running

field to `false`. You can tell that there's a breakpoint there now by the red circle in the column. When we debug our program, it will run until it reaches the breakpoint, then stop.

Go back to the Unity editor and run the game. It will take a little while, because running all the debugging stuff slows things down, but eventually you'll see that the program stopped at the breakpoint in Visual Studio because Windows gives the Visual Studio window focus at that point (if it doesn't have focus already). You'll see the line of code at the breakpoint highlighted in yellow and the red circle next to it will have a yellow arrow in it. See Figure 7.6 to see what Visual Studio looks like for us at this point.



**Figure 7.6. Test Case 1: Checking Timer Functionality**

Our approach as we debug will be that we look at the values of the fields in the `Timer` class as it runs to find our problem. If you put the mouse over a particular field in the code and wait a moment, you'll get a small popup with the field's name and its current value. Do that now for the `running` field and you'll see that it's currently `true` (we haven't executed the line of code at the breakpoint yet). Press F10, which "steps over" (executes) the current line of code and check the `running` field again; it's now `false` because the line of code that sets that field to `false` just executed.

At this point the timer ran to completion once, and we know that worked correctly. Next, we'll add another breakpoint where we set the `started` field to `true` in the `Run` method. After adding that breakpoint, press F5 or select Run > Continue from the menu bar at the top to make the code start running again; it should stop almost immediately at the new breakpoint.

Let's make it a little more convenient to check the values of our fields at this point. Click the Locals tab in the window on the lower left if the window in that location isn't currently the Locals window. Click the small arrow next to "this" in that window to show all the fields, and their current values, for the `Timer` class. As you can see, everything still looks fine at this point. Press F10 twice to watch the value `running` field change from `false` to `true`.

Okay, press F5 again to run the code until it hits one of our two breakpoints; it should stop next at our original breakpoint. Click the arrow next to "this" if necessary and look at all the current values; our Locals window is shown in Figure 7.7.

Name	Value	Type
this	"Main Camera (Timer)"	Timer
base	"Main Camera (Timer)"	UnityEng
Finished	false	System.B
Running	true	System.B
elapsedSeconds	3.020657	System.S
running	true	System.B
started	true	System.B
totalSeconds	3	System.S

Figure 7.7. Locals Window Contents

Aha! Look at the value for the `elapsedSeconds` field. It says the timer has been running for 3.020657 seconds, but we just started the timer for the second time. We initialized the `elapsedSeconds` field to 0 when we declared the field, so everything worked fine the first time, but we didn't reset that field back to 0 when we restarted the timer. That means the timer "thought" it had been running for a long time when `Update` was called on the next frame, so it immediately finished. In the `Run` method, add a line of code that sets the `elapsedSeconds` field to 0 at the end of the body of the if statement.

### Test the Code Again

When we run our game again, we get the output shown in Figure 7.8.

Console	
<input type="button" value="Clear"/>	<input type="button" value="Collapse"/>
<input type="button" value="Clear on Play"/>	<input type="button" value="Error Pause"/>
(1) Timer ran for 3.004451 seconds	UnityEngine.MonoBehaviour:print(Object)
(1) Timer ran for 3.000025 seconds	UnityEngine.MonoBehaviour:print(Object)
(1) Timer ran for 3.000004 seconds	UnityEngine.MonoBehaviour:print(Object)
(1) Timer ran for 3.000042 seconds	UnityEngine.MonoBehaviour:print(Object)
(1) Timer ran for 3.015715 seconds	UnityEngine.MonoBehaviour:print(Object)
(1) Timer ran for 3.015715 seconds	UnityEngine.MonoBehaviour:print(Object)

Figure 7.8. Test Case 1: Checking Timer Functionality

As you can see in the Console window, the timer didn't run for exactly 3 seconds each time for the reason discussed above, but the test case passes because the timer ran for approximately 3 seconds each time.

## 7.6. Putting It All Together

Let's build on our final teddy bear game from the previous chapter by adding two new features. Specifically, we'll give each teddy bear a lifespan so they actually "die" after a given period of time and we'll spawn new teddy bears in the game at random intervals.

Here's our problem description:

Move teddy bears around the screen until the player quits the game. Each teddy bear should only live for 10 seconds, then should be removed from the game. New teddy bears of random colors should be spawned into the game at random screen locations at random intervals between 1 and 2 seconds.

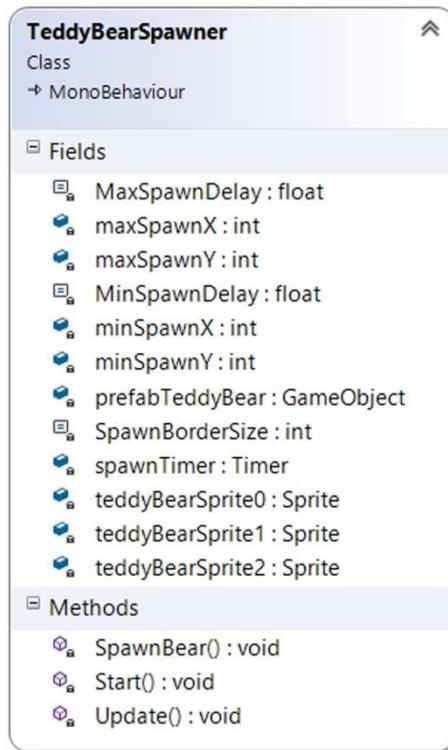
### *Understand the Problem*

There's really nothing confusing about the problem, though random seems to appear an awful lot! The problem description doesn't specify how many bears to start with, so we'll just start with none and let the spawner populate the game with teddy bears. Let's move on to our design.

### *Design a Solution*

You should probably have already realized that the `Timer` class from the previous section will come in handy both for deciding when to kill teddy bears (fiend!) and for deciding when to spawn teddy bears.

We'll add the dying functionality to our `TeddyBear` script from the previous chapter but we'll write a new `TeddyBearSpawner` script, which we'll attach to the Main Camera, to handle the spawning. The UML for the `TeddyBearSpawner` script is shown in the figure below; we'll discuss the details when we Write the Code.



**Figure 7.9. `TeddyBearSpawner` Class UML**

## *Write Test Cases*

To test the game, we'll simply run it for a while to see if it meets the required functionality. With so much randomness, we're not going to be able to specify precise expected results, so instead we list the things to look for while the game runs.

### **Test Case 1**

#### **Checking Game Behavior**

Step 1. Input: None.

Expected Result: Game runs with the following characteristics:

Each teddy bear dies (is removed from the game) after approximately 10 seconds

New teddy bears appear at random intervals of approximately 1 to 2 seconds

New teddy bears are randomly selected from the 3 teddy bear colors

New teddy bears are spawned at random screen locations

## *Write the Code*

Let's start by adding the teddy bear death functionality to the `TeddyBear` script. Because our `Timer` class will help us do this, we declare a new `Timer` field called `deathTimer` (ouch!). We also add a constant at the top of the script for the teddy bear lifespan; that way, we can easily change that value if we need to tune our game later. Here's what that code looks like (download the code from [www.burningteddy.com](http://www.burningteddy.com) and open it up to see the complete `TeddyBear` code):

```
// death support
const float TeddyBearLifespanSeconds = 10;
Timer deathTimer;
```

We also need to add code to the `Start` method to add the timer component, set its duration and start it:

```
// create and start timer
deathTimer = gameObject.AddComponent<Timer>();
deathTimer.Duration = TeddyBearLifespanSeconds;
deathTimer.Run();
```

Finally, remember how we deleted the `Update` method from the `TeddyBear` script because we didn't need it? Well, we need it now so we can kill the teddy bear when the death timer finishes (man, you gotta love this stuff). Here's the complete `Update` method:

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // kill teddy bear if death timer finished
    if (deathTimer.Finished)
    {
        Destroy(gameObject);
    }
}
```

The only thing new here is our call to the Unity `Destroy` method. This method removes whatever object we pass in as an argument from the game, so by passing it the game object the `TeddyBear` script is attached to (we access the game object a script is attached to using `gameObject`) we destroy that teddy bear game object.

Those are the only changes we needed to make to the `TeddyBear` script. We actually ran the previous version of the game with these changes included to make sure all 3 teddy bears died as appropriate (they did).

Okay, let's walk through the new `TeddyBearSpawner` script:

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

/// <summary>
/// A teddy bear spawner
/// </summary>
public class TeddyBearSpawner : MonoBehaviour
{
    // needed for spawning
    [SerializeField]
    GameObject prefabTeddyBear;
```

When it's time to spawn a new teddy bear game object, we'll write code that creates a new instance of a teddy bear prefab. We mark this field with `[SerializeField]` so we can just drag the prefab from the prefabs folder in the Project window onto the field in the Inspector in the Unity editor.

```
// saved for efficiency
[SerializeField]
Sprite teddyBearSprite0;
[SerializeField]
Sprite teddyBearSprite1;
[SerializeField]
Sprite teddyBearSprite2;
```

We mark these fields with `[SerializeField]` so we can just drag the sprites from the sprites folder in the Project window onto the fields in the Inspector in the Unity editor. Although we could look up the sprites at run time whenever we needed to spawn a teddy bear, it's more time-efficient to store them in our fields instead, then just grab the one we need when we spawn a teddy bear.

This is actually a very common tradeoff in programming. We've decided to save CPU cycles at run time, but it costs us extra memory (the memory to store the 3 fields) to do so. We're definitely making the right choice in this particular case, but if we were planning to deploy our "game" to a device that had very little memory, we might decide to make our game take a little longer to run instead.

```
// spawn control
const float MinSpawnDelay = 1;
const float MaxSpawnDelay = 2;
Timer spawnTimer;
```

We're going to spawn a new teddy bear regularly, but in a random range between 1 and 2 seconds each time. By declaring constants for the minimum and maximum spawn delays, we're making the code that actually selects the new spawn delay more readable. We use the `spawnTimer` field to determine when it's time to spawn a new teddy bear. Because we can set a new timer duration by accessing the `spawnTimer` `Duration` property and restart the timer, we only need a single `Timer` object; we don't need to create a new timer every time we spawn a teddy bear.

```
// spawn location support
const int SpawnBorderSize = 100;
int minSpawnX;
int maxSpawnX;
int minSpawnY;
int maxSpawnY;
```

Although the problem description doesn't address it, we're going to make sure we spawn each new teddy bear totally within the screen. In part, that's because it would be ugly to spawn a new teddy bear that's partially outside the screen, but even more importantly, we don't want to spawn a new teddy bear with its collider intersecting with one of the edge colliders for the screen edges. If we do that, the physics result can be pretty interesting!

That implies that when we do spawn a new teddy bear (we'll discuss that later in the code) we should make sure we don't spawn on top of one of the teddy bears that's already in the scene. We'll do that later in the book, but we need to know about while loops first before we can do that in a reasonable way, so we'll defer that until the chapter that covers while loops.

```
/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // save spawn boundaries for efficiency
    minSpawnX = SpawnBorderSize;
    maxSpawnX = Screen.width - SpawnBorderSize;
    minSpawnY = SpawnBorderSize;
    maxSpawnY = Screen.height - SpawnBorderSize;
```

As the comment above says, we save the spawn boundaries for efficiency. If we don't do it that way, we'd have to do two subtractions (for the max spawn x and y values) every time we spawned a new teddy bear. It certainly doesn't take modern CPUs that long to do, but every little bit counts, especially in game development!

We used the `Screen` class, which gives us access to display information, to find out the current width and height of the screen window. That's an important thing to do here so our code will keep working properly independent of the resolution the game is running at.

```
// create and start timer
spawnTimer = gameObject.AddComponent<Timer>();
spawnTimer.Duration = Random.Range(MinSpawnDelay, MaxSpawnDelay);
spawnTimer.Run();
}
```

The first and third lines of code above work just like they did in our `TimerTest` script in the previous section. For the timer duration, we need a random number between 1 and 2 seconds. We're using the same `Random` `Range` method we used in the previous chapter to calculate the impulse force to get our teddy bears moving to get that random number.

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // check for time to spawn a new teddy bear
    if (spawnTimer.Finished)
    {
        SpawnBear();
```

Every frame (`Update` is called every frame), we check to see if the spawn timer is finished. If it is, it's time to spawn a new bear. We do that by calling a separate method we wrote, so we'll discuss that method shortly.

Why did we decide to write a separate method instead of just including the code to spawn the new bear here instead? Because the code to spawn the new teddy bear is somewhat long, and our code tends to be more readable if we keep each method somewhat small. There are no hard and fast rules about what the optimal size of a method is, so we'll just demonstrate good practices for this throughout the book. This is something you'll get a sense for the more you develop code and the more you look at the code written by other people, and it might even be specified in the coding standards at your company.

```
// change spawn timer duration and restart
spawnTimer.Duration =
    Random.Range(MinSpawnDelay, MaxSpawnDelay);
spawnTimer.Run();
}
```

Now that we've spawned a new teddy bear, it's time to pick a new random delay between 1 and 2 seconds and start the spawn timer again; that's what the two lines of code above do.

```
/// <summary>
/// Spawns a new teddy bear at a random location
/// </summary>
void SpawnBear()
{
    // generate random location and create new teddy bear
    Vector3 location = new Vector3(Random.Range(minSpawnX, maxSpawnX),
        Random.Range(minSpawnY, maxSpawnY),
        -Camera.main.transform.position.z);
    Vector3 worldLocation = Camera.main.ScreenToWorldPoint(location);
```

The code above generates the world location for the new teddy bear in two steps: first we generate the location in screen coordinates, then we convert those screen coordinates to world coordinates.

The first line of code generates the random location in screen coordinates as a new `Vector3` object. The first argument to the `Vector3` constructor generates a random x location in the appropriate range and the

second argument does the same for the random y location. The third argument requires a little more discussion.

We need to use the Main Camera (which we can access using `Camera.main`) to get our new teddy bear placed properly because the location and other characteristics of the camera determine where something in the world is shown on the screen. If this seems strange to you, think of zooming in or out with a digital camera. The "thing" you're aiming at with the camera is at a specific location in the world, but changing the zoom settings on the camera changes where that "thing" appears on the camera preview screen. The same thing happens if you walk toward or away from the "thing" with the camera; by changing the position of the camera in the world you change the image on the camera preview screen.

So how do we pick the appropriate z value for the location in screen coordinates for our new teddy bear? Intuitively, 0 seems like the correct choice because we're trying to work in 2D, but 0 isn't the correct z value for the location in screen coordinates; what we really need is for z to end up as 0 in world coordinates so all our 2D objects are at  $z = 0$  in the game world. Based on the discussion in the previous paragraph, we need to use the current location of the Main Camera as we calculate our new z location in screen coordinates. It turns out that simply negating the z position of the Main Camera (in our current game, the Main Camera is at  $z = -10$ , so the third argument evaluates to 10 above) will result in the  $z = 0$  we need in the game world after we execute the second line of code above.

The second line of code calls the `Camera.ScreenToWorldPoint` method to convert the screen coordinates of our new teddy bear into world coordinates so we can actually place the new teddy bear into the correct location in the scene. The argument we pass to the method is the location in screen coordinates. The `ScreenToWorldPoint` method returns a `Vector3` object (we learned that by reading the documentation), so we store the result in a `Vector3` variable called `worldLocation`.

```
GameObject teddyBear = Instantiate(prefabTeddyBear) as GameObject;
teddyBear.transform.position = worldLocation;
```

We saw in the previous chapter that prefabs can be very useful when we're working in the Unity editor, but they're also great when we need to create new instances of game objects at run time. The first line of code above uses the Unity `Instantiate` method to create a new instance of the prefab we've saved in the `prefabTeddyBear` field. That method returns an `Object`, though (yes, we read the documentation again), and we actually need a `GameObject` instead. The end of the line of code – the `as GameObject` part – converts the `Object` to a `GameObject`. If that feels like a type cast to you, that's because it is a type cast, just for reference types rather than value types.

The second line of code simply sets the position of our new teddy bear game object to the location in world coordinates that we worked so hard to generate.

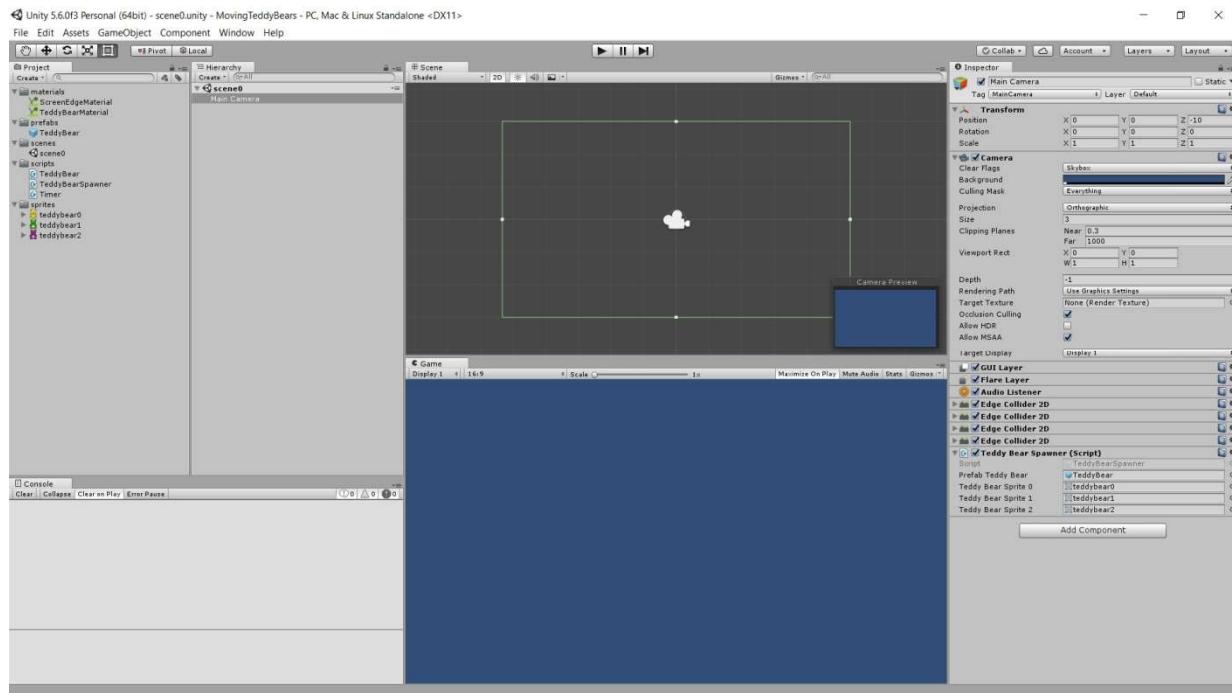
```
// set random sprite for new teddy bear
SpriteRenderer spriteRenderer =
    teddyBear.GetComponent<SpriteRenderer>();
int spriteNumber = Random.Range(0, 3);
if (spriteNumber == 0)
{
    spriteRenderer.sprite = teddyBearSprite0;
}
else if (spriteNumber == 1)
{
```

```
        spriteRenderer.sprite = teddyBearSprite1;
    }
} else
{
    spriteRenderer.sprite = teddyBearSprite2;
}
}
```

Wow, another if statement! The chunk of code simply picks a random number (0, 1, or 2) and sets the sprite to one of the three teddy bear sprites based on that number. A 0 will pick the yellow sprite, a 1 will pick the green sprite, and a 2 will pick the magenta sprite.

We're actually using a different overload of the `Random` `Range` method that works with integers instead of floats. We still provide min and max values for the arguments when we call the method, but for this overload min is inclusive and max is exclusive. In other words, min is possible as one of the values returned by the method but max will never be returned by the method. That's why we know our method call above can only return 0, 1, or 2.

We'll be good to go with just a little more setup. Add the `TeddyBearSpawner` script to the Main Camera in the scene and populate the fields appropriately. Your final project (with Main Camera selected) should look like Figure 7.10.

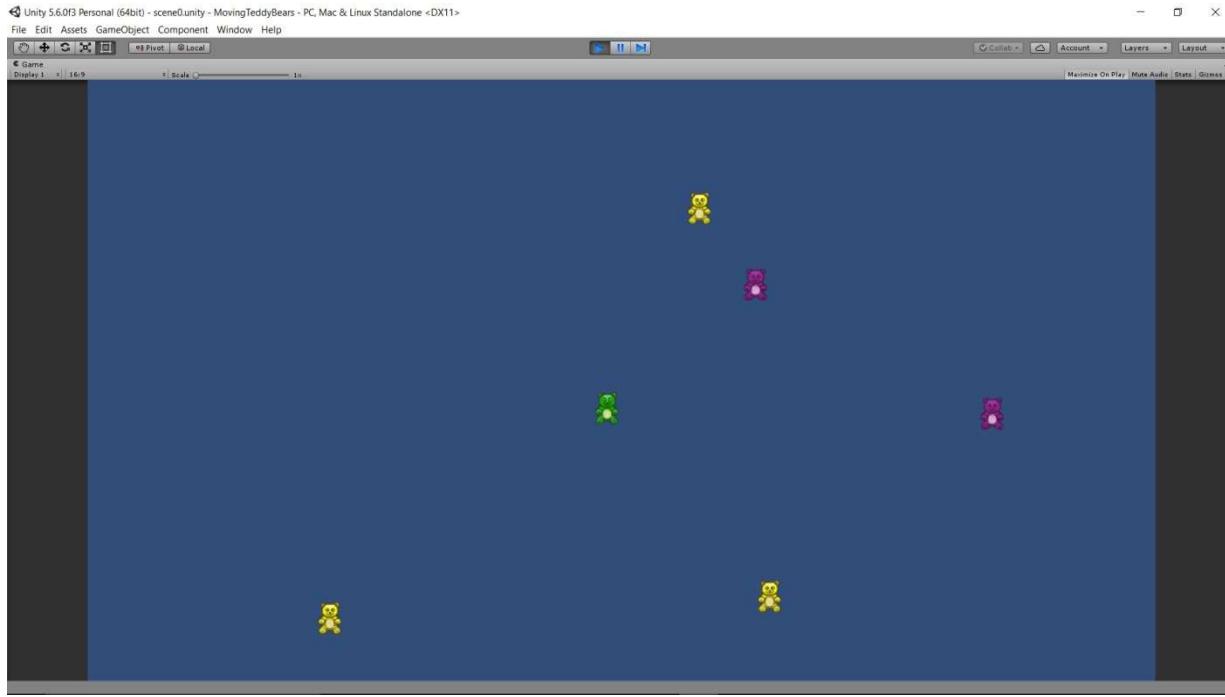


**Figure 7.10. Final Project**

### *Test the Code*

Finally, we run our test case. If we run the game and wait for a while, you'll end up with something like Figure 7.11. You do need to check all the criteria in the test case (teddy bears disappear after about 10

seconds, new teddy bears spawn in random locations, and so on), but our solution is working fine, so we're done solving this problem.



**Figure 7.11. Test Case 1: Checking Game Behavior**

## 7.7. Common Mistakes

### *Using = Instead of == in a Boolean Expression*

Remember, using `=` assigns a value to a variable, while using `==` compares two values for equality. If you try to use `=` in a Boolean expression, you'll get a compilation error.

### *Missing Breaks in a Switch Statement*

When you're done processing a particular alternative in the switch statement you need to include a `break` to leave the switch statement (go to the statement following the close curly brace for the switch statement). If we forget this `break` after an alternative, the compiler will give us an error saying that we're not allowed to "fall through" from one case label (alternative) to another.

### *Not Including All Possible Values in Switch Statement*

C# doesn't require that you include ALL possible values of the switch variable in the alternatives, but it's a good idea to include them all anyway. If there are some variable values for which you don't need to do anything, simply include

```
default:  
    break;
```

as the last alternative in your switch statement.

*Trying to use 0 for z in screen coordinates*

Although using 0 for z in screen coordinates feels intuitive, what we really need is  $z == 0$  in world coordinates. Negating the Main Camera's z position in screen coordinates will convert to 0 for z in world coordinates.

## Chapter 8. Unity Mice, Keyboards, and Gamepads

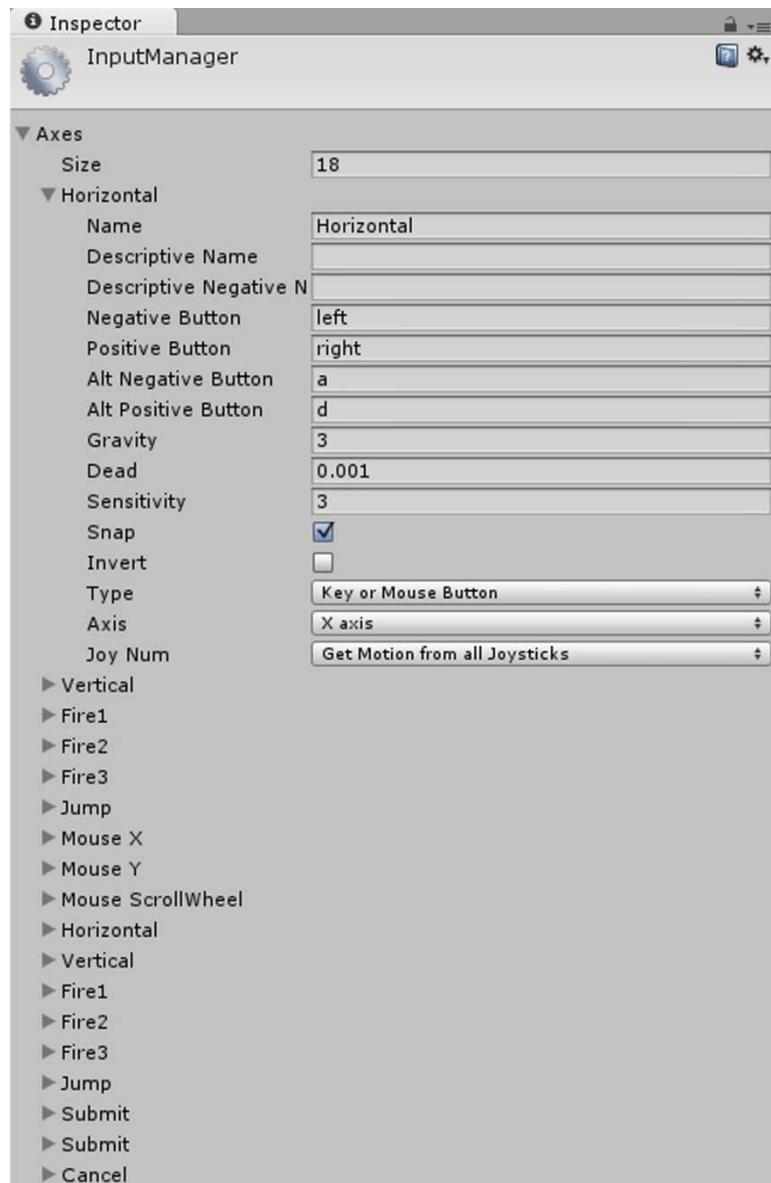
We now know how to make our programs do a variety of interesting things, but we're missing the thing that makes a game a game – interactivity. Sure, we've had the user enter text using the keyboard in console applications and even done things based on that text, but we haven't actually built any Unity games that take and act on user input.

In this chapter, we explore how we can use the Unity input system to get and use mouse, keyboard, and gamepad inputs to let the player interact with our game world. Unity also supports mobile device input, but we won't cover that in this book.

Unity actually has a number of built-in character controllers you can use to have the player control their character in the game. We're using the Unity input system directly in this chapter to give you a solid understanding of how the input system works and how you can program character movement based on player input, but you should of course feel free to explore the built-in character controllers on your own as well if you'd like.

### 8.1. The Input Manager

Before we discuss the details of using mouse, keyboard, and gamepad input, let's look briefly at the Unity Input Manager. Create a new Unity 2D project or open one of the projects you've already created. Select Edit > Project Settings > Input from the menu bar. The input manager will appear in the Inspector as shown in Figure 8.1 (we clicked the arrow next to Axes and the arrow next to the top Horizontal label to expand those regions).



**Figure 8.1. Input Manager**

As you can see, the Input Manager gives us the ability to configure a variety of input axes; we can also add new axes as we need to. We'll hit the high points for the Horizontal axis shown here, then dig into the details as necessary in later sections.

The Name property lets us get the axis by name from a script. Negative Button moves along the axis in the negative direction and Positive Button moves along the axis in the positive direction. Alt Negative Button and Alt Positive Button are additional buttons for those directions. As you can see from the figure, the default for the Horizontal axis is the left/right and a/d keyboard keys, which is as you'd expect in a typical 2D game.

The Type of input for an axis is Key or Mouse Button (as shown above), Mouse Movement (which is the input type for the Mouse X and Mouse Y axes), or Joystick Axis (which we use for gamepad input). The Axis value doesn't really matter if our input type is Key or Mouse Button, but we use X axis for horizontal input and Y axis for vertical input from Mouse Movement and Joystick Axis input types. Joy

Num only matters for the Joystick Axis input type, and it lets us filter which joystick(s) can control the given axis.

The bottom line is that we have a very flexible, configurable way to process player input; we can even let the player configure their input at run time. Let's (finally!) start adding interactivity to our games!

## 8.2. Mouse Input

We'll start by looking at how we can get and use mouse input. As a first step, we'll see how we can make a teddy bear follow the mouse around the screen. We'll then add the ability to blow up the teddy bear (of course!) by pressing the left mouse button.

Create a new Unity 2D project, save the scene into a scenes folder, import the teddy bear sprite and the explosion sprite sheet into a sprites folder, and create empty prefabs and scripts folders.

Now we're ready to start working on our Teddy Bear game object and the associated `TeddyBear` script. Drag the teddybear sprite from the sprites folder in the Project window onto the Hierarchy window and change the name of the new game object to `TeddyBear`. Right click the scripts folder in the Project window, create a new `TeddyBear` script, and open the new script in Visual Studio. To make the `TeddyBear` follow the mouse around the screen, we'll make the `TeddyBear`'s location match the current mouse position on each frame. That means the code we need to add should be in the `Update` method.

It turns out that the `Input` class exposes a `mousePosition` field that gives us the current position of the mouse (in screen coordinates) as a `Vector3`. We can then just convert those screen coordinates to world coordinates and set the position of the teddy bear to match those world coordinates to make the teddy bear follow the mouse. Here's the complete method:

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // calculate mouse world position
    Vector3 mousePosition = Input.mousePosition;
    mousePosition.z = -Camera.main.transform.position.z;
    Vector3 worldPosition = Camera.main.ScreenToWorldPoint(mousePosition);

    // move teddy bear to mouse location
    transform.position = worldPosition;
}
```

The first line of code gets a copy of the mouse position so that the second line can change the z coordinate just like we did when we were spawning teddy bears in Section 7.6. Recall that we needed to do that so that all our 2D objects are at `z == 0` in the game world. The third line of code converts the mouse position from screen coordinates to world coordinates. The last line of code moves the teddy bear to the mouse position in the world.

Wow, that was really easy! When we run the game, we see that the `TeddyBear` follows the mouse around just as we wanted. Awesome. Drag the `TeddyBear` from the Hierarchy window to the prefabs folder in the Project window to create a `TeddyBear` prefab.

Let's actually make the teddy bear stay on the screen even when the mouse goes outside the screen. This is called *clamping* the teddy bear in the screen, and it's a very common technique in game development.

One way to accomplish this is to create edge colliders around the sides of the screen, add a collider to the TeddyBear game object, and let the Unity physics engine handle this like we did in Section 6.5. Let's actually do this a different way, because we might want to create a game where physical objects (like projectiles) can actually leave the screen rather than bouncing off the edges of the screen. This will give you more flexibility in the games you develop and will also expand your toolbox of programming solutions to this problem so you can select the most appropriate approach based on the game you're building.

We'll still want a collider attached to our TeddyBear, so add a Box Collider 2D to the game object and edit the collider boundaries so the collider fits tightly to the shape of the TeddyBear. Be sure to Apply your changes to the prefab.

The approach we'll follow will be to check the left, right, top, and bottom edges of the collider on each frame to make sure they're not outside the screen. If they are, we'll clamp them to the appropriate edge of the screen. For example, if the left edge of the collider is outside the left edge of the screen, we'll change the position of the TeddyBear so the left edge of the collider lines up with the left edge of the screen.

The first thing we'll do is add some fields to our `TeddyBear` class to store the world coordinates of the four sides of the screen. This makes it so we don't have to convert those edges (which are in screen coordinates) to world coordinates to do the boundary check on every frame. Instead, we'll do the conversion once in the `Start` method.

The other thing we'll do in the `Start` method – again, for efficiency – is save half the width and half the height of the TeddyBear collider. Doing it this way will make it easier and faster to do the comparison between the edges of the TeddyBear collider and the edges of the screen on every frame.

```
/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // save screen edges in world coordinates
    float screenZ = -Camera.main.transform.position.z;
    Vector3 lowerLeftCornerScreen = new Vector3(0, 0, screenZ);
    Vector3 upperRightCornerScreen = new Vector3(
        Screen.width, Screen.height, screenZ);
    Vector3 lowerLeftCornerWorld =
        Camera.main.ScreenToWorldPoint(lowerLeftCornerScreen);
    Vector3 upperRightCornerWorld =
        Camera.main.ScreenToWorldPoint(upperRightCornerScreen);
    screenLeft = lowerLeftCornerWorld.x;
    screenRight = upperRightCornerWorld.x;
    screenTop = upperRightCornerWorld.y;
    screenBottom = lowerLeftCornerWorld.y;

    // save collider dimension values
    BoxCollider2D collider = GetComponent<BoxCollider2D>();
    Vector3 diff = collider.bounds.max - collider.bounds.min;
```

```

    colliderHalfWidth = diff.x / 2;
    colliderHalfHeight = diff.y / 2;
}

```

The block of code that saves the screen edges creates `Vector3` locations for the lower left and upper right corners of the screen, then uses the Main Camera to convert those points from screen coordinates to world coordinates. It then saves the locations of all 4 screen edges in world coordinates. In our example (we didn't change the default camera settings), we get the following values: `screenLeft` is -9.735, `screenRight` is 9.735, `screenTop` is 5, and `screenBottom` is -5.

The second block of code first gets the collider for the TeddyBear game object. The second line of code in that block takes advantage of the fact that the `BoxCollider2D` class exposes a `bounds` field, which is a `Bounds` object. The `Bounds` class exposes `max` and `min` fields, which give us the upper right and lower left corners of the collider. By subtracting `min` from `max`, we get the width (the `x` component of the resulting `Vector3`) and height (the `y` component of the resulting `Vector3`) of the collider, so we can just divide by 2 to set the `TeddyBear` `colliderHalfWidth` and `colliderHalfHeight` fields.

Now that we've saved all the values we need to efficiently do the clamping on each frame, it's time to actually do it! We wrote a new `ClampInScreen` method that we call at the end of the `Update` method:

```

/// <summary>
/// Clamps the teddy bear in the screen
/// </summary>
void ClampInScreen()
{
    // check boundaries and shift as necessary
    Vector3 position = transform.position;
    if (position.x - colliderHalfWidth < screenLeft)
    {
        position.x = screenLeft + colliderHalfWidth;
    }
    if (position.x + colliderHalfWidth > screenRight)
    {
        position.x = screenRight - colliderHalfWidth;
    }
    if (position.y + colliderHalfHeight > screenTop)
    {
        position.y = screenTop - colliderHalfHeight;
    }
    if (position.y - colliderHalfHeight < screenBottom)
    {
        position.y = screenBottom + colliderHalfHeight;
    }
    transform.position = position;
}

```

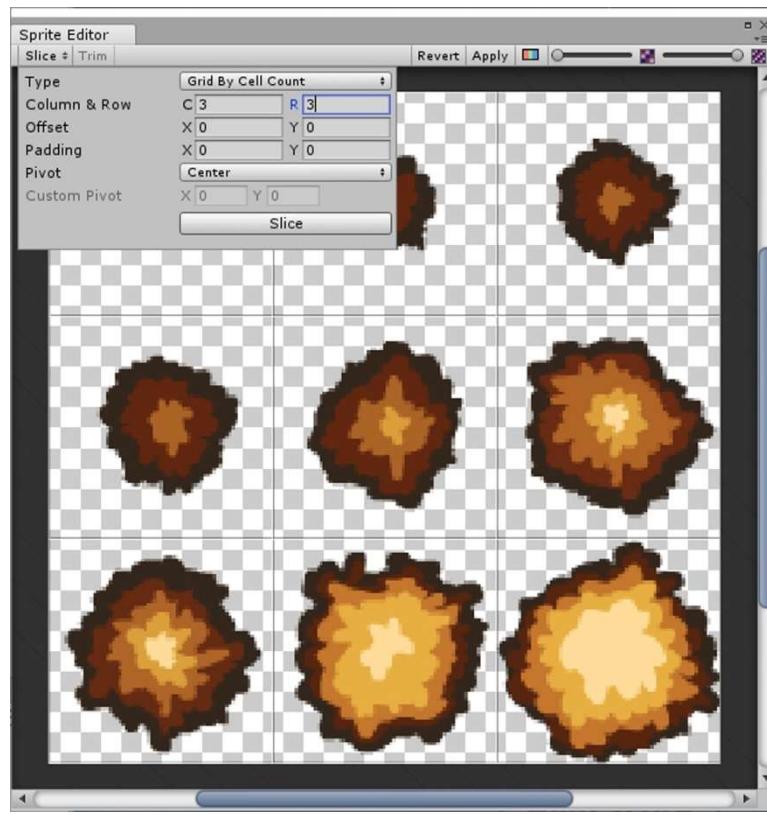
As you can see, we first retrieve the position of the TeddyBear into a local variable so we can modify it as necessary to do the clamping. We then use a sequence of if statements to check each of the four edges of the screen, shifting the TeddyBear back into the screen as necessary. The final line of code sets the `transform.position` field of the `TeddyBear` game object to actually move the game object to position.

Okay, we've made the TeddyBear follow the mouse around and also made sure that the TeddyBear stays in the screen. It's time to blow the TeddyBear up with the left mouse button!

Before we can do that, we have to build an Explosion prefab that we can create when we blow up the TeddyBear. Getting the visual portion of the prefab set up properly is more complicated than we've seen in the past because the explosion sprite is animated rather than static.

Select the explosion sprite in the sprites folder of the Project window and use the dropdown in the Inspector to change the Sprite Mode to Multiple; this tells Unity that you're using a *sprite sheet* rather than a single image. We still need to get the individual frames of the animation set properly, so click the Sprite Editor button in the Inspector.

When the Sprite Editor popup appears, select the Slice dropdown at the upper left and change the Type to Grid by Cell Count. Change the C and R values to 3 (because there are 3 columns and 3 rows in our sprite sheet) and click the Slice button; you'll end up with the image shown in Figure 8.2. Click the Apply button near the top of the Sprite Editor, then close the Sprite Editor using the X at the upper right of the popup. If you click the arrow to the left of the explosion sprite in the sprites folder of the Project window, you'll see that the sprite now has 9 separate frames. You can left-click each of them and see that frame at the bottom of the Inspector.



**Figure 8.2. Explosion in Sprite Editor**

Now that we've got our separate animation frames set up, it's time to work on our Explosion prefab. Drag the explosion sprite (not one of the individual frames, the sprite with the arrow to the left of it) from the sprites folder of the Project window onto the Hierarchy window and change the name of the new game object to Explosion.

The next thing we need to do is add an Animator component to the Explosion game object. Click the Add Component button and select Miscellaneous > Animator (NOT Animation) to do that now. The Explosion still won't play as an animation, because we need to set up a Controller for the Animator component.

Create new empty animations and controllers folders in the Project window. Right-click the new controllers folder and select Create > Animator Controller. Name the new controller explosionController. Select the Explosion game object in the Hierarchy window and drag the explosionController from the controllers folder in the Project window onto the Controller field of the Animator component in the Inspector.

Next, we need to implement the explosionController. Double-click the explosionController in the controllers folder in the Project window to open the Animator window for the animator controller. Make sure you have the Animator window selected, then select Window > Animation on the menu bar to open a new Animation window. Select the Explosion game object in the Hierarchy window. In the Animation window, click the Create button to create a new Animation Clip. In the file navigation popup, select the animations folder, navigate into that folder, change the File name to Exploding, and click Save. Expand the explosion sprite in the sprites folder in the Project window, select all 9 children, and drag them into the Animation window where you see all the vertical lines. Change the Samples text box (near the upper left of the Animation window) to 30 so the explosion animation plays at 30 frames per second. Close the Animation window and the Animator window (right-click the tab for the window and select Close Tab).

Click the Play button in the Unity editor to test the animation; you should just see a looping explosion animation wherever you placed your Explosion game object in the scene. This might have seemed like a lot of work to get this simple animation working, but that's because Unity has a very robust animation system. If you want to learn more about the Unity animation system – which you'll definitely want to do if you start building character or other more complex animations – you'll probably find that the 2D Character Controllers tutorial from Unity is really helpful (it includes a discussion about building 2D animations from sprites).

Although it's kind of fun to have explosions loop forever once they're added to our game, that's not very realistic, so we'll fix that now. Create a new `Explosion` script in the scripts folder in the Project window and drag the script onto the Explosion game object in the Hierarchy window. Here's the complete script:

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

/// <summary>
/// An explosion
/// </summary>
public class Explosion : MonoBehaviour
{
    // cached for efficiency
    Animator anim;

    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
```

```

{
    anim = GetComponent<Animator>();
}

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // destroy the game object if the explosion has finished its animation
    if (anim.GetCurrentAnimatorStateInfo(0).normalizedTime >= 1)
    {
        Destroy(gameObject);
    }
}
}

```

Because we need to check if the animation is finished on every frame, we retrieve a reference to the Animator component and save it in the `anim` field in the `Start` method so we don't have to get that component every frame.

In the `Update` method, we check to see if the animation has finished by retrieving the `normalizedTime` field of the current animator state info. The `normalizedTime` field is a `float`, where the integer part is the number of times the animation has looped and the fractional part represents the percent progress in the current loop. That means that when the `normalizedTime` hits 1, it has just completed the first loop through the animation frames. When that happens, we destroy the game object the script is attached to.

Drag the Explosion game object from the Hierarchy window into the prefabs folder in the Project window to create the prefab. Delete the Explosion game object from the Hierarchy window, because we'll be creating the Explosion at run time on a left click.

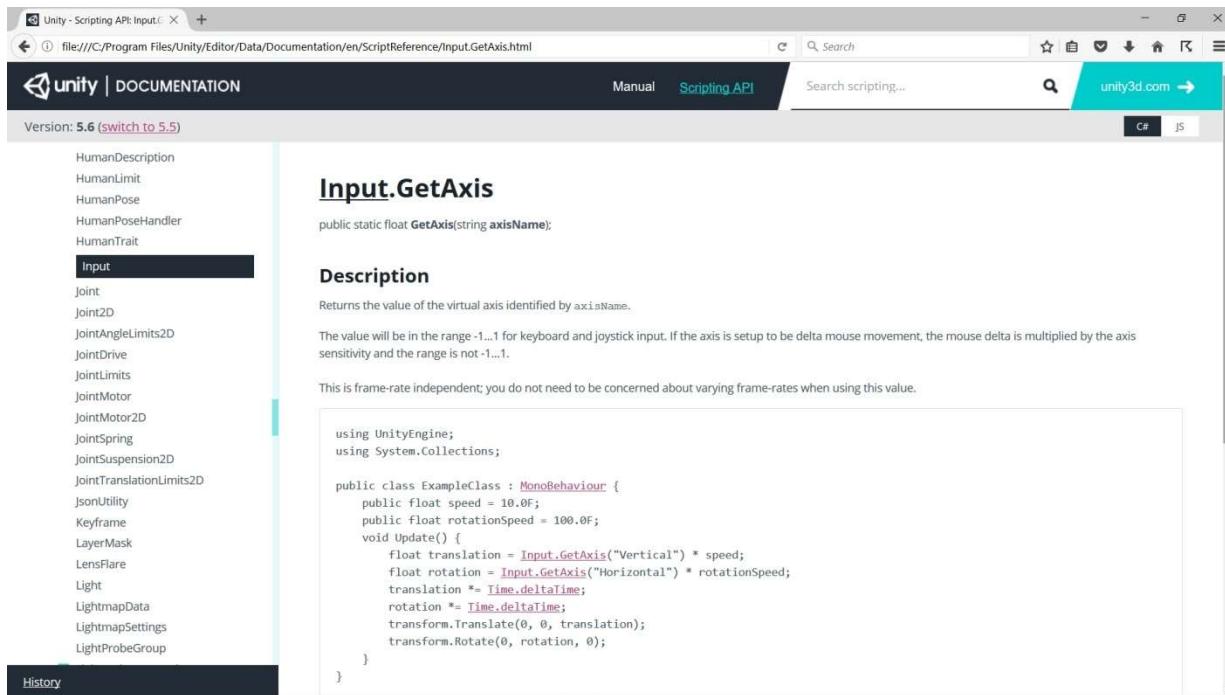
Now we need to make it so the `TeddyBear` script can spawn the Explosion prefab on a left click. We could have written this functionality into a different script instead, but the `TeddyBear` is really blowing itself up on a left click, so it makes sense that the `TeddyBear` creates the Explosion in the scene.

First, we add a `GameObject` field called `prefabExplosion` to the `TeddyBear` class and mark it with `[SerializeField]`. Select the `TeddyBear` game object in the Hierarchy window, then drag the Explosion prefab from the prefabs folder in the Project window onto the `Prefab` `Explosion` field in the `Teddy Bear` script component in the Inspector to populate the field.

Although we could actually detect a left click without using an input axis from the Input Manager, it will actually be more convenient in this case to add a `BlowUpTeddy` axis in the Input Manager. In the Unity editor, select `Edit > Project Settings > Input`. To add a user-defined axis, add one to the number in the `Size` text box near the top of the Inspector and press Enter.

Expand the bottom axis (its name defaults to `Cancel`) and change the Name to `BlowUpTeddy`. Set the Positive Button to space (we'll use that when we process keyboard input later in the chapter) and set the Alt Positive Button to mouse 0 (mouse button 0 is the left mouse button).

To access an input axis from a script, we use the `Input.GetAxis` method; see Figure 8.3 for an excerpt of the documentation for that method.



**Figure 8.3. GetAxis Documentation**

Because we've only set positive buttons for the `BlowUpTeddy` axis, and because keys and mouse buttons are only pressed or not, the value we get back from a call to the `GetAxis` method for that axis will only be either 0 or 1.

To make the `TeddyBear` destroy itself on a left click, we add the following code to the end of the `Update` method:

```
// on left click, create explosion at teddy bear and destroy teddy bear
if (Input.GetAxis("BlowUpTeddy") > 0)
{
    Instantiate(prefabExplosion, transform.position, Quaternion.identity);
    Destroy(gameObject);
}
```

The Boolean expression checks to see if one of the buttons for the `BlowUpTeddy` axis is pressed (yes, at this point we can also blow up the teddy bear with the space bar). We're calling the `Instantiate` method with three arguments this time: the prefab to instantiate, the location where it should be instantiated, and the rotation to be applied to the new object (`Quaternion.identity` means don't rotate it at all). The last line of code destroys the `TeddyBear` game object, so the only object in the scene after that line executes is the `Explosion` game object we just instantiated from the prefab.

Run the game to confirm that the teddy bear still follows the mouse around and that it explodes properly when you click the left mouse button. If it does, select the `TeddyBear` game object in the Hierarchy window and click the Prefab Apply button near the top of the Inspector so that the `TeddyBear` prefab has the Prefab Explosion field set properly.

And that's it for mouse input. We'll actually see another way to handle mouse input in Chapter 9, but you definitely have everything you need at this point to let the player use mouse input to interact with your game world.

### 8.3. Keyboard Input

Now that we know how mouse input works in Unity, we'll find that lots of the same concepts apply to getting keyboard input. Let's change the `TeddyBear` `Update` method so we can move the teddy bear around with the keyboard instead of having it follow the mouse position. Don't worry, we'll blow up the teddy bear soon! Moving the teddy bear using the keyboard actually turns out to be more complicated than following the mouse around was – let's see why.

We'll be using the Horizontal and Vertical axes from the Input Manager to move the `TeddyBear` around, but as we know from the `GetAxis` documentation, we'll be getting a value between -1 and 1. If you look at the default buttons for those Axes in the Inspector, you'll see that they're the expected keyboard keys for moving left/right and up/down. Because they're keyboard keys, which can only be pressed or not, the value we get back from a call to the `GetAxis` method for those axes will only be -1, 0, or 1.

So how do we use the value to actually move the `TeddyBear`? A good way to think about this is that the value is used to apply thrust to move the teddy bear in a particular direction. First, we'll add a `moveUnitsPerSecond` field to our `TeddyBear` class and mark it with `[SerializeField]`. We mark the field with `[SerializeField]` so we can easily change its value in the Inspector to tune the teddy bear's movement speed. Select the `TeddyBear` game object in the Hierarchy window of the Unity editor and set the Move Units Per Second field in the `Teddy Bear` script in the Inspector to 5. Of course, you can change this value to make the teddy bear move faster or slower as you see fit.

We also need to make significant changes to the `TeddyBear` `Update` method; here's the new method:

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    Vector3 position = transform.position;

    // get new horizontal position
    float horizontalInput = Input.GetAxis("Horizontal");
    if (horizontalInput != 0)
    {
        position.x += horizontalInput * moveUnitsPerSecond *
                      Time.deltaTime;
    }

    // get new vertical position
    float verticalInput = Input.GetAxis("Vertical");
    if (verticalInput != 0)
    {
        position.y += verticalInput * moveUnitsPerSecond *
                      Time.deltaTime;
    }
}
```

```
// move and clamp in screen
transform.position = position;
ClampInScreen();

// on space bar, create explosion at teddy bear and destroy teddy bear
if (Input.GetAxis("BlowUpTeddy") > 0)
{
    Instantiate(prefabExplosion, transform.position,
        Quaternion.identity);
    Destroy(gameObject);
}
}
```

Let's look at moving horizontally in detail; moving vertically works in a similar way. First we get the Horizontal axis from the Input Manager. The Boolean expression checks to see whether or not there's an input on this axis; if there is, the line of code in the if body changes the x location of the teddy bear.

How does it do this? By taking the value of the axis (which will be either -1 or 1 in the if body), which tells whether we're moving to the left or right, then multiplying by `moveUnitsPerSecond` and `Time.deltaTime`. From the `deltaTime` documentation, we know that this is the time in seconds it took to complete the last frame. That means that multiplying `moveUnitsPerSecond` by `Time.deltaTime` gives us the magnitude of the units to move in this frame.

As we said, moving vertically works similarly. The rest of the method, which changes the teddy bear's location, clamps the teddy bear in the screen, and responds to the `BlowUpTeddy` axis, is virtually unchanged from what we had when processing mouse input.

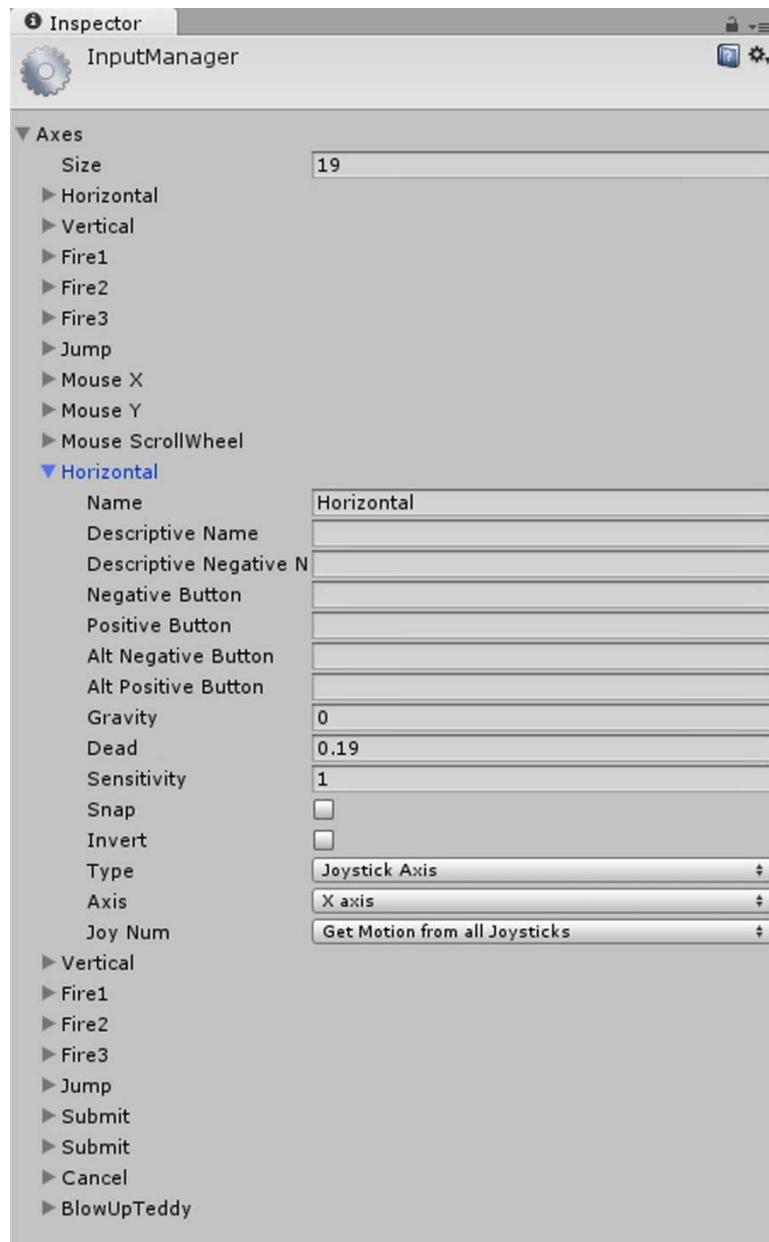
If you run the game now, you'll see that you can drive the teddy bear around using the arrow or wasd keys and you can blow the teddy bear up using the space bar. Excellent.

Before we move on to gamepad input, you should know that there's actually a subtle bug in the code above. If you think carefully about how the code works, you'll realize that if the player holds down both a vertical key and a horizontal key to move diagonally, they'll actually move `moveUnitsPerSecond * Time.deltaTime * square root of 2` units along the diagonal instead of `moveUnitsPerSecond * Time.deltaTime` units. That means the player can move around faster through the game world if they always move diagonally. There's actually a famous bug in Doom called the Doom Strafe 40 bug that does essentially the same thing (for essentially the same reason). You should think about how you'd fix that bug on your own.

## 8.4. Gamepad Input

Processing gamepad input is very similar to processing keyboard input; we'll use a 360 controller as our gamepad, but Unity supports a wide variety of controllers. For our example, we won't have to change any code at all, we'll just need to add one more user-defined input axis. We'll explore gamepad input using the same example we used above, but this time the teddy bear will move around based on left thumbstick movement and we'll blow the teddy bear up by pressing the A button on the controller.

By default, our Unity game contains 2 Horizontal axes: one with Type set to Key or Mouse Button (as shown in Figure 8.1.) and one with Type set to Joystick Axis (as shown in Figure 8.4. below).



**Figure 8.4. Joystick Horizontal Axis**

In fact, we get 2 of many of the default axes, which makes it very easy to port the input processing from mouse/keyboard input to gamepad input. In our current example, we don't even need to change any code to do that.

One of the benefits of using a gamepad over a keyboard is that the value of our call to `GetAxis` for the Horizontal and Vertical axes is in the range from -1 to 1. In other words, if the thumbstick is only partially deflected, the magnitude of the value we get is greater than 0 and less than 1. This is very useful for games in general, because the amount of deflection in each axis gives us very fine-grained player control over the objects in the game world.

We still need to add another axis to blow up the teddy bear using a joystick button, though. Because each input axis can only have two positive buttons (Positive Button and Alt Positive Button), and

because our `BlowUpTeddy` axis already uses both of them for the space bar and the left mouse button, we don't have any more room in that axis.

The good news is that we can create another axis with the same name to detect the joystick button press. When we call the `GetAxis` method in our script, it gets the input from all the axes with the name we provided; that's why we can have two Horizontal axes in the default game.

Add another input axis called `BlowUpTeddy`, set the Positive Button to joystick button 0 and make sure the Alt Positive Button is blank. Make sure the Type is set to Key or Mouse Button, which is used for joystick buttons as well. We know we're not trying to use a Joystick Axis to blow up the teddy bear, we're trying to use a joystick button.

By the way, on a 360 controller joystick button 0 is A, joystick button 1 is B, and so on. You can easily figure out the button to number mapping by changing the number of the button you're responding to and seeing which button on the controller blows up the teddy bear! We know someone who had an inordinate amount of fun doing that ...

That's all we need to do to get and process gamepad input in our games.

## 8.5. Putting It All Together

Once we give the player the ability to provide input to our game, we open up a whole new world of possibilities! Here's a problem description for an actual (though still pretty simple) game:

The player uses the keyboard to drive a fish around the game world, eating teddy bears with the head of the fish. The teddy bears bounce off the sides of the screen as they move around until they're eaten by the fish, at which point they're removed from the game. If a collision between the fish and a teddy bear isn't at the head of the fish, the teddy bear bounces off the fish. Teddy bears spawn at random intervals between 1 and 2 seconds.

### *Understand the Problem*

There are lots of details to work out as we build the game, but the rules of the game seem pretty straightforward, so we can move on to our design.

### *Design a Solution*

We only need two game objects in the game: `TeddyBear` and `Fish`. The `TeddyBear` game object behaves just as it did in the "game" in Section 7.6, so we'll just reuse the `TeddyBear` script and all the game object components we used there here as well. We'll also reuse the `TeddyBearSpawner` and `Timer` scripts from our solution to that problem.

The `Fish` game object is a different story; we're definitely going to need a new `Fish` script to implement the required functionality. There are two key pieces of functionality we need in the `Fish` game object: moving and facing the right way based on player input and handling collisions with `TeddyBear` game objects properly.

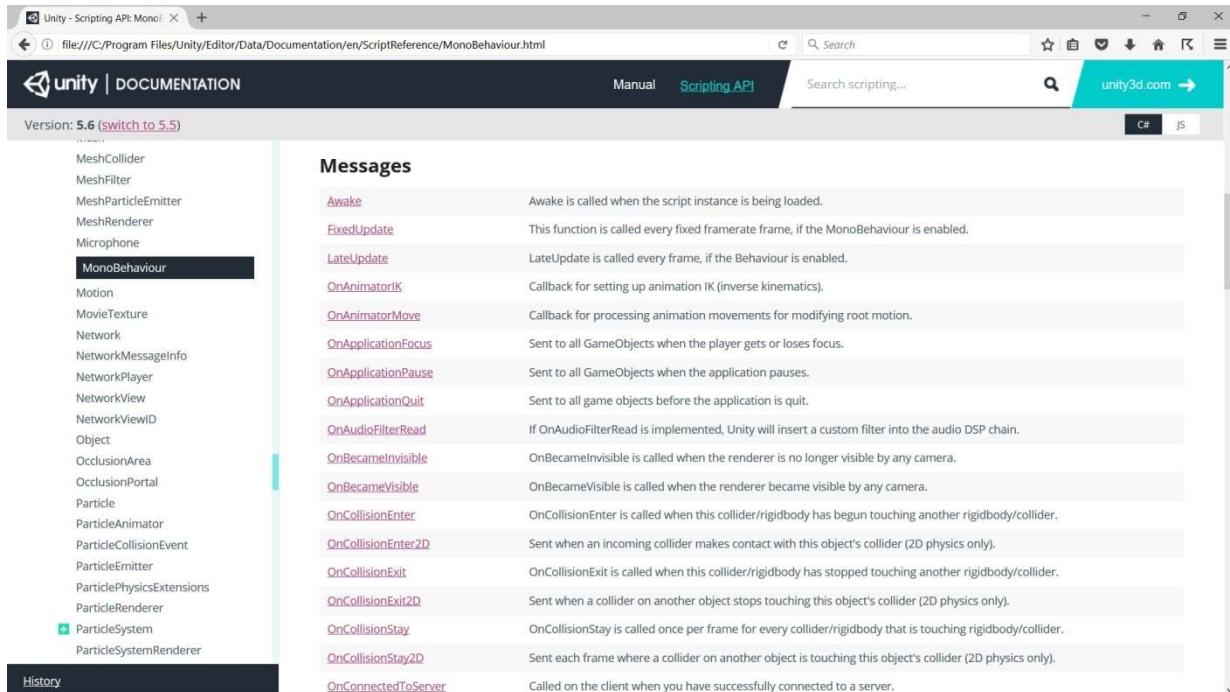
Let's look at the fields we need first. Because the `Fish` game object movement is like the `TeddyBear` game object responding to keyboard input, we'll start with the fields for the movement amount, fish

collider dimensions, and the screen edges. Because we need different sprites for facing left and facing right, we'll include fields for each of those sprites. Because we'll need a reference to the `SpriteRenderer` component every time we need to change the sprite, we'll store that in a field for efficiency.

For methods, we'll need the `Start` method to initialize lots of our fields and `Update` and `ClampInScreen` methods to respond to player input. What about handling collisions with a teddy bear?

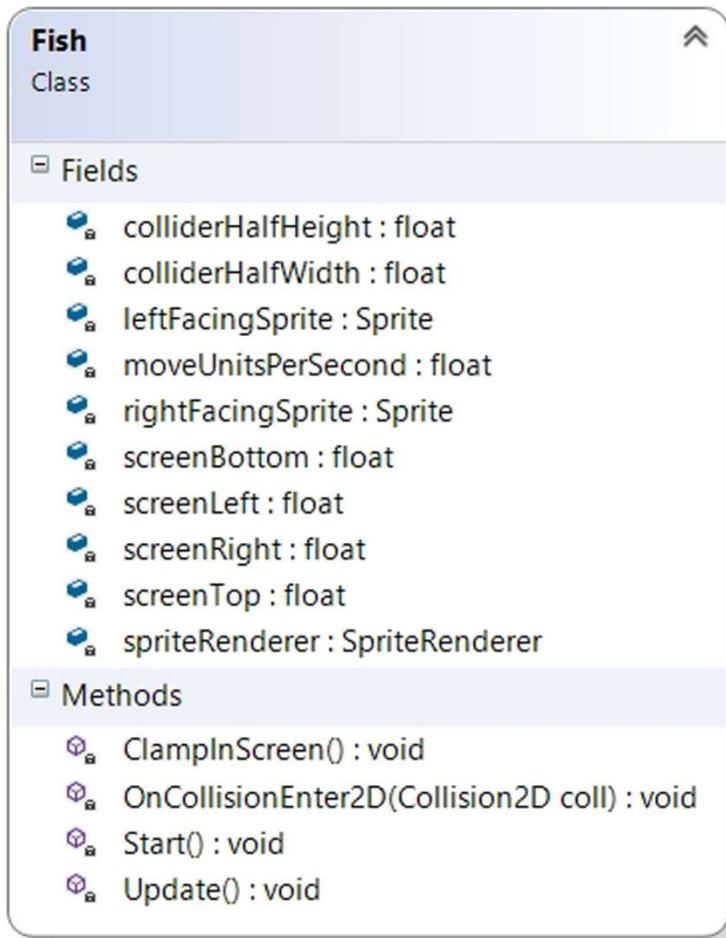
Although it might make sense to you that we should use the `Update` method for this, we should use the `OnCollisionEnter2D` method instead. That's because the `OnCollisionEnter2D` method is called "when an incoming collider makes contact with this object's collider."

But wait a minute (you say), how did we magically know about the `OnCollisionEnter2D` method? Recall that all our C# scripts in Unity are child classes of the `MonoBehaviour` class. One thing that gives us is the ability to include any of the methods listed in the Messages section of the `MonoBehaviour` class (see excerpt below) in our scripts. When we need to respond to a specific event, like a collision or something else that happens in our game, it's always a good idea to check the `MonoBehaviour` Messages section to see if there's a method that we can use there.



**Figure 8.5. MonoBehaviour Messages (Methods)**

Now that we've identified the fields and methods for our `Fish` class (we don't need any properties for this class), we can generate our UML and move on to the Write Test Cases step.

**Figure 8.6. Fish UML**

### *Write Test Cases*

Our first test case is the same one we used to test our "game" in Section 7.6:

#### **Test Case 1**

##### **Checking Teddy Bear and Spawn Behavior**

Step 1. Input: None.

Expected Result: Game runs with the following characteristics:

Each teddy bear dies (is removed from the game) after approximately 10 seconds

New teddy bears appear at random intervals of approximately 1 to 2 seconds

New teddy bears are randomly selected from the 3 teddy bear colors

New teddy bears are spawned at random screen locations

We also need to test the fish movement:

## **Test Case 2**

### **Checking Fish Movement**

Step 1. Input: Left arrow key

Expected Result: Fish moves and faces left

Step 2. Input: Up arrow key

Expected Result: Fish moves up and keeps facing left

Step 3. Input: Right arrow key

Expected Result: Fish moves right and face right

Step 4. Input: Down arrow key

Expected Result: Fish moves down and keeps facing right

Step 5. Input: Left and Up arrow keys simultaneously

Expected Result: Fish moves up and left and faces left

Step 6. Input: Right and Down arrow keys simultaneously

Expected Result: Fish moves down and right and faces right

Finally, we want to make sure the collisions between the fish and a teddy bear work properly:

## **Test Case 3**

### **Checking Fish/Teddy Bear Collisions**

Step 1. Input: Collide with teddy bear with head of fish

Expected Result: Teddy bear removed from game

Step 2. Input: Collide with teddy bear with top, bottom, or tail of fish

Expected Result: Teddy bear bounces off fish

### *Write the Code*

To start, copy the entire folder for the "game" from Section 7.6. into a new folder. This saves us from redoing the work we did there. Run the game to make sure everything still works fine; Test Case 1 should pass. Let's work on getting Test Case 2 to pass next.

Use your operating system to copy the fish sprite sheet into the sprites folder for the project. The fish sprite sheet contains two images (facing right and facing left). These aren't two frames for an animation, they're the images we want to display based on which direction the fish is facing. Select the fish sprite in the sprites folder in the Project window and change the Sprite Mode in the Inspector to Multiple. Use the Sprite Editor to slice the sprite into two separate images (don't forget to click the Apply button before closing the Sprite Editor).

Drag the fish sprite (not fish\_0 or fish\_1, the sprite with the arrow to the left of it) from the sprites folder in the Project window into the Hierarchy window and change the name of the new game object to Fish. Add a Box Collider 2D component and edit the collider to tightly match the body of the fish; the fins should be mostly outside the collider. You can check that the collider fits both fish\_0 and fish\_1 properly by clicking the small circle next to the Sprite field in the Sprite Renderer component and selecting the sprite you want to check.

Add a new `Fish` script to the scripts folder in the Project window and drag the new script onto the Fish game object in the Hierarchy window. Open the `Fish` script in Visual Studio and paste the code from the `TeddyBear` script from the Section 8.3. solution. Change the name of the class to `Fish`, delete the `prefabExplosion` field, delete the code in the `Update` method that blows the game object up on space bar (no exploding fish here!), and change all the comments from teddy bear to fish.

Select the Fish game object in the Hierarchy window and set the Move Units Per Second field in the script in the Inspector to 5. If you run the game now, you should be able to drive the fish around with the keyboard, but it won't face left when you move to the left. You probably also noticed that the teddy bears already bounce off the fish – that's the Unity physics system at work!

Okay, let's get the fish to face left and right correctly. Add `Sprite` fields marked with `[SerializeField]` (so we can populate them in the Inspector) to hold the facing right (`facingRightSprite`) and facing left (`facingLeftSprite`) sprites. Add a field called `spriteRenderer` to hold a reference to the `SpriteRenderer` component. Add code to the `Start` method to save the `SpriteRenderer` component in the `spriteRenderer` field and to set `spriteRenderer.sprite` to `facingRightSprite`.

Now we need to change the code in our `Update` method that moves the fish based on Horizontal input:

```
// set sprite and get new horizontal position
float horizontalInput = Input.GetAxis("Horizontal");
if (horizontalInput < 0)
{
    spriteRenderer.sprite = leftFacingSprite;
    position.x += horizontalInput * moveUnitsPerSecond * Time.deltaTime;
}
else if (horizontalInput > 0)
{
    spriteRenderer.sprite = rightFacingSprite;
    position.x += horizontalInput * moveUnitsPerSecond * Time.deltaTime;
}
```

Our code is a little more complicated because we need to know the direction of the Horizontal input so we can set the sprite for the sprite renderer properly.

Go to the Unity editor and select the Fish game object in the Hierarchy window. Drag the `fish_0` sprite from the Project window onto the Right Facing Sprite field in the script in the Inspector and drag the `fish_1` sprite from the Project window onto the Left Facing Sprite field in the script in the Inspector. At this point, Test Case 2 should pass when you run it.

All we have left is making it so the fish can eat teddy bears with its head. Remember that we'll do that in the `OnCollisionEnter2D` method. To make sure we get the method header correct, we'll copy the entire method from the example in the documentation, add a comment above the method, and delete all the code in the method body (the code between the `{` and the `}`). We'll fill in the method body soon, but we need to figure out what we need to put there first.

Let's talk about our general idea for detecting collisions with the fish's head, then figure out the details. One way to do this would be to use a bounding box around the fish's head; see Figure 8.7.



**Figure 8.7. Fish Head Bounding Box**

When we detect a collision with a teddy bear, if the collider for the teddy bear intersects with the bounding box at the fish's head, we'll eat (destroy) the teddy bear. When a collision occurs, we'll have to first move the fish head bounding box to the appropriate side of the fish (the left if the fish is facing left and the right if the fish is facing right). How do we actually get the collider for the teddy bear, though?

If we look at the method header for the `OnCollisionEnter2D` method, we see there's a `Collision2D` object as the only parameter. The `Collision2D` documentation shows that we can access the `collider` field of that object to get the teddy bear's collider. The collider is a `Collider2D` object, which has a `bounds` field of type `Bounds` for the "world space bounding area of the collider." `Bounds` objects are bounding boxes (see Figure 8.8.), so if we have `Bounds` objects for the teddy bear collider and the fish head, we can check to see if they intersect to determine whether or not the collision occurred at the head of the fish.

The screenshot shows the Unity Documentation website for version 5.6. The left sidebar contains a navigation tree with categories like BillboardAsset, BillboardRenderer, BitStream, BoneWeight, BoundingSphere, Bounds (which is selected and highlighted in black), BoxCollider, BoxCollider2D, BuoyancyEffector2D, Caching, Camera, Canvas, CanvasGroup, CanvasRenderer, CapsuleCollider, CapsuleCollider2D, CharacterController, CharacterInfo, CharacterJoint, CircleCollider2D, Cloth, ClothSkinningCoefficient, ClothSphereColliderPair, ClusterInput, ClusterNetwork, and History. The main content area is titled 'Bounds' and describes it as a struct in UnityEngine. It includes sections for 'Description' (representing an axis aligned bounding box), 'Variables' (center, extents, max, min, size), 'Constructors' (Bounds), and 'Public Functions'. A search bar at the top right allows searching for scripting API.

**Figure 8.8. Bounds Documentation**

So how did we know about all these different classes and fields? We didn't! We just started at the `OnCollisionEnter2D` documentation and used the links to explore what information was available in each of the classes and fields to figure out what would help us solve our current problem. You'll find yourself doing this all the time as a game developer, even an experienced game developer!

It looks like we have all the pieces we need, so let's start implementing the actual code. We'll start by adding several fields to the `Fish` class:

```
// head bounding box support
Bounds headBoundingBox;
const float HeadPercentageOfCollider = 0.2f;
Vector3 headBoundingBoxLocation;
float headBoundingBoxXOffset;
BoxCollider2D fishCollider;
```

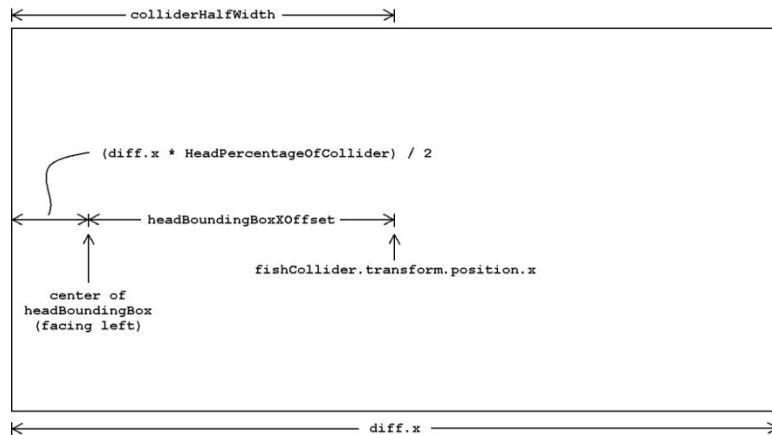
The `headBoundingBox` field holds the bounding box for the fish's head. The constant tells us how much of the total fish collider represents the head of the fish; this will be useful when we create the bounding box. We'll use the `headBoundingBoxLocation` field to move the bounding box to the correct side of the fish on a collision. We save the x offset we need to apply from the center of the collider to move the bounding box the correct amount to get it to the front of the fish, and we save a reference to the `BoxCollider2D` component of the Fish so we don't have to look that up every time we have a collision.

For the `headBoundingBoxLocation` field, we could create a new `Vector3` object every time in the `OnCollisionEnter2D` method instead. Those objects would simply be used once and then take up memory until the garbage collector runs, though, so we avoid that approach. We can't actually control when the garbage collector runs, and it can run at very inopportune times, so if we avoid creating garbage we delay the need to collect garbage. For many games and platforms this won't matter at all, but a good general rule of thumb is to avoid repeatedly creating "short lifetime" objects if possible.

The next thing we do is add code to the `Start` method to initialize the fields we just added:

```
// initialize head bounding box fields
headBoundingBoxXOffset = colliderHalfWidth -
    (diff.x * HeadPercentageOfCollider) / 2;
headBoundingBoxLocation = new Vector3(
    fishCollider.transform.position.x + headBoundingBoxXOffset,
    fishCollider.transform.position.y,
    fishCollider.transform.position.z);
headBoundingBox = new Bounds(headBoundingBoxLocation,
    new Vector3(diff.x * HeadPercentageOfCollider,
    diff.y, 0));
```

We obviously had some math to do here! For the calculation of the `headBoundingBoxXOffset`, we drew a picture to help us figure out the required equation. We draw pictures all the time as we figure out how to calculate stuff, though they're usually not quite as neat as Figure 8.9! By the way, we already have the `diff` variable from our earlier work in the method saving the collider dimension value.



**Figure 8.9. Head Bounding Box X Offset Calculation**

Because we'll be changing the `center` field of `headBoundingBox` to move it to the correct side of the fish (yes, we read the documentation again), we need half the width of the bounding box for the fish's head; that's what we're calculating using `(diff.x * HeadPercentageOfCollider) / 2`. If we subtract that value from `colliderHalfWidth`, we have exactly the amount we need to subtract (if we're facing left) from the `center` of the fish collider (`fishCollider.transform.position.x`) to put the head bounding box on the left side of the fish. If the fish is facing right, we'll simply add the x offset instead of subtracting it. Whew!

The second line of code creates a new `Vector3` for the location of the head bounding box. We set the location as though the fish is facing right, but we'll change both the x and y components of this vector when we're in the `OnCollisionEnter2D` method so the bounding box location is consistent with the current location and orientation of the fish.

The final line of code calls the `Bounds` constructor to create a new object for our `headBoundingBox` field. The constructor takes 2 arguments: a `Vector3` for the center of the bounding box and a `Vector3` for the size (width, height, and depth) of the bounding box.

Don't worry, the code we need in the `OnCollisionEnter2D` method is much less "math intensive"! Here's our implementation of that method:

```

/// <summary>
/// Checks whether or not to eat a teddy bear
/// </summary>
/// <param name="coll">collision info</param>
void OnCollisionEnter2D(Collision2D coll)
{
    // move head bounding box to correct location
    headBoundingBoxLocation.y = fishCollider.transform.position.y;
    if (spriteRenderer.sprite == leftFacingSprite)
    {
        headBoundingBoxLocation.x = fishCollider.transform.position.x -
            headBoundingBoxXOffset;
    }
    else
    {

```

```

        headBoundingBoxLocation.x = fishCollider.transform.position.x +
            headBoundingBoxXOffset;
    }
    headBoundingBox.center = headBoundingBoxLocation;

    // destroy teddy bear if it collides with head bounding box
    if (coll.collider.bounds.Intersects(headBoundingBox))
    {
        Destroy(coll.gameObject);
    }
}

```

When we did our initial implementation of the `OnCollisionEnter2D` method, we discovered we weren't destroying any of the teddy bears, even if we collided them with the head of the fish. We decided we wanted to make the teddy bears move much more slowly so we could debug the problem; at that point, we marked the `minImpulseForce` and `maxImpulseForce` variables in the `TeddyBear` class with `[SerializeField]` so we could manipulate those values in the Inspector. We actually set both to 0 for debugging – it's easy to run into a stationary teddy bear<sup>22</sup>! By the way, the bug was that we calculated the new `headBoundingBoxLocation` correctly but forgot to set the `headBoundingBox` center field to it.

When we ran the game, we could eat teddy bears with the head of the fish, but we had to be pretty precise with the collision. From a gameplay perspective, that was really irritating, so we actually made the `headBoundingBox` stick out from the front of the collider a small amount and also made it slightly taller than the actual collider (see the code accompanying the chapter). All the collisions still worked properly, but the change made it easier to actually eat teddy bears, making the game much more fun to play.

There are of course a number of other ways to solve the problem of determining whether or not the collision occurs at the fish's head. One alternative we considered was to have a normal collider for the 80% of the fish that's not the head and a trigger collider (a collider with the Is Trigger box checked) for the 20% of the fish that's the head. We decided not to use that approach because every time the fish changed its horizontal direction, we'd need to move the collider and the trigger collider to the back and front of the Fish game object. We prefer to use our approach, where we only have to move the head bounding box to the front of the fish when a collision occurs, but we wanted to remind you that there are other solutions to this problem.

We just have a little cleanup to do before we move on to the Test the Code step. Drag the Fish game object from the Hierarchy window into the prefabs folder in the Project window to make a Fish prefab. Although that doesn't really help us in this game, if we built multiple scenes it would make it much easier to spawn a Fish at the start of each scene.

### *Test the Code*

As a reminder, here are our test cases:

---

<sup>22</sup> If you set them both to 0, you can also actually "herd" the teddy bears around with the fish. Not that you'd have fun doing something like that ...

### **Test Case 1**

#### **Checking Teddy Bear and Spawn Behavior**

Step 1. Input: None.

Expected Result: Game runs with the following characteristics:

Each teddy bear dies (is removed from the game) after approximately 10 seconds

New teddy bears appear at random intervals of approximately 1 to 2 seconds

New teddy bears are randomly selected from the 3 teddy bear colors

New teddy bears are spawned at random screen locations

### **Test Case 2**

#### **Checking Fish Movement**

Step 1. Input: Left arrow key

Expected Result: Fish moves and faces left

Step 2. Input: Up arrow key

Expected Result: Fish moves up and keeps facing left

Step 3. Input: Right arrow key

Expected Result: Fish moves right and face right

Step 4. Input: Down arrow key

Expected Result: Fish moves down and keeps facing right

Step 5. Input: Left and Up arrow keys simultaneously

Expected Result: Fish moves up and left and faces left

Step 6. Input: Right and Down arrow keys simultaneously

Expected Result: Fish moves down and right and faces right

### **Test Case 3**

#### **Checking Fish/Teddy Bear Collisions**

Step 1. Input: Collide with teddy bear with head of fish

Expected Result: Teddy bear removed from game

Step 2. Input: Collide with teddy bear with top, bottom, or tail of fish

Expected Result: Teddy bear bounces off fish

All our test cases pass, so we're done.

# Chapter 9. Arrays and Collection Classes

We're really starting to build a good foundation of C# and Unity knowledge – we know how to create objects from classes and use them, we know how to use selection to make our games do the things we want them to do, and we know how to do more basic things like declaring variables and constants as well. We also know how to effectively use some of the classes provided by the C# language and the Unity engine, an approach we've used extensively throughout the book. Strictly speaking, that's sufficient to solve lots of the problems we'd encounter in an introductory course (don't worry, we'll get to the final control structure in the next couple of chapters), but there may be some cases in which simply using the above knowledge would be somewhat awkward.

For example, suppose we had 12,000 students at a university and we wanted to store the GPAs for all of them. We'll learn an effective way to read them in in the next chapter (though somebody's fingers will hurt typing them in until we learn how to use files), but where do we put them all? With our current knowledge we could declare 12,000 distinct variables, one for each GPA, but there's got to be a better way! There is, of course: we can use an *array* to store all the GPAs. With an array, we only need to do a little more work than we do when we declare any other kind of variable. Let's take a closer look.

## 9.1. Declaring and Creating Array Objects

The reason we have to do a little more work is because we need to create a new object for the array variable. The syntax is:

---

SYNTAX: Creating an Array Variable

```
elementType[] variableName = new elementType[arraySize];
```

*elementType*, the data type for each element in the array

*variableName*, the name of the array variable

*arraySize*, the number of elements in the array being created

---

This syntax looks a lot like what we used to create objects. That's for a very good reason – arrays in C# ARE objects!

To make this more concrete, let's work through an example.

*Example 9.1. Storing GPAs for 10 students*

Problem Description: Declare and create the array variable required to store 10 GPAs.

And here's how we do it:

```
float[] gpas = new float[10];
```

With the above array variable creation, our array will consist of 10 "boxes" or *elements*. In C# arrays (as in many other programming languages), the elements are numbered starting at 0, so the elements of our

array will be numbered from 0 to 9. The number for a particular element is called the *index* of that element. Each element in the array will be a `float`, which also seemed to be a clear choice given that each element holds a GPA. No matter what data type we pick for the array elements, every single element of the array will be of that type. In other words, we can't have some `float` elements, some `int` elements, etc. in a single array.

A pictorial representation of `gpas` appears in Figure 9.1. The array consists of 10 elements, numbered from 0 to 9, and each element holds a `float`. Note that if the elements of the array are a value type, then each element of the array is initialized with the same initial value a variable of that data type would be initialized to. For `float`, that initial value is 0.0.

0	0.0
1	0.0
2	0.0
3	0.0
4	0.0
5	0.0
6	0.0
7	0.0
8	0.0
9	0.0

**Figure 9.1. Pictorial Representation of `gpas`**

So that's how we declare and create an array variable. And if we wanted to change this array variable to hold 12,000 students instead of 10, all we'd have to do is change the `10` we put between the square brackets above to `12000`!

There's actually an alternate way to create an array object, where we also assign values to the array elements when we create the object; the syntax for that is provided below. When we use this syntax, we

provide the values for each of the array elements between curly braces, separated by commas. The first value provided goes into element 0 of the array, the second value goes into element 1, and so on. Notice that we don't have to explicitly say how many elements are in the array; C# simply creates an array object with just enough elements to hold the values we provide.

---

#### SYNTAX: Creating and Initializing an Array Variable

```
elementType[] variableName = { value0, value1, ... };
```

*elementType*, the data type for each element in the array

*variableName*, the name of the array variable

*value0, value1, ..., valueN*, the initial values for the elements in the array

---

## 9.2. Accessing Elements of the Array

Now that we have a variable for the array, how do we use it? Well, we use the variable in the same way we've used other variables – the only real difference is that we do assignments, input, output, and calculations with single elements of the array rather than the entire array<sup>23</sup>. Let's make this a little more concrete; the syntax description below shows us the proper syntax for these operations.

---

#### SYNTAX: Array Element Operations

##### Assignment

```
variableName[index] = value;
```

*variableName*, the name of the array variable

*index*, the number of the element in which we want to store the value

*value*, the value we want to put into that element

##### Input (for an array of `float` elements)

```
variableName[index] = float.Parse(Console.ReadLine());
```

*variableName*, the name of the array variable

*index*, the number of the element in which we want to store the user input

##### Output

```
Console.WriteLine(variableName[index]);
```

*variableName*, the name of the array variable

*index*, the number of the element we want to output

---

<sup>23</sup>In fact, there are some operations we can perform on entire arrays. They won't really be useful for the problems we'll solve in this book, though, so we won't bother covering them here.

### Calculations (one example)

```
sum += variableName[index];
```

*variableName*, the name of the array variable  
*index*, the number of the element we want to add to *sum*

---

The key difference between array variables and the other kinds of variables we've used up to this point is that we have to say which element of the array we want to use. The bottom line for all these accesses is that we reference a particular element in the array by providing the array variable name followed by an index between square brackets. For example, if we wanted to set the first element of our *gpas* array to 4.0, we could simply say

```
gpas[0] = 4.0;
```

Similarly, if we wanted to print out the 8<sup>th</sup> element of the *gpas* array, we could use

```
Console.WriteLine("8th GPA: " + gpas[7]);
```

Remember, because we start counting at 0 the 8<sup>th</sup> GPA is at index 7. We could also read a GPA into the 0<sup>th</sup> element using

```
Console.Write("Enter First GPA: ");
gpas[0] = float.Parse(Console.ReadLine());
```

Finally, if we were adding the last element in *gpas* to a variable called *sum*, for example, we'd use

```
sum += gpas[9];
```

## 9.3. Arrays of Objects

Our arrays up to this point have been arrays containing value types. But there might be times when we want to use an array of objects, right? Can we do that also? You bet.

This will be easier to understand if we work through an example together, so let's build an array containing 5 *Card* objects (remember the *Card* class from the Classes and Objects chapter?). Based on the syntax for creating an array variable as described in section 9.1, we'd use:

```
Card[] hand = new Card[5];
```

This will give us an array of 5 elements, each of which will be a *Card* object. But wait a minute, you say. Even though we've created the array object, don't we need to actually create each of the card objects contained in the array? Right you are! At this point, each element in the array is *null* because that's what reference types get for their initial value.

Let's fill the array with some really good cards:

```
// fill hand with cards
hand[0] = new Card("Ten", "Spades");
hand[1] = new Card("Jack", "Spades");
hand[2] = new Card("Queen", "Spades");
hand[3] = new Card("King", "Spades");
hand[4] = new Card("Ace", "Spades");
```

Now, we hope you have one more question about arrays of objects. This would be a good question: How do you access a property or call a method for one of the array elements? For example, how would we print the `Rank` and `Suit` properties of the third card in the array?

It's actually really easy. Remember that we refer to an element in the array by using the syntax

```
variableName[index]
```

Well, we do the same thing with our objects in the array. So we can access the `Rank` property for the third card in the array using

```
hand[2].Rank
```

Because we know the `hand` array holds `Card` elements, we know that `hand[2]` is a `Card` object. That means we can treat `hand[2]` like any other `Card` object, so we can access its properties and methods using the standard dot notation. So we could print the rank and suit of the third card in the array using

```
Console.WriteLine(hand[2].Rank + " of " + hand[2].Suit);
```

Arrays of objects can be useful, and they're pretty easy to use too. Just remember that you need to create both the array AND the object for each element of the array and you'll be fine.

## 9.4. Refactoring the Card Class

We've already done some refactoring in previous problem solutions, but as a quick reminder refactoring means changing the structure of the code with the goal of making it faster or easier to understand. When we developed the `Card` class in Chapter 4, we didn't know about enumerations yet; that's why we used `string` for the rank and suit of the card. But we know better now, so it's time to refactor! Let's define two enumerations for use in our `Card` class. Here's the first one:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RefactoredCards
{
    /// <summary>
    /// An enumeration for card ranks
    /// </summary>
    public enum Rank
    {
        Ace,
        Two,
        Three,
```

```

        Four,
        Five,
        Six,
        Seven,
        Eight,
        Nine,
        Ten,
        Jack,
        Queen,
        King
    }
}

```

**Figure 9.2.** Rank.cs

and the second one is even shorter:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RefactoredCards
{
    /// <summary>
    /// An enumeration for card suits
    /// </summary>
    public enum Suit
    {
        Clubs,
        Diamonds,
        Hearts,
        Spades
    }
}

```

**Figure 9.3.** Suit.cs

Now that we have enumerations for card rank and suit, we change the `Card` class to use those enumerations instead of `string`; the refactored class is shown below.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RefactoredCards
{
    /// <summary>
    /// A playing card
    /// </summary>
    public class Card
    {
        #region Fields

```

```
Rank rank;
Suit suit;
bool faceUp;

#endregion

#region Constructors

/// <summary>
/// Constructs a card with the given rank and suit
/// </summary>
/// <param name="rank">the rank</param>
/// <param name="suit">the suit</param>
public Card(Rank rank, Suit suit)
{
    this.rank = rank;
    this.suit = suit;
    faceUp = false;
}

#endregion

#region Properties

/// <summary>
/// Gets the card rank
/// </summary>
public Rank Rank
{
    get { return rank; }
}

/// <summary>
/// Gets the card suit
/// </summary>
public Suit Suit
{
    get { return suit; }
}

/// <summary>
/// Gets whether or not the card is face up
/// </summary>
public bool FaceUp
{
    get { return faceUp; }
}

#endregion

#region Public methods

/// <summary>
/// Flips the card over
/// </summary>
```

```

    public void FlipOver()
    {
        faceUp = !faceUp;
    }

    #endregion
}
}

```

**Figure 9.4. Card.cs**

We changed a number of things in the `Card` class. First, we changed the `rank` field to be a `Rank` rather than a `string` and we changed the `suit` field to be a `Suit` rather than a `string`. This gives us all the benefits of enumerations that we discussed previously for these fields.

We also needed to change the parameters for the constructor to be a `Rank` and a `Suit` rather than the two `string` parameters that we originally had. The last thing we did was change the return types for the `Rank` and `Suit` properties. For example, we changed the first line of the `Rank` property from

```
public string Rank
```

to

```
public Rank Rank
```

This might now look a little confusing to you, but if you remember that the first `Rank` in the line above is the data type for the property and the second `Rank` is the actual name of the property it should be clear.

Some beginning programmers might try to avoid confusing themselves by using something like

```
public Rank CardRank
```

instead. This is a bad approach, though, because when we access the property using a `Card` object called `myCard` (for example) we'd have to use `myCard.CardRank` instead of `myCard.Rank`; the `CardRank` name is less intuitive. The bottom line is that making things a little clearer for the class developer makes things harder for the class consumer, and our goal should always be to make our classes as easy to use as possible.

Speaking of the consumer of the class, we're done refactoring the `Card` class itself so we should look at how those changes affect consumers of the class. We still declare the array of `Card` objects the same way we did before, so the first place we need to change is the code that actually fills the hand with cards. Recall that we used to have

```
// fill hand with cards
hand[0] = new Card("Ten", "Spades");
hand[1] = new Card("Jack", "Spades");
hand[2] = new Card("Queen", "Spades");
hand[3] = new Card("King", "Spades");
hand[4] = new Card("Ace", "Spades");
```

which we need to change to

```
// fill hand with cards
hand[0] = new Card(Rank.Ten, Suit.Spades);
hand[1] = new Card(Rank.Jack, Suit.Spades);
hand[2] = new Card(Rank.Queen, Suit.Spades);
hand[3] = new Card(Rank.King, Suit.Spades);
hand[4] = new Card(Rank.Ace, Suit.Spades);
```

We used to pass two `string` arguments to the constructor, but we now need to pass a `Rank` argument and a `Suit` argument to the constructor. What about our output? We leave it exactly the same as it was! Here's the line of code for printing information about the third card:

```
Console.WriteLine(hand[2].Rank + " of " + hand[2].Suit);
```

When the `Rank` and `Suit` properties returned `strings`, it was easy to see how the above code would print the concatenation of the three `strings`. It's less clear, however, why this works now that they return `Rank` and `Suit` objects.

Because the properties are accessed as part of the argument to the call to `Console.WriteLine`, the values of those properties are automatically converted to `strings`. This has been happening all along, of course; when we include an `int` variable in a call to `Console.WriteLine`, the value of that variable is automatically converted to a `string`. This just seemed like the right time to talk about it.

That concludes our discussion of the refactored `Card` class, so let's take a look at multi-dimensional arrays.

## 9.5. Multi-Dimensional Arrays

So far, the only arrays we've dealt with have been one-dimensional. Can we make multi-dimensional arrays in C#? Sure we can! Let's say we wanted an array with 3 rows and 4 columns of integers. Here's how we declare and create our array variable:

```
int[,] grid = new int[3,4];
```

then we access elements of this array using the general form

```
variableName[rowNumber, columnNumber]
```

For example, to put a 1 into the upper left corner of the array (row 0, column 0), we'd say

```
grid[0,0] = 1;
```

Columns are numbered from left to right, and rows are numbered from top to bottom. We're not limited to two-dimensional arrays either; we can essentially use as many array dimensions as we want.

## 9.6. Collection Classes

Arrays are very useful for storing multiple values that are all the same data type, but there is one significant limitation. We have to know the maximum number of elements that will be in the array when we create the array object using `new`. This adds some complexity to processing the array in a number of

ways. If the array is only partially filled, for example, we need to keep track of how many elements are currently in the array so we know where to add the next element in the array. If we end up filling the array and needing more space because we misjudged how many elements we'd need to store, we'd need to create a new (larger) array and copy all the elements from the old array to the new one. Arrays are very powerful structures that are provided by most modern programming languages, but they have some important limitations when we need to dynamically grow the arrays.

C# provides a set of built-in *collection classes* that help solve this problem. Basically, these classes let us store a collection of elements, and they also provide lots of useful properties and methods for manipulating those collections. We'll focus on the `List` class, which is in the `System.Collections.Generic` namespace, but you can take a look at the complete listing of collections by searching the documentation for the `System.Collections.Generic` and `System.Collections` namespaces.

The `List` class is called a *generic class* because it's generic enough to hold elements of any type you want. You will, however, have to specify what that type is when you create the new `List` object. In other words, all the elements in a list need to be all the same data type just like they were in an array.

Let's make our hand of cards a `List` rather than an array. This is actually a really good idea, because typically the number of cards in a hand changes a lot, and that's much easier to deal with using a `List`. Here's how we initialize the hand:

```
List<Card> hand = new List<Card>();
```

This looks a lot like creating objects of any other type, except we have the `<Card>` part added both when we declare the type (before the variable name) and when we create an object of the type (after the `new`). Remember, the `List` class is a generic class, so we have to tell it the data type of the elements it will hold. We do that between the `<` and the `>`, so the above code declares and creates a `List` object that will hold `Card` elements.

As we said above, there are lots of useful properties and methods provided by the `List` class; some of the most useful ones for beginning programmers are shown below (of course, the full list is available from the documentation).

## Methods

Add

Adds an element to the end of the list

Clear

Removes all the elements from the list

Contains

Determines whether a particular value is in the list

IndexOf

Returns the zero-based index of the first occurrence of a particular value

Remove

Removes the first occurrence of a particular value from the list

**Property****Count**

Gets the number of elements contained in the list

Given the above information, we now know how to fill the hand with the 5 cards we used before:

```
// fill hand with cards
hand.Add(new Card(Rank.Ten, Suit.Spades));
hand.Add(new Card(Rank.Jack, Suit.Spades));
hand.Add(new Card(Rank.Queen, Suit.Spades));
hand.Add(new Card(Rank.King, Suit.Spades));
hand.Add(new Card(Rank.Ace, Suit.Spades));
```

Although the list of members and properties above are useful, they don't tell us how to actually access a specific element in the list. Under the hood, we do that by accessing the `Item` property of the list, but rather than using the syntax we're used to using for properties, we use the square brackets just as we did to access specific elements of an array. In other words,

```
Console.WriteLine(hand[2].Rank + " of " + hand[2].Suit);
```

This works exactly the same way on our `List` as it did when we were using an array. Pretty slick, huh?

Although we'll use the `List` class extensively throughout this book from now on, there are lots of collection classes that you'll find useful as you pursue more advanced programming projects.

## 9.7. Putting It All Together

Let's get a little more practice using the `List` class by solving the following problem:

- Start with a TeddyBear game object, centered in the window, not moving
- On every right mouse click, add a Pickup game object where the mouse was clicked
- When the player left clicks the TeddyBear, the teddy starts collecting the pickups in the order in which they were placed
- The TeddyBear collects a Pickup by colliding with it, but this only works for the Pickup the Teddy has currently "targeted for collection"
- Once the last Pickup has been collected, the Teddy stops moving
- If the player adds more Pickups while the Teddy is moving, the Teddy picks them up as well
- If the player adds more Pickups while the Teddy is stopped, the player has to left click on the Teddy again to start it collecting again

### *Understand the Problem*

Wow, that problem seems quite a bit more complicated than we've solved so far, even at the ends of the previous chapters. That's okay, though, because it will give us a chance to practice using Lists and will help us enhance our Unity skills as well.

The required program behavior should be clear, even though we have lots of other work ahead of us.

## Design a Solution

First of all, let's identify the active game objects in our game. It turns out that we only have two kinds: the Teddy Bear and the Pickups. We'll definitely want a Pickup prefab because we'll be creating new pickup objects in the scene when the player right clicks on the screen. Although we'll only have a single Teddy Bear game object in the game, we'll create a TeddyBear prefab also.

What about scripts? We'll start by figuring out what scripts we need for the active game objects, then identify any additional scripts we need. We can immediately tell that we'll need a `TeddyBear` script because the Teddy Bear game object has specific behaviors it has to perform during the game (like starting to collect pickups when the mouse is left clicked on it).

Do we need a `Pickup` script? No, we don't. The Pickups in the game don't actually do anything, they just sit there waiting to be picked up.

That's it for the active game objects. Do we need any higher-level scripts to run the game? That depends on how we want to keep track of the Pickups currently waiting to be picked up. One approach would be to hold those in a list in the `TeddyBear` script, in which case we don't need any more scripts.

We're going to take a different approach here. We'll write a `TedTheCollector` script, which we'll attach to the Main Camera, to hold the list of Pickups in the game. From an object-oriented perspective, it doesn't really make sense to have the Teddy Bear keep track of other kinds of game objects in the game; instead, the Teddy Bear should just keep track of its own state and behavior. We'll find that we almost always need a high-level "game manager script" that handles game-level kinds of things, and this is a better object-oriented approach as well. We'll start following that approach in our solution to this problem.

Let's design our `TeddyBear` script first, starting with the fields. We really only need to keep track of whether or not the Teddy Bear is currently collecting Pickups because we only want to respond to left clicks on the Teddy Bear if it's not currently collecting; that field should be a `bool`. You might think we'd also need a field to store the next Pickup the Teddy Bear needs to collect (we certainly thought that!), but when we Write the Code you'll see we don't actually need that field.

We'll need two methods (at this point) as well, but they're already provided in the default script. We'll use the `Start` method to make sure the Teddy Bear is centered in the screen when the game starts and we'll use the `Update` method to respond to left mouse clicks when they occur on the Teddy Bear. We'll discover later on that we actually need another method, but we'll add that when we discover that we need it.

The UML for the `TeddyBear` script is shown in Figure 9.5. Note that we don't need any properties for this script since there won't be any consumers that need access to its state.



**Figure 9.5. TeddyBear UML**

For the `TedTheCollector` script, we know we'll need a list of the Pickups that are currently in the game; we'll make that a list of `GameObjects`s. That's actually the only state information we need for this script.

Because the `TedTheCollector` script is managing the list of Pickups in the game, we need to keep the default `Update` method so we can detect right mouse clicks and add a Pickup to the game when that happens.

We'll also, for the first time in the book, need some properties and methods (in Unity) that other classes use rather than just having methods for "internal use." Let's think about why we need those properties and methods by thinking about how the game works.

When the `TeddyBear` `Update` method detects that the left mouse button has been left clicked on the Teddy Bear game object the script is attached to, it needs to start moving toward the oldest Pickup in the game. The `TedTheCollector` script is the class that has the list of Pickups, though, so it should expose a `TargetPickup` property that returns the oldest Pickup in the game. The `TeddyBear` class can then access that property to get its target when it's left clicked.

We also need to think about what happens when the Teddy Bear collides with the Pickup that it's currently targeting. The Teddy Bear should get its next target (which it can get from the `TedTheCollector` `TargetPickup` property), but the Pickup that has just been collected should also be removed from the game. Again, the `TedTheCollector` script is the class that manages the Pickups in the game, so it should expose a `RemovePickup` method that removes a given Pickup from the list of Pickups and destroys that pickup to remove it from the game. The `TeddyBear` script can then call that method when it collects a Pickup.

The UML for the `TedTheCollector` class is shown in Figure 9.6.



**Figure 9.6. TedTheCollector UML**

### *Write Test Cases*

We can do all our testing in a single test case as shown below.

#### **Test Case 1**

##### **Checking Game Behavior**

Step 1. Input: Right Click

Expected Result: Pickup placed at click location

Step 2. Input: Left Click on Teddy Bear

Expected Result: Teddy Bear collects Pickup and stops

Step 3. Input: Left Click on Teddy Bear

Expected Result: No response (there's no pickup to collect)

Step 4. Input: Right Click

Expected Result: Pickup placed at click location

Step 5. Input: Right Click

Expected Result: Pickup placed at click location

Step 6. Input: Left Click on Teddy Bear

Expected Result: Teddy Bear collects oldest Pickup then moves toward next pickup

Step 7. Input: Right Click between Teddy Bear and Pickup while Teddy Bear is moving toward Pickup

Expected Result: Pickup placed at click location, Teddy Bear collects oldest Pickup, then collects final Pickup, then stops

We have a lot of steps here, but this test case thoroughly tests the required game behavior. Notice that Step 7 has a timing constraint on the input so we can make sure Pickups that are added while the Teddy Bear is moving are collected properly.

### *Write the Code*

We start by creating a new Unity 2D project, saving the scene into a scenes folder, importing a teddy bear sprite and a pickup sprite into a sprites folder, and creating an empty prefabs folder. Next, we'll build the Pickup and TeddyBear prefabs, then move on to the scripting piece. We'll alternate between the `TeddyBear` and `TedTheCollector` scripts as we implement the game functionality a little at a time. Drag the teddy bear sprite from the sprites folder in the Project window and drop it in the Hierarchy window. Change the name of the game object in the Hierarchy window to `TeddyBear`. Create a new scripts folder in the Project window, then create a new C# Script called `TeddyBear` in the scripts folder. Drag the `TeddyBear` script onto the `TeddyBear` game object in the Hierarchy window, then drag the

TeddyBear game object from the Hierarchy window onto the prefabs folder in the Project window. We now have the TeddyBear prefab we need for the game.

Next, drag the pickup sprite from the sprites folder in the Project window and drop it in the Hierarchy window. Change the name of the game object in the Hierarchy window to Pickup. Drag the Pickup game object from the Hierarchy window onto the prefabs folder in the Project window. We now have the Pickup prefab we need for the game. Since we don't want any Pickups in the scene when we start the game, right-click the Pickup game object in the Hierarchy window and Delete it.

Create another C# script called `TedTheCollector` in the scripts folder in the Project window and drag that script onto the Main Camera.

Okay, we're ready to start working on our scripts. Because we know the fields, properties, and methods we need in each script from our design work, we'll implement the fields and *interface* (the properties and methods each script exposes) for each script, then implement the actual functionality iteratively. As we do this, we'll create *stubs* for the `TargetPickup` property and the `RemovePickup` method in the `TedTheCollector` class. A stub basically has the minimum amount of code required to make it compile. As we iteratively write our code, we add the required functionality to our classes until we've "fleshed out" all the stubs (and added any other fields, properties, and methods we discovered we needed).

Let's start with the `TeddyBear` script; our initial cut at the code is shown below:

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

/// <summary>
/// A collecting teddy bear
/// </summary>
public class TeddyBear : MonoBehaviour
{
    #region Fields

    bool collecting = false;

    #endregion

    #region Methods

    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
    {
        // center teddy bear in screen
        Vector3 position = transform.position;
        position.x = 0;
        position.y = 0;
        position.z = 0;
        transform.position = position;
    }
}
```

```

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

#endregion
}

```

We initialize the `collecting` field to `false` because the teddy bear isn't collecting when it's added to the scene at run time.

In the `Start` method, we're making sure the Teddy Bear is centered in the screen and at  $z == 0$ ; with the camera aiming at the origin in world coordinates, putting the Teddy Bear at  $(0, 0)$  in x and y centers it in the screen. The code that does that may seem more complicated to you than is necessary, but we can't change the components of `transform.position` directly. That means we need to copy it into a local `Vector3` variable (our `position` variable), make the changes we want to that local variable, then copy the local variable back into `transform.position`. Although our Teddy Bear prefab defaults to being at  $(0, 0, 0)$  in x, y, and z, including the centering code in the `Start` method ensures that inadvertent position changes in the Unity editor don't result in incorrect program behavior for the centered on startup requirement.

If you run the game now, you'll see that the game starts with the Teddy Bear centered on the screen. You can even verify the centering by changing the location of the `TeddyBear` game object in the scene in the Unity editor; when you run the game, the Teddy Bear is centered as required.

Let's move over to the `TedTheCollector` script; our starting point for that code is shown below:

```

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

/// <summary>
/// Game manager
/// </summary>
public class TedTheCollector : MonoBehaviour
{
    #region Fields

    List<GameObject> pickups = new List<GameObject>();

    #endregion

    #region Properties

    /// <summary>
    /// Gets the next target pickup for the teddy bear to collect
    /// </summary>
    /// <value>target pickup</value>
    public GameObject TargetPickup
    {

```

```

        get { return null; }
    }

#endregion

#region Methods

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{

}

/// <summary>
/// Removes the given pickup from the game
/// </summary>
/// <param name="pickup">the pickup to remove</param>
public void RemovePickup(GameObject pickup)
{
}

#endregion
}

```

When we created the `pickups` field we also called the `List` constructor to create our new `List` object; that way we don't have to create the `List` object the first time we want to add a Pickup to the field. Finally, the `TargetPickup` property and the `Update` and `RemovePickup` methods are stubs as discussed above.

Of course, the game behaves exactly the same as it did before when we run it because we're not doing anything in this script yet.

Okay, we've been doing a lot of work so far, but if we were to run our test case we wouldn't even get past Step 1! Let's fix that now by making it so a Pickup is added to the game on a right click. As we discussed in the Design a Solution step, we'll do this in the `TedTheCollector` `Update` method. As soon as we try to do that, though, we realize that we need another field in the `TedTheCollector` class: specifically, we need a field for the Pickup prefab so we can add a new instance of that prefab to the game on a right click.

It's important that you realize that our design will almost always evolve as we learn more details during the Write the Code step. This is a perfectly normal occurrence, and it doesn't mean we did our design "wrong"; this is just how it works. You may hear some people claim that you should get your design perfectly correct and complete before you start coding, but this is almost never feasible in practice. That's why the development we demonstrate throughout the book shows a more typical sequence of actions rather than an ideal, unrealistic process.

Add a `prefabPickup` field (as a `GameObject`) to the `TedTheCollector` class and mark it with `[SerializeField]`, go to the Unity editor, select the Main Camera in the Hierarchy window, and drag the Pickup prefab from the prefabs folder in the Project window onto the new field in the Inspector.

Now we can implement the `TedTheCollector` `Update` method:

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // add pickup on right click
    if (Input.GetMouseButtonDown(1))
    {
        // calculate world position of mouse click
        Vector3 mousePosition = Input.mousePosition;
        mousePosition.z = -Camera.main.transform.position.z;
        Vector3 worldPosition = Camera.main.ScreenToWorldPoint(mousePosition);

        // create pickup and add to list
        GameObject pickup = Instantiate<GameObject>(prefabPickup);
        pickup.transform.position = worldPosition;
        pickups.Add(pickup);
    }
}
```

From the Unity scripting manual, the `Input` `GetMouseButtonDown` method "Returns true during the frame the user pressed the given mouse button." Furthermore, the documentation shows that we use 0 for the left button and 1 for the right button, so the Boolean expression for our if statement evaluates to `true` on the first frame in which the right button is pressed. You should be able to see why we don't want to place a pickup on every frame the right button is pressed – that would be a lot of pickups! – just on the first frame.

Strictly speaking, this isn't actually a mouse click, which consists of a mouse button press followed by a release of that mouse button. You'll probably actually run into situations in your game development where you need to detect a "true mouse click", but in this problem we'll just place the pickup on the first frame in which the mouse button is pressed. That will actually feel more natural to the player, because it wouldn't feel as responsive if the pickup were placed on the button release of a true mouse click instead.

The first line of code in the if body gets a copy of the mouse position so that the second line can change the z coordinate just like we did when we were spawning teddy bears in Section 7.6. Recall that we needed to do that so that all our 2D objects are at  $z = 0$  in the game world. The third line of code converts the mouse position from screen coordinates to world coordinates.

The fourth line of code in the if body instantiates our Pickup prefab, but you may have noticed that we used a different form of the `Instantiate` method. The one we're using here is the generic form of the method, so we provide `<GameObject>` as part of the method call rather than including `as GameObject` at the end as we did previously. They both yield the same result – a new instance of the prefab in the game world – so we figured we'd show you both versions. Just pick whichever one you prefer!

The fifth line of code in the if body sets the position of our new Pickup game object to the world location of where the right mouse button was pressed, and the sixth line of code adds the new Pickup game object to the list of pickups that the script maintains.

If you run the game now, you should be able to place pickups in the game by right-clicking the mouse. Great, we're past Step 1 in our Test Case; let's get Step 2 working.

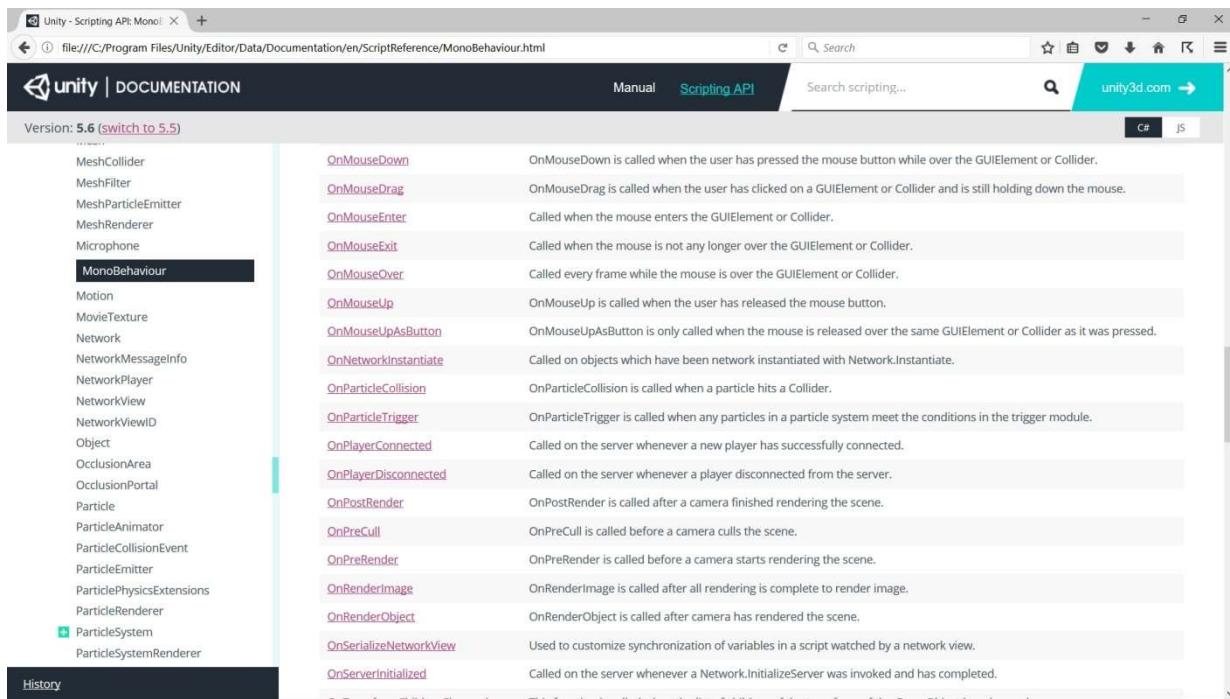
Now we need to make the `TeddyBear` game object start moving toward the oldest Pickup in the game when the player left-clicks the teddy bear. Recall that the `TedTheCollector` class exposes the `TargetPickup` property to support this, so let's implement that property now. You should realize that we're deliberately making a mistake in the property, which we'll discover later in our coding; see if you can figure out what the mistake is.

```
/// <summary>
/// Gets the next target pickup for the teddy bear to collect
/// </summary>
/// <value>target pickup</value>
public GameObject TargetPickup
{
    get { return pickups[0]; }
}
```

The `get` accessor simply returns the first `GameObject` (a `Pickup` game object) in the list of pickups that the class maintains; remember, the first item in the list is at index 0.

Because the `TedTheCollector` script recognizes right clicks in its `Update` method, our initial thought was that the `TeddyBear` script would recognize left clicks in its `Update` method. This is a little more complicated than right clicks, though, because we don't want to recognize any left click, we only want to recognize left clicks that are actually on the `TeddyBear` game object.

It turns out that instead of using the `Update` method for this, we should use the `OnMouseDown` method instead. That's because the `OnMouseDown` method is called when the user has pressed the mouse button over a collider; the documentation means the left mouse button when it says "the mouse button", so this method will get called just when we need it to be. Go ahead and delete the `TeddyBear` `Update` method now since we won't need it after all. As reminder, it's always a good idea to check the `MonoBehaviour` Messages section (excerpt below) to see if there's a method that we can use for what we need.



**Figure 9.7. MonoBehaviour Messages (Methods)**

Because our TeddyBear game object doesn't have a collider component yet (which we need for `OnMouseDown` to be called), we'll add one now. Double-click the TeddyBear game object in the Hierarchy window of the Unity editor; this zooms in on the game object in the Scene view. Click the Add Component button in the Inspector and select Physics 2D > Box Collider 2D. As you can see in the Scene view, the Box Collider 2D is too large because the sprite image has some border transparency. Click the button to the left of Edit Collider in the Box Collider 2D component, drag the edges of the collider in the Scene view to fit the TeddyBear game object more tightly, then click the button to the left of Edit Collider in the Box Collider 2D component again. Click the small box next to Is Trigger in the Box Collider 2D component.

That last step sets the collider to be a trigger rather than a collider from a physics perspective. This will be important when the TeddyBear collides with Pickups in the game because we don't actually want the TeddyBear and Pickup objects to respond to each other physically (don't be weird!), but we do want to know when they collide.

While we're at it, we might as well add a Rigidbody 2D component to the TeddyBear as well. We're going to just use the Unity physics engine to move the TeddyBear, and we'll need a Rigidbody 2D component attached to it to do that. Click the Add Component button in the Inspector and select Physics 2D > Rigidbody 2D. Click the Apply button to the far right of Prefab near the top of the Inspector to apply the changes to the TeddyBear prefab.

If you run the game now, the TeddyBear game object falls off the bottom of the screen. Select Edit > Project Settings > Physics 2D from the menu bar at the top of the editor and set the Y value for Gravity in the Inspector to 0. Run the game again to verify that you turned gravity off correctly.

The first thing we'll need to do in the `OnMouseDown` method is access the `TedTheCollector` `TargetPickup` property – and we immediately have another problem! The `TeddyBear` class, which is

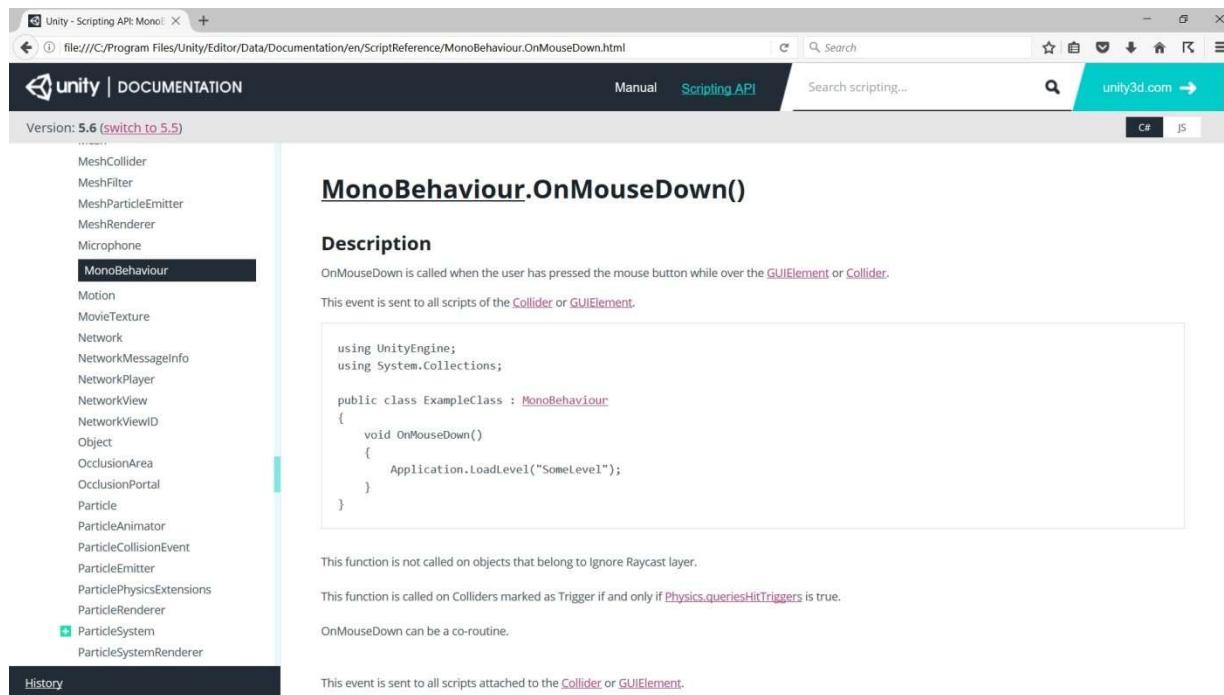
attached to the `TeddyBear` game object, doesn't have a reference to the `TedTheCollector` class, which is attached to the Main Camera.

We've actually already learned all the pieces we need to solve this problem, we just have to put them together in a new way. We know how to get access to the Main Camera using `Camera.main` and we know how to get a component attached to a game object using the `GetComponent` generic method. That means we can use the following to get the reference we need:

```
Camera.main.GetComponent<TedTheCollector>()
```

Because we need to use this reference every time we have a left click on the `TeddyBear` game object, for efficiency we add a `tedTheCollector` field to the `TeddyBear` class and get the reference in the `Start` method; we do the same thing for the reference to the Rigidbody 2D component as well.

Now we can get working on the `OnMouseDown` method. Because it's not provided in the template script when we create a new C# script in Unity, we need to add the entire method (not just the body of the method like we've been doing for `Start` or `Update`). How do we know what the method should like? The documentation again, of course!



**Figure 9.8. OnMouseDown Documentation**

The easiest approach to use to make sure we get the method correct is to simply copy the entire method from the example in the documentation, add a comment above the method, delete all the code in the method body (the code between the `{` and the `}`), and add the code we need in the method body. Here's what we end up with when we do that:

```
/// <summary>
/// OnMouseDown is called when the user has pressed the mouse button
/// over the collider
/// </summary>
```

```

void OnMouseDown()
{
    // ignore mouse clicks if already collecting
    if (!collecting)
    {
        // calculate direction to target pickup and start moving toward it
        targetPickup = tedTheCollector.TargetPickup;
        Vector2 direction = new Vector2(
            targetPickup.transform.position.x - transform.position.x,
            targetPickup.transform.position.y - transform.position.y);
        direction.Normalize();
        rigidbody2D.AddForce(direction * ImpulseForceMagnitude,
            ForceMode2D.Impulse);

        collecting = true;
    }
}

```

The if statement makes sure we only respond to left mouse clicks on the `TeddyBear` game object if that object isn't already collecting Pickups. The first line of code in the if body retrieves the target pickup the teddy bear should go collect. We're saving that target pickup in a new field in the `TeddyBear` class for reasons that will become clear soon.

The second line of code in the if body creates a `Vector2` that points from the `TeddyBear` game object to the Pickup game object it should go collect. The magnitude of that `Vector2` is dependent on the distance between the `TeddyBear` and the Pickup, though, so we want to normalize that vector so it points in the same direction with a magnitude of 1; that's what the call to the `Normalize` method in the third line of code does.

We added an `ImpulseForceMagnitude` constant at the top of the `TeddyBear` class, so the fourth line of code in the if body adds an impulse force to the `TeddyBear`'s `rigidbody` toward the pickup with a magnitude of `ImpulseForceMagnitude`. By normalizing our vector and multiplying by our constant, we're ensuring that the `TeddyBear` object always moves at the same speed toward its target Pickups independent of the distance between the teddy bear and the target Pickup it's going to collect.

We also set the `collecting` flag to `true` because the `Teddy Bear` is now collecting.

Run Steps 1 and 2 of the test case and you'll see that the `TeddyBear` moves toward the Pickup, but then keeps on going after it reaches it. That's because we haven't added the code we need to actually detect when the `TeddyBear` reaches its target Pickup.

What we really need to do here is detect a collision between the `TeddyBear` and the target Pickup. To do that in Unity, both game objects need to have a `Collider2D` component attached to them. At this point, only the `TeddyBear` has a collider, so now we need to add one to the Pickup prefab.

Drag a Pickup prefab from the prefabs folder in the Project window onto the Scene view (don't put it on top of the `TeddyBear`). Click the Add Component button in the Inspector and select Physics 2D > Circle Collider 2D. As you can see in the Scene view, the Circle Collider 2D is too large because the sprite image has some border transparency. Click the button to the left of Edit Collider in the Circle Collider 2D component, drag one of the boxes on the collider in the Scene view to fit the Pickup game object more tightly, then click the button to the left of Edit Collider in the Circle Collider 2D component again.

Click the Apply button to the far right of Prefab near the top of the Inspector to apply the changes to the Pickup prefab. Delete the Pickup game object from the scene.

Now we need to have the `TeddyBear` script take the appropriate action when the collision we're looking for occurs. Remember how we added the `OnMouseDown` method to respond to left clicks on the `TeddyBear`? Well, we can also add an `OnTriggerEnter2D` method that automatically gets called when the collider for the `TeddyBear` (remember, we made it a trigger) collides with another collider (a collider for a Pickup game object). We need to add the entire method like we did for `OnMouseDown`; see below.

```
/// <summary>
/// Called when another object enters a trigger collider
/// attached to this object
/// </summary>
void OnTriggerEnter2D(Collider2D other)
{
    // only respond if the collision is with the target pickup
    if (other.gameObject == targetPickup)
    {
        // remove collected pickup from game and go to the next one
        tedTheCollector.RemovePickup(targetPickup);
        rigidbody2D.velocity = Vector2.zero;
        GoToNextPickup();
    }
}
```

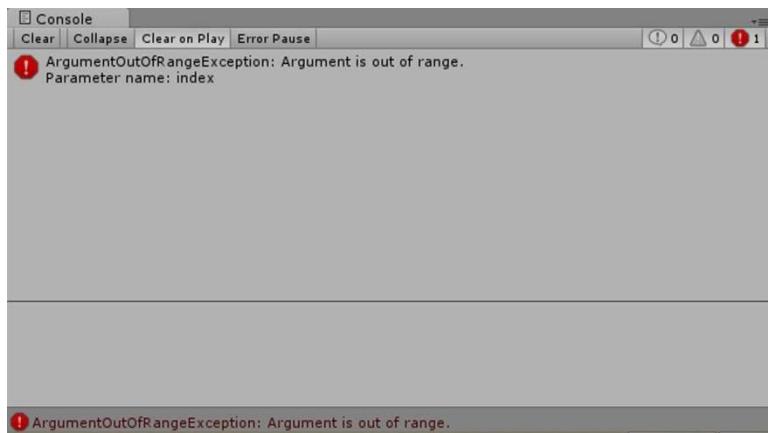
Remember, we're supposed to ignore any collisions we have with Pickups that aren't currently our target Pickup; that's what the if statement checks for. The first line of code in the if body removes the target pickup from the game (once we implement the body of the `TedTheCollector RemovePickup` method, which we'll do soon). The second line of code stops the `TeddyBear` by setting its velocity to 0. We do this so we get straight lines from one Pickup to the next as the `TeddyBear` collects them.

For the third line of code, we realized that the code we have in the if body of the `OnMouseDown` method is identical to the code we need here. Rather than copying and pasting that code, we instead wrote a new method called `GoToNextPickup` and had both the `OnMouseDown` method and the `OnTriggerEnter2D` method call that new method. That way, we only have the required code in a single place.

Implementing the body of the `TedTheCollector RemovePickup` method is easy, because the `List` class exposes a `Remove` method to remove a specific element from the list. Here are the lines of code we put in the body of the method to remove the Pickup from both the list and the game:

```
pickups.Remove(pickup);
Destroy(pickup);
```

Run Steps 1 and 2 of the test case. The good news is that the `TeddyBear` goes to the target Pickup and stops and the Pickup is removed from the game. The bad news is that we get the error shown in the Console window in Figure 9.9.



**Figure 9.9. Console Window Error**

Oh dear (replace dear with your favorite expletive as necessary). The Argument is out of range error (we'll discuss exceptions later in the book) occurs when we try to access an element in an array or collection with an index larger or smaller than the valid indexes for that array or collection. The list of pickups is the only collection we have in our game, and we only access an element in that list with a specific index in the `TedTheCollector` TargetPickup property, so let's use the debugger to see what's going on.

Open the `TedTheCollector` script in Visual Studio and put a breakpoint next to the get accessor in the `TargetPickup` property by left-clicking in the gray column to the left of that line of code. If you do that properly, a red circle will appear in the column at that location. Select Run > Attach to Process ... from the menu bar at the top and double-click the Unity Editor process in the resulting dialog. Run the game in the Unity editor, wait until the play/pause/advance buttons turn blue, then right-click to place a pickup and left-click the teddy bear to start it collecting.

When the game pauses at the breakpoint (Windows gives the Visual Studio window focus at that point), hover the mouse over `pickups` in the get accessor. As you can see, the `pickups` list has 1 element at this point (`Count == 1`), so we should be fine for now. Press F5 to continue debugging and wait in Visual Studio until the game stops at the breakpoint again. At this point, the teddy bear has collected the pickup and is asking for the next target pickup to go to. If you hover the mouse over `pickups` again you'll see that the `pickups` list is now empty.

Accessing the first element (at index 0) of an empty list has got to be trouble, because there is no first element! Press F10 to step over the breakpoint (stepping over simply executes that line of code) and go back to the Unity editor to see the error message. Stop running the game in the editor, stop debugging in Visual Studio by selecting Run > Stop on the top menu bar, and remove the breakpoint by clicking the red dot to the left of the get accessor in Visual Studio.

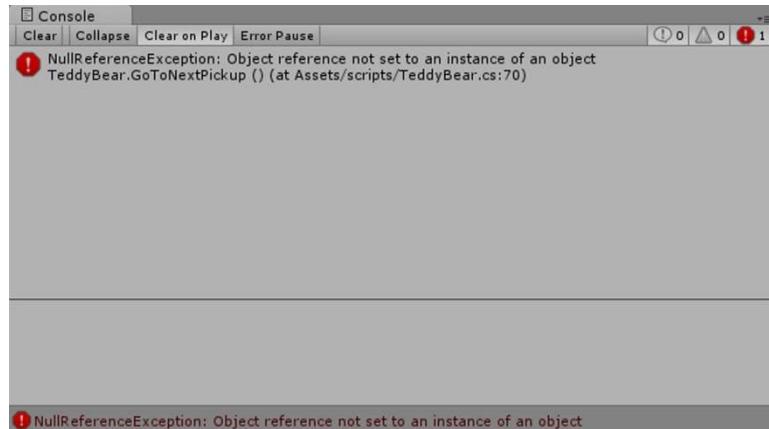
Remember we told you we were deliberately making a mistake when we implemented the `TargetPickup` property; now we know what the mistake is! We need to make sure that we only return the first pickup in the list if the list isn't empty. That's easy to do with an if statement, but what should the property return if the list is empty? One very common technique is to return `null` from properties and methods that usually return an object (in this case, a `GameObject`) but can't return a valid object given the current state of the game (in this case, there are no more pickups to collect). That's what we'll do here, so we change the body of the get accessor to:

```

if (pickups.Count > 0)
{
    return pickups[0];
}
else
{
    return null;
}

```

Run Steps 1 and 2 from the test case again; now we get the error shown in Figure 9.10.



**Figure 9.10. Null Reference Error**

Amazing! We seem to just be going from one problem to another, but trust us, we're actually moving forward with the game. This error gives us a little more detail than the previous one, because it tells us the problem is in the `TeddyBear` `GoToNextPickup` method.

We can figure out the problem here without even using the debugger. The line of code that calculates the direction vector from the teddy bear to the target pickup accesses the `transform` field of `TeddyBear` `targetPickup` field, which holds the result of accessing the `TedTheCollector` `TargetPickup` property. If the `TargetPickup` property returns `null`, then the `targetPickup` field is `null`, and we can't access the `transform` field of an object that doesn't exist! To solve this problem, we change the body of the `GoToNextPickup` method to:

```

// calculate direction to target pickup and start moving toward it
targetPickup = tedTheCollector.TargetPickup;
if (targetPickup != null)
{
    Vector2 direction = new Vector2(
        targetPickup.transform.position.x - transform.position.x,
        targetPickup.transform.position.y - transform.position.y);
    direction.Normalize();
    rigidbody2D.AddForce(direction * ImpulseForceMagnitude,
        ForceMode2D.Impulse);
    collecting = true;
}
else
{

```

```

    collecting = false;
}

```

If the `TargetPickup` property returns an actual `GameObject`, the code works just as before. If it returns `null`, the teddy bear doesn't do anything; we do need to set the `collecting` flag to `false`, though, because the Teddy Bear is no longer collecting at this point.

Finally, Steps 1 and 2 of the test case work the way they're supposed to! In fact, the entire test case should work at this point, so let's Test the Code.

### *Test the Code*

Our test case now passes, so strictly speaking, we're done. Unfortunately, there's actually still a bug in the code that we should fix. Try placing two pickups close to each other, with the first pickup further away than the second one, then click the teddy bear and watch what happens. Why does the teddy bear pick up the first pickup but not the second one?

Before we fix the bug, we now have another test case we should include in our test plan, so we'll add that now. By the way, this problem shows how hard it is to develop a set of test cases that will find every possible bug in our code; in fact, except for the simplest programs it's impossible!

### *Write Test Cases (again)*

## Test Case 2

### **Checking Close Pickup Behavior**

Step 1. Input: Right Click

Expected Result: Pickup placed at click location

Step 2. Input: Right Click close to the first pickup on a line between the teddy bear and the first pickup

Expected Result: Pickup placed at click location

Step 3. Input: Left Click on Teddy Bear

Expected Result: Teddy Bear collects oldest Pickup, then collects other pickup, then stops

### *Write the Code (again)*

Let's use the debugger again to try to figure out what's going on. We know that the teddy bear actually picks up each pickup in the `TeddyBear` `OnTriggerEnter2D` method, so that's a good place for us to set a breakpoint.

Open up the `TeddyBear` script in Visual Studio and set a breakpoint on the first line of the `OnTriggerEnter2D` method. Attach Visual Studio to the Unity Editor process and start the game in the Unity editor.

The first time we hit the breakpoint is when the teddy bear collides with the non-target pickup; press F5 to continue. The second time we hit the breakpoint is when the teddy bear collides with the target pickup, so we go into the if statement to remove that pickup and get the new target. Press F10 to step over each line of code until the yellow line is on the call to the `GoToNextPickup` method, then press F11 to step into that method. Press F10 one more time and hover the mouse over `targetPickup`; as you can see, we have a non-`null` target pickup. Press F5 to continue to the breakpoint in the `OnTriggerEnter2D` method.

Hmmm ... we never hit the breakpoint the third time, when we should be picking up the second pickup. Stop the game in the Unity editor. If we go back to the `MonoBehaviour OnTriggerEnter2D` method, we see the part of the description that says "... when another object enters a trigger collider ..." That seemed just right to us when we decided to use the `OnTriggerEnter2D` method, but what if our next target pickup is actually already in the trigger collider for the teddy bear because the pickups were so close to each other? Our next target pickup never enters the trigger collider because it's already there!

It looks like the `OnTriggerEnter2D` method isn't the one we should be using after all. Luckily, if we look a little further down in the documentation we find the `OnTriggerStay2D` method, which is called "... each frame where another object is within a trigger collider..." This should work for us, because even if our next target pickup is actually already in the trigger collider for the teddy bear when we set it as the target, the `OnTriggerStay2D` method will get called. Change the `TeddyBear OnTriggerEnter2D` method to an `OnTriggerStay2D` method instead and remove the breakpoint.

So why didn't we just tell you the correct method to use in the first place? Because we're walking you through exactly how we solved this problem. Programmers make mistakes, and figuring out what those mistakes are and how to solve them is an important skill for you to have. We spend a lot of time in this book on the process of programming in addition to the mechanics of programming because the process is just as, if not more, important.

### *Test the Code (again)*

As expected, both test cases complete successfully, so technically we're done with our problem solution. We're not quite happy, though; read on.

### *Understand the Problem (again)*

It actually looks strange if the `TeddyBear` ignores a `Pickup` it collides with on its way to its target `Pickup` because it passes under the ignored `Pickup`; it feels like it would look better for the `TeddyBear` to pass over the ignored `Pickup` instead. Let's add one more explicit requirement to the Problem Description:

- The `TeddyBear` passes over `Pickups` it collides with that aren't the `Pickup` the `Teddy` has currently "targeted for collection"

### *Write Test Cases (for the third time)*

We can make a slight revision to Expected Result for Step 7 of our first test case to address the new requirement:

#### **Test Case 1**

##### **Checking Game Behavior**

Step 1. Input: Right Click

Expected Result: Pickup placed at click location

Step 2. Input: Left Click on Teddy Bear

Expected Result: Teddy Bear collects Pickup and stops

Step 3. Input: Left Click on Teddy Bear

Expected Result: No response (there's no pickup to collect)

Step 4. Input: Right Click

Expected Result: Pickup placed at click location

Step 5. Input: Right Click

Expected Result: Pickup placed at click location

Step 6. Input: Left Click on Teddy Bear

Expected Result: Teddy Bear collects oldest Pickup then moves toward next pickup

Step 7. Input: Right Click between Teddy Bear and Pickup while Teddy Bear is moving toward Pickup

Expected Result: Pickup placed at click location, Teddy Bear collects oldest Pickup, passing over final Pickup, then collects final Pickup, then stops

#### *Write the Code (for the third time)*

Our solution no longer passes the first test case because we changed the requirements after we were done! We don't have to change anything in our design to fix this, though, so we can move directly to making the changes here. We don't actually have to change our scripts at all, but we do have to make some changes in the Unity editor.

To figure out what changes we need to make, we need to understand two more Unity features: *Sorting Layers* and *Order in Layer*. In Unity, we can place objects in different sorting layers to control the order in which the sprites are rendered in the scene. The template Unity 2D game project only contains a single sorting layer called Default. If you select the TeddyBear and Pickup prefabs and look at the Sorting Layer value in their Sprite Renderer components, you'll see that they're both set to Default; that means both of them will be drawn when the sprites in the Default sorting layer are drawn.

One solution to our problem would be to add a new Sorting Layer to the project by selecting Edit > Project Settings > Tags & Layers. Expanding the Sorting Layers section in the Inspector shows all the sorting layers in the game. To add a new layer, we'd click the + at the bottom right of the list of sorting layers. The sorting layers are rendered back to front from the top of the list, so if we name our new sorting layer TeddyBear all the sprites in the Default sorting layer would be rendered first, then all the sprites in the TeddyBear sorting layer would be rendered. If we went back to our TeddyBear prefab and used the Sorting Layer dropdown in the Sprite Renderer component to assign the TeddyBear sorting layer, our TeddyBear game object would be drawn in front of everything else (including Pickup game objects, which are still in the Default sorting layer). That's one way to solve our problem.

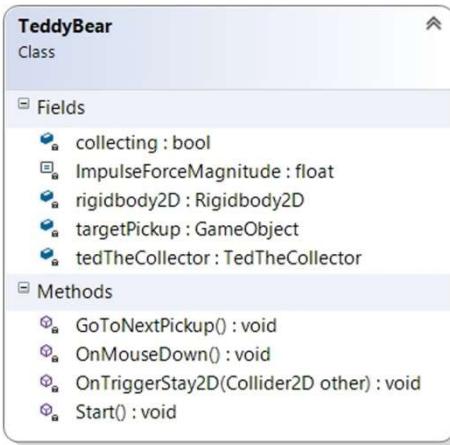
Just so you can see another reasonable solution, let's use Order in Layer instead. Order in Layer is used to control the order in which sprites are rendered within a single Sorting Layer. If you select the TeddyBear and Pickup prefabs and look at the Order in Layer value in their Sprite Renderer components, you'll see that they're both set to 0 (make sure they both have Sorting Layer set to Default). Because sprites with a lower Order in Layer number are rendered behind sprites with a higher Order in Layer number, we can simply set the Order in Layer value for the TeddyBear prefab to 1 to solve our problem.

#### *Test the Code (for the third time)*

Both test cases pass successfully! Whew.

Everything now works fine in our solution, but remember that we made quite a few changes to our design for the `TeddyBear` class as we uncovered details during the Write the Code step. In Visual Studio, changing the code in a class automatically changes the UML diagram for that class if you've

added a UML diagram for that class to your Visual Studio solution. We've provided the revised UML for the class below so you can see the final design.



**Figure 9.11. TeddyBear Final UML**

## 9.8. Common Mistakes

### *Indexing Past the End of the Array*

It should come as no surprise that, if we try to index outside the array (i.e., access an array element that doesn't exist), our program will blow up. For example, if our array indices go from 0 to 9 and we try to look at element 10, C# will throw an `ArrayIndexOutOfBoundsException`. Indexing outside the array occurs most commonly when we're actually calculating the indices of the elements we're referencing.

### *Indexing Outside the Contents of a Collection*

Similarly, if we try to index outside the contents of a collection (i.e., access a collection element that doesn't exist), our program will blow up. We saw this when we tried to return the first pickup from an empty list of pickups. In these cases, C# will throw an `ArgumentOutOfRangeException`.

### *Trying to Read or Print the Entire Array at Once*

Because we can think of array variables as similar in many ways to the other variables we've been using, you may be tempted to try to read in the entire array or print out the entire array all at once. You can't do that! Remember, arrays are actually objects, and you have to read or print each element of the array separately.

### *Forgetting to Create Objects in Array or Collection*

It's not very common to forget to create a new array or collection when we declare our variable, but for an array or collection of objects you may forget to actually create the objects within the array or collection. If you forget, C# will throw a `NullReferenceException` when you try to access properties or methods of any of those array or collection elements. Remember, with arrays or collections of objects we need to create both the array or collection and the objects in the array or collection.

# Chapter 10. Iteration: For and Foreach Loops

So far, we've learned how to use C# to solve problems that require execution of a set of sequential steps and/or one or more selections. The set of problems we can solve is still somewhat limited, though, and this chapter presents some of the C# constructs used to implement the last of the three basic control structures – iteration (also commonly called looping). We'll cover one form of iteration here and another in the following chapter. Let's take a closer look.

## 10.1. Iteration Control Structure

The ability to accomplish certain steps in our code multiple times (such as printing out the cards in a hand that changes as the program runs) is the last control structure we need for our problem solutions. In other words, we'd like to iterate (repeat) those steps in the algorithm. That's where the third and final control structure comes in – the iteration control structure. Iteration control structures are often called *loops*, because we can loop back to repeat steps in the algorithm.

Let's look at an example:

### *Example 10.1. Printing All the Cards in a Hand*

Problem Description: Write an algorithm that will print out all the cards in a hand

Algorithm: Here's one solution:

```
For each card in the hand
    Print card information
```

We use indentation to show what happens inside the loop, just as we used indentation to show what happened inside our selection algorithms.

Basically, our solution looks at each card in the hand and prints the information about that card. The step inside the loop body – printing the info about a single card – is executed repeatedly. We already printed out the cards in a hand in the previous chapter, but remember what our code looked like for five cards:

```
Console.WriteLine(hand[0].Rank + " of " + hand[0].Suit);
Console.WriteLine(hand[1].Rank + " of " + hand[1].Suit);
Console.WriteLine(hand[2].Rank + " of " + hand[2].Suit);
Console.WriteLine(hand[3].Rank + " of " + hand[3].Suit);
Console.WriteLine(hand[4].Rank + " of " + hand[4].Suit);
```

We actually had to know how many cards were in the hand and print out exactly that many cards. This is a real problem, especially if the number of cards in the hand changes over time as it typically does. Iteration lets us overcome this problem by letting us repeat the printing card information step precisely the number of times we need to when we run the program, even if the number of times changes while we're running the program.

## 10.2. For Loops

In Example 10.1, we knew how many times the loop should execute. One of the common ways to implement this kind of loop in C# is with a *for loop*. The for loop syntax is described here:

### SYNTAX: For Loop

```
for (initializer; condition; iterator)
{
    loop body
}
```

*initializer*, code that initializes the loop control variable (a variable whose value controls execution of the loop)

*condition*, a Boolean expression that's evaluated each time through the loop to see if we'll keep looping

*iterator*, an expression that says how to modify the loop control variable at the end of each time through the loop (we often increment by one)

*loop body*, the code that's executed each time through the loop

We do want to make a comment about the for loop before we try one out. C# will actually let us use a variety of data types for the upper and lower bounds of our for loops. For the problems in this book, though, we'll usually be using integer bounds.

All right, ready to try one? Let's actually implement the code for the algorithm in Example 10.1. We'll assume that a `hand` variable has already been declared and instantiated as a `List<Card>` object and already has some cards added to it (though the loop below works with 0 cards also).

First we'll add the start and end of the for loop:

```
// print cards in hand
for (int i = 0; i < hand.Count; i++)
{
}
```

Our initializer is the chunk of code that says `int i = 0`; `i` is our loop control variable, and it's initialized to 0. The loop will stop when the condition evaluates to `false` (in other words, when `i` is no longer less than `hand.Count`). Notice that we access the `Count` property of our `hand` list to determine how many times to loop; that way, the loop works no matter how many cards are in the hand. So why don't we use `i <= hand.Count` instead; if we use `<`, won't we skip the last card in the hand? No, because the indexes of the list elements are zero-based, not one-based. The maximum index of an element in the `hand` list is `hand.Count - 1`, so using `i < hand.Count` rather than `i <= hand.Count` is the right choice.

When we use `i++` as our iterator, we're saying to add 1 to `i` at the end of each time through the loop. Now we modify the loop code to add the loop body, yielding

```
// print cards in hand
for (int i = 0; i < hand.Count; i++)
{
    Console.WriteLine(hand[i].Rank + " of " + hand[i].Suit);
}
```

The output when we run this code fragment (using our royal flush hand) will look like this:

```
Ten of Spades
Jack of Spade
Queen of Spades
King of Spades
Ace of Spades
```

The loop executes 5 times because there are 5 cards in the hand, and it prints out the information about each card based on the value of `i` in the loop body on each iteration.

What? Think of it this way. The first time we enter the loop body, `i` is equal to 0, so we print out the rank and suit for `hand[0]`. The second time we enter the loop body, `i` is equal to 1, so we print out the rank and suit for `hand[1]`. It keeps working that way until the loop stops. This is really convenient, because instead of hard-coding literals for our indexes like we had to do in the previous chapter, here we can just use the loop control variable as our index. You'll see and write for loops like the one above many, many times as you program games and other software applications, so make sure you understand how they work.

Now you may have already realized this from the above example, but you should know that we don't have to know how many times the for loop will execute when we write the program – we just have to make sure we know how many times the for loop will execute before the program reaches that point in its execution. Consider Example 10.2.

#### *Example 10.2. Printing Squares of the Integers from 1 to n*

Problem Description: Write a program that will print the squares of the integers from 1 to `n`, where `n` is provided by the user.

Algorithm: Here's one possible algorithm:

```
Prompt for and get n
For each of the integers from 1 to n
    Print the square of that integer
```

And when we implement the algorithm in C#, we get

```
// get number of squares to print
Console.Write("Enter the number of squares to print: ");
int n = int.Parse(Console.ReadLine());

// print the squares
for (int i = 1; i <= n; i++)
{
    Console.WriteLine("The square of {0} is {1}", i, i * i);
```

Although we do things slightly differently in this for loop – we start `i` at 1 rather than 0 and we use `<=` in our condition rather than `<` – the ideas are exactly the same and the loop works just as you'd expect.

So that's it. When we know how many times to loop, either when we write the program or simply before we get to the loop during program execution, we can easily use a for loop. In case you're wondering how to test our code when it contains a for loop, we test that part of the code just like we tested the sequence control structure; we just run it to see if it works.

### 10.3. Foreach Loops

There's actually another form of loop provided in C# that we can use when we want to loop a set number of times – the foreach loop. The foreach loop can be used to iterate over the elements in an array or a collection. That makes foreach loops a little more limited than the for loop – we couldn't use it to solve our "print the squares" problem, for example – but you'll find they're incredibly handy. Here's the syntax:

#### SYNTAX: Foreach Loop

```
foreach (dataType variableName in arrayName or collectionName)
{
    loop body
}
```

`dataType`, the data type of the elements in the array or collection

`variableName`, the name of the variable that will hold an element in the loop body

`arrayName or collectionName`, the name of the array or collection to iterate over

`loop body`, the code that's executed each time through the loop

Let's print the cards in our hand again, this time using a foreach loop:

```
// print cards in hand
foreach (Card card in hand)
{
    Console.WriteLine(card.Rank + " of " + card.Suit);
}
```

Because each element in our list is a `Card` object, we use `Card` as the data type for the `card` variable. The `card` variable will be assigned to each element of the list as we execute the loop body. In other words, the first time through the loop, `card` will be set to `hand[0]`, so we print out the rank and suit for `hand[0]`. The second time through the loop, `card` will be set to `hand[1]`, so we print out the rank and suit for `hand[1]`. It works this way until we've iterated over all the elements in the `hand` list. Finally, we needed to provide an array or collection over which we want to iterate; because we wanted to print all the cards in the hand, we provided `hand` for the collection.

## 10.4. Choosing Between For and Foreach Loops

Given that we have the option of using a for loop or a foreach loop when we want to iterate a set number of times, how do you decide which one to use? In general, C# programmers tend to use foreach loops whenever possible, though there isn't a set C# standard to do so. There are a number of cases where the for loop is the correct choice, however.

One example is solutions in which we need to know the index of the element we're currently processing in the loop body. In that case, we need to use a for loop so we can access the loop control variable inside the loop body. For example, if we were trying to collect the locations of all the cards in the hand with a specific characteristic (all the Jacks, say), a for loop would be the way to go.

A second example is when we want to execute the loop body a certain number of times but we're not processing the elements of an array or collection. We've already seen an example of this in Example 10.2, where we printed a certain number of integers and their squares.

A third example is when we actually want to change the contents of a collection we're iterating over. We're allowed to do this with for loops – we can remove each Jack as we find it, for example – but we're not allowed to change the contents of a collection we're iterating over using a foreach loop.

## 10.5. Nested Loops

So, you might be asking "If we can put any kind of executable statements in the loop body, can we put another loop in the loop body?" Of course we can (we probably wouldn't have brought it up if you couldn't)! When we do that, we have what's called *nested loops*, because one loop is nested inside the other. We can, of course, nest as many loops inside each other as we want, though we don't usually have too many levels of nesting. Let's look at an example.

### *Example 10.3. Generating All the Cards in a Deck*

Problem Description: Write a code fragment that will generate all the cards in a deck.

We know we have 4 different suits, with 13 different ranks in each suit, so this seems like the kind of problem where nested for or foreach loops would be just the thing to use. Let's initialize the deck of cards first using:

```
List<Card> deck = new List<Card>(52);
```

We're actually using a different overload of the `List` constructor that lets us specify the initial capacity of the list when we create it. Why are we doing that?

Remember when we discussed how you'd have to grow an array if you needed to add more elements than you initially allocated space for? It turns out that the `List` class actually uses an array as the underlying structure to hold all the list elements. If we carefully read the documentation of the `List Add` method, we find that adding elements to the list is usually very fast. If, however, the capacity of the list needs to be increased to fit the new element, the `Add` method takes much longer to complete. Although the `Add` method will automatically grow the capacity of the list as needed, we can avoid the extra time it would take to do that by simply creating an initial list that has a capacity of 52 cards. That way, the

capacity of the list doesn't have to be increased as we add the cards to it. By the way, we can always find out the current capacity of a list by accessing its `Capacity` property.

Now that we have a deck to hold all the cards, let's develop the code to fill the deck with cards.

Algorithm: Here's an algorithm using nested foreach loops:

```
For each possible suit
  For each possible rank
    Add a card with the rank and the suit to the deck
```

Let's write the code, then talk about how it works:

```
// fill the deck with cards
foreach (Suit suit in Enum.GetValues(typeof(Suit)))
{
  foreach (Rank rank in Enum.GetValues(typeof(Rank)))
  {
    deck.Add(new Card(rank, suit));
  }
}
```

Okay, let's look at the outer foreach loop first. This loop is set up to loop through each of the possible suits in the `Suit` enumeration, but we're obviously using some C# features we haven't used before. At least the first part is familiar; each element we look at will be a `Suit`, and the variable that will hold the current element in the loop body is called `suit`. It's the specification of the array we want to iterate over that looks different.

Recall that the `Int32` structure is used to provide `int`-specific methods; in much the same way, the `Enum` class is used to provide `enum`-specific methods (remember, our `Suit` enumeration is declared as an `enum`). The `GetValues` method in the `Enum` class returns an array of the values defined by the enumeration provided as the argument to the method.

The last piece of the puzzle, then, is understanding why we have to pass `typeof(Suit)` as the argument to the `GetValues` method rather than `Suit`. The issue is that the argument to the method actually needs to be a `Type` object that represents the type declaration of a particular type rather than the type itself. Luckily, the `typeof` operation can be used to retrieve the `Type` object for a specific type (like `Suit`).

We know that feels pretty complicated given where you are in your programming knowledge at this point. For now, just knowing the syntax you need to use to get an array of the values in an enumeration is enough; you can develop a deeper understanding of `typeof` and `Type` as you do more advanced programming later.

The inner foreach loop works in a similar way to iterate over the possible ranks for the cards. Now that we understand the details about how each foreach loop is set up, we can move on to understanding how the code will actually work when it executes.

When we reach the outer loop, the `suit` variable is set to `Suit.Clubs` since `Clubs` is the first value we defined in the `Suit` enumeration. We then move to the loop body of the outer loop, which contains the inner loop. In the inner loop, the `rank` variable is set to `Rank.Ace`. In the loop body for the inner loop,

we create a new Ace of Clubs card and add it to the deck. We then execute the next iteration of the inner loop, where we create a new Two of Clubs card and add it to the deck. We keep iterating through the inner loop until we've covered all the possible card ranks, creating a new King of Clubs card and adding it to the deck on the last iteration.

Once the inner loop is done, we execute the next iteration of the outer loop, where the `suit` variable is set to `Suit.Diamonds`, the second value we defined in the `Suit` enumeration. We then generate all the Diamond cards in the inner loop in the same way we generated the Clubs cards above, going from Ace to King. Once the outer loop has executed for all the suits, the code is done and our deck contains all 52 cards.

Pretty cool, huh? The only thing you might find a little confusing is that the inner loop "starts over" each time we get to it on each iteration of the outer loop. It's just like when we get to a loop that's not nested, though. The program just starts the loop when it reaches it; it's just that we reach the inner loop 4 times in the above code instead of only once.

## 10.6. Arrays and Loops - "Walking the Array"

Although there are certainly times when we want to use specific elements of an array as we discussed in the previous chapter, there are many other times that we want to look at all the elements in the array – in some sense, "walking the array" from top to bottom. We've already shown how to use both for loops and foreach loops to do this on collections, and it works almost exactly the same way for arrays! Let's revisit Example 10.1, but this time we'll use an array for the hand of cards instead of a list.

### *Example 10.4. Printing All the Cards in a Hand*

Problem Description: Write an algorithm that will print out all the cards in a hand

Algorithm: Our algorithm stays exactly as it was before:

```
For each card in the hand
    Print card information
```

and our code only changes in one place (we use the `Length` property of the array rather than the `Count` property of the `List`):

```
// print cards in hand
for (int i = 0; i < hand.Length; i++)
{
    Console.WriteLine(hand[i].Rank + " of " + hand[i].Suit);
}
```

There's obviously not much difference here at all. Rather than exposing a `Count` property like the `List` class does to tell how many elements are in the list, arrays expose a `Length` property to tell the number of elements there are in the array. By simply using the appropriate property, we easily get the same behavior we got with a `List` earlier. We actually allocated the array to hold exactly 5 elements when we created the array object, so we could have used 5 instead of `hand.Length` above, but then if we changed the size of the array, we'd also have to change the bound in the for loop. When you're using arrays, you should get into the habit of using the `Length` property since that's a more robust approach.

You should realize, though, that the above code will only work if we have exactly five cards in the `hand` array. If there are some “empty slots” in the array that don’t hold cards (so those elements are `null` instead), the `Console.WriteLine` will fail when we try to print the rank and suit for one of those empty slots. That’s another reason we prefer using the `List` class when the size of our collection can change over time.

Things also get a little trickier when we deal with 2D arrays, so let's look at another example.

#### *Example 10.5. Adding Up the Numbers in a Grid*

Problem Description: Add up all the numbers in a 3 by 4 grid of integers.

First, let's come up with an algorithm:

```
Initialize sum to 0
Loop through the rows in the array
    Loop through the columns in the array
        Add current number to sum
```

Before converting our algorithm into code, we'll declare the 2D array that holds the numbers:

```
int[,] numbers = new int[3,4];
```

You should also assume we've filled up that array with values, probably using nested loops.

Before we convert our algorithm into code, we need to figure out how to get the size of each dimension in the array. We can't use the `Length` property, which tells us the total size of the array (12 elements in this case), so we need something else. Luckily, array objects provide a `GetLength` method that will give us the number of elements in a particular zero-based dimension. In other words, calling the `GetLength` method with an argument of 0 will tell us how many rows there are in the array. Now we have enough information to convert our algorithm into code:

```
// add up all the numbers in the array
int sum = 0;
for (int row = 0; row < numbers.GetLength(0); row++)
{
    for (int column = 0; column < numbers.GetLength(1); column++)
    {
        sum += numbers[row, column];
    }
}
```

This time we use nested for loops to walk all the dimensions of the array. When the above block of code is done executing, the `sum` variable will hold the sum of all the integers in the array.

## 10.7. Blowing Up Teddies, Take 1

Let's add a little functionality to our teddy bear spawning game. Specifically, we'll make it so pressing the left mouse button blows up all the yellow teddy bears in the game, pressing the right mouse button

blows up all the green teddy bears in the game, and pressing the middle mouse button blows up all the purple teddy bears in the game. Because teddy bears keep spawning, we can do this again and again ...

First, we'll need 3 input axes to respond to the three different mouse button presses. We can certainly add new axes as we did in Chapter 8 if we want to, but we can also simply rename the Fire1, Fire2, and Fire3 axes we get by default because those axes respond to the mouse buttons we want to respond to. The default axes also respond to keyboard keys, so if we want to ONLY respond to mouse buttons we can remove the keyboard key responses from those axes. That's the approach we took in our solution.

Next, we'll need a script that checks for input on these axes on every frame so the teddy bears get blown up as appropriate. Because this is really functionality at the game level rather than the object level, we'll follow our typical approach of attaching the script to the Main Camera. We already have a `TeddyBearSpawner` script attached to the Main Camera, but it's perfectly fine to have multiple scripts attached to game objects. Although we could just add the new functionality to the `TeddyBearSpawner` script, that would be a poor choice because blowing up teddy bears is most definitely NOT spawner functionality! Instead, we'll create a new `BlowingUpTeddies` script that we'll attach to the Main Camera. Here's our new script, with explanations included as appropriate:

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

/// <summary>
/// Blows up teddies in response to player input
/// </summary>
public class BlowingUpTeddies : MonoBehaviour
{
    [SerializeField]
    GameObject prefabExplosion;
```

We include the `prefabExplosion` field so we can instantiate a new `Explosion` game object when necessary; the field is marked with `[SerializeField]` so we can populate it in the Inspector. Recall that in Chapter 8 it was the `TeddyBear` class that had a `prefabExplosion` field. That's because for those problems, the `TeddyBear` game object blew itself up based on player input. In contrast, in this game a different class (`BlowingUpTeddies`) is blowing up the teddy bears, so the `TeddyBear` class doesn't need to have any knowledge of the `Explosion` game objects.

```
[SerializeField]
Sprite yellowTeddySprite;
[SerializeField]
Sprite greenTeddySprite;
[SerializeField]
Sprite purpleTeddySprite;
```

We include these three `Sprite` fields so we can easily check the sprite color for each of the `TeddyBear` game objects in the scene so we only blow up the `TeddyBears` that are the color currently being blown up. The fields are marked with `[SerializeField]` so we can populate them in the Inspector.

```
List<GameObject> gameObjects = new List<GameObject>();
```

When we detect an input from one of the mouse buttons, we're going to need a list of all the game objects in the scene so we can find and destroy the teddy bears of the appropriate color. Rather than creating a new `List` object every time that happens, we'll use a field instead so we don't generate extra `List` objects for the garbage collector to retrieve.

```
/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // FindObjectsOfType is slow, so only call it
    // if at least one of the axes has input
    if (Input.GetAxis("BlowUpYellowTeddies") > 0 ||
        Input.GetAxis("BlowUpGreenTeddies") > 0 ||
        Input.GetAxis("BlowUpPurpleTeddies") > 0)
    {
        gameObjects.Clear();
        gameObjects.AddRange(Object.FindObjectsOfType<GameObject>());
    }
}
```

We know from the Unity documentation that the `FindObjectsOfType` method is slow, so we don't want to call it every frame, we only want to call it if we have input on one or more of the 3 input axes we respond to. If we do have input on at least one of those axes, we clear the list of game objects in the scene, then use the `List AddRange` method to add the current game objects in the scene to the list. Our call to the `Object FindObjectsOfType` method returns an array of all the game objects in the scene, so we pass that array as the argument to the `AddRange` method.

```
// blow up teddies as appropriate
if (Input.GetAxis("BlowUpYellowTeddies") > 0)
{
    BlowUpTeddies(TeddyColor.Yellow, gameObjects);
}
if (Input.GetAxis("BlowUpGreenTeddies") > 0)
{
    BlowUpTeddies(TeddyColor.Green, gameObjects);
}
if (Input.GetAxis("BlowUpPurpleTeddies") > 0)
{
    BlowUpTeddies(TeddyColor.Purple, gameObjects);
}
```

We use the if statements above to call a separate `BlowUpTeddies` method we wrote to blow up all the teddy bears of a particular color (we'll discuss the details of that method soon). The second argument we provide is obviously the list of game objects currently in the scene, but the first argument requires some explanation. Let's leave the `BlowingUpTeddies` script briefly to discuss the first argument.

`TeddyColor` is an enumeration (like `Suit` and `Rank`). Recall that an enumeration essentially defines a new data type with a specific set of values. Take a look at the `TeddyColor` code below:

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

```

/// <summary>
/// An enumeration of the teddy bear colors
/// </summary>
public enum TeddyColor
{
    Green,
    Purple,
    Yellow
}

```

Variables and arguments of the `TeddyColor` type can only have one of three values: `TeddyColor.Green`, `TeddyColor.Purple`, or `TeddyColor.Yellow`. We always need to precede the value with the name of the enumeration. As you know, data types also specify operations that are valid for variables of that data type. Although there are a variety of useful things we can do with enumerations in C#, including converting back and forth between `int` and the enumeration type, in this book we'll be using variables of enumeration types to simply store and compare values.

Before we move on, you might be wondering why we use enumerations at all. For the colors of the teddy bears, for example, why don't we just use an `int` where 0 means green, 1 means purple, and 2 means yellow<sup>24</sup>? There are a number of reasons. First, it's much easier to read code where `TeddyColor.Purple` means the color purple than trying to remember that 1 means purple – remember our previous magic number discussion? You could certainly argue that we already know how to solve the magic number problem, though; we could just declare constants as follows:

```

const int Green = 0;
const int Purple = 1;
const int Yellow = 2;

```

That certainly solves our magic number problem, but it leads to another problem. Let's say that we use an `int` as an argument to the `BlowUpTeddies` method rather than a `TeddyColor` (as it's currently defined). We know that an `int` can store any of  $2^{32}$  unique values, while in this case we only want the variable to be able to hold 0, 1, or 2. Anything other than a 0, 1, or 2 would be an invalid value, so we'd need to add extra code to make sure that whenever the `BlowUpTeddies` method is called that the first argument is a valid value. This check will have to be done at run time, which of course costs us CPU cycles that we could use for something else in our game. If we use the `TeddyColor` enumeration instead, it's impossible to use an invalid value for that argument because the compiler will give us an error (and the check happens at compile time, not run time).

As a reminder, enumerations give us a number of valuable things. They give us code that's easier to read and understand, they enforce a set of specific values for variables and arguments of the enumeration type, and they improve efficiency because all the checking happens at compile time. Enumerations are great, and as you can see from `TeddyColor`, we can define our own enumerations when we need them.

Okay, back to the `BlowingUpTeddies` script:

```

/// <summary>
/// Blows up all the teddies of the given color
/// </summary>

```

---

<sup>24</sup> The same points in the following discussion apply if we decided to use "green", "purple", and "yellow" strings instead of numbers.

```

/// <param name="color">color</param>
/// <param name="gameObjects">the game objects in the scene</param>
void BlowUpTeddies(TeddyColor color, List<GameObject> gameObjects)
{
    // blow up teddies of the given color
    for (int i = gameObjects.Count - 1; i >= 0; i--)
    {
        SpriteRenderer spriteRenderer =
            gameObjects[i].GetComponent<SpriteRenderer>();
        if (spriteRenderer != null)
        {
            Sprite sprite = spriteRenderer.sprite;
            if ((color == TeddyColor.Green &&
                sprite == greenTeddySprite) ||
                (color == TeddyColor.Purple &&
                sprite == purpleTeddySprite) ||
                (color == TeddyColor.Yellow &&
                sprite == yellowTeddySprite))
            {
                BlowUpTeddy(gameObjects[i]);
            }
        }
    }
}

```

Our for loop works through the list of game objects in the scene from back to front because we may be destroying objects in the list (teddy bears of the appropriate color) as we go. Although we won't actually remove those objects from the list, it's a good idea to always go back to front in these situations.

You should never actually use a for loop that goes from the start to the end of a list if the body of the for loop might remove an object from the list. The problem is that we would end up skipping over elements of the list when elements are removed because the elements after the removed element are shifted down in the list to fill the hole the removed element left in the list.

Here's a specific example. Say you're using a standard for loop that starts at 0 to iterate over a list of 4 elements, incrementing the loop control variable on each iteration. If you remove element 2 from the list in the body of the for loop, element 3 shifts down to element 2 and element 4 shifts down to element 3. We now finish the body of the for loop and increment the loop control variable from 2 to 3. That means the loop body will now process element 3 (the old element 4) and will never process the old element 3 (which is now element 2). We skipped over the old element 3, which never got processed by the loop.

In the loop body above, we retrieve the `SpriteRenderer` component for the game object we're currently processing and make sure it isn't `null`. In this particular case, we need to do this because the Main Camera will be one of the game objects in the list; because we want to access the `sprite` field of the sprite renderer for the `TeddyBear` game objects, we need to make sure we don't try to access the `sprite` field of the Main Camera's (non-existent, and therefore `null`) sprite renderer. That's what the `if` statement is for.

If the sprite renderer isn't `null`, we access its `sprite` field. The complicated Boolean expression for the inner `if` statement compares the sprite for the game object to the sprite for the color we're trying to blow up. In more natural language, the Boolean expression checks if we're trying to blow up green teddy bears and the current sprite is green, or we're trying to blow up purple teddy bears and the current sprite

is purple, or we're trying to blow up yellow teddy bears and the current sprite is yellow. If one of those is `true` – remember, for `||` only one of the operands needs to be `true` for the Boolean expression to evaluate to `true` – we call the `BlowUpTeddy` method we wrote:

```
/// <summary>
/// Blows up the given teddy
/// </summary>
/// <param name="teddy">the teddy to blow up</param>
void BlowUpTeddy(GameObject teddy)
{
    Instantiate(prefabExplosion, teddy.transform.position,
    Quaternion.identity);
    Destroy(teddy);
}
```

The `BlowUpTeddy` method simply instantiates an `Explosion` game object at the teddy bear game object's location then destroys the teddy bear game object.

And that's the end of our `BlowingUpTeddies` script.

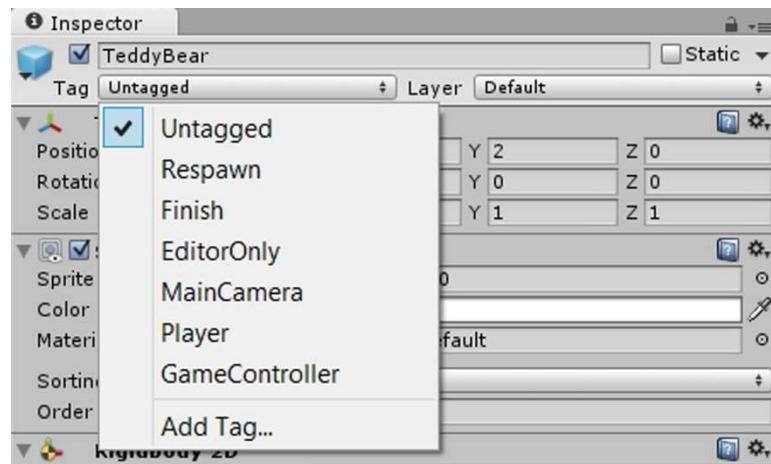
Before our game will work, we need to populate the fields of the `BlowingUpTeddies` script. Select the Main Camera in the Hierarchy window and drag the `Explosion` prefab onto the `Prefab Explosion` field of the script component in the Inspector and drag the appropriate sprites onto the sprite fields.

Run the game to see that we can blow up the teddy bears by color properly.

## 10.8. Blowing Up Teddies, Take 2

One of the things you might have noticed in the previous section is that the `BlowUpTeddies` method has to do more work than it feels like it should have to do because we're getting all the game objects that are currently in the scene and then we have to filter them for the color we're currently trying to blow up. It would be much easier if we could just ask Unity to give us a list of all the green teddy bears when we're trying to blow up green teddy bears (for example). If we could do that, we could just blow up every game object in the provided list without having to check the color of each one first.

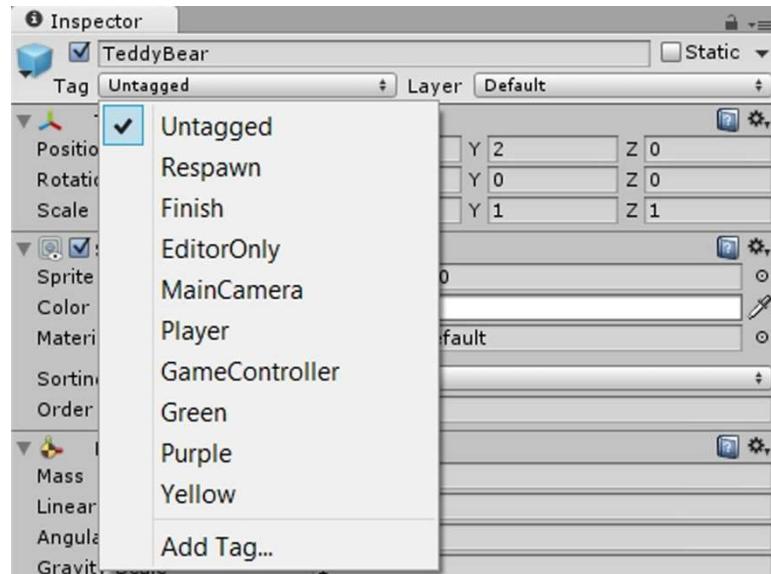
Of course, there is a way to do that in Unity using *tags*. The general idea is that we can tag game objects with a specific tag, then find game objects by their tag instead of by their type. Select the `TeddyBear` prefab in the `prefabs` folder in the Project window. Click the dropdown next to the `Tag` field near the top of the Inspector to see the list of default tags shown in Figure 10.1.



**Figure 10.1. Default Tags**

As you can see, we can select one of the default tags provided by Unity or we can Add Tag... Select Add Tag... and the Unity editor displays the list of user-defined tags in the Inspector. Click the + button at the bottom right of that list to add a new tag to the list; add Green, Purple, and Yellow tags to the list.

Select the TeddyBear prefab in the prefabs folder in the Project window. Click the dropdown next to the Tag field near the top of the Inspector to see the new list of tags shown in Figure 10.2. Set the Tag field for the TeddyBear prefab to Yellow.



**Figure 10.2. Revised Tags**

The next thing we need to do is make sure the `TeddyBearSpawner` `SpawnBear` method sets the tag properly when it spawns a new TeddyBear game object. Here's the new chunk of code that selects a random sprite and sets the sprite and tag appropriately:

```
// set random sprite and tag for new teddy bear
SpriteRenderer spriteRenderer = teddyBear.GetComponent<SpriteRenderer>();
int spriteNumber = Random.Range(0, 3);
```

```

if (spriteNumber < 1)
{
    spriteRenderer.sprite = greenSprite;
    teddyBear.tag = "Green";
}
else if (spriteNumber < 2)
{
    spriteRenderer.sprite = purpleSprite;
    teddyBear.tag = "Purple";
}
else
{
    spriteRenderer.sprite = yellowSprite;
    teddyBear.tag = "Yellow";
}

```

We decided to rename our `Sprite` fields so we don't have to remember which sprite number is which color (we had to populate those fields in the Inspector again because renaming them clears them as well). As you can see, we can simply set the `tag` field for our new `TeddyBear` game object to give it the appropriate tag based on the random sprite we selected.

Next, we modify (and simplify) our `BlowingUpTeddies` `Update` method:

```

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // blow up teddies as appropriate
    if (Input.GetAxis("BlowUpYellowTeddies") > 0)
    {
        BlowUpTeddies(TeddyColor.Yellow);
    }
    if (Input.GetAxis("BlowUpGreenTeddies") > 0)
    {
        BlowUpTeddies(TeddyColor.Green);
    }
    if (Input.GetAxis("BlowUpPurpleTeddies") > 0)
    {
        BlowUpTeddies(TeddyColor.Purple);
    }
}

```

As you can see, we removed the code that retrieves all the game objects that are currently in the scene and also removed the second argument in our call to the `BlowUpTeddies` method. Speaking of that method:

```

/// <summary>
/// Blows up all the teddies of the given color
/// </summary>
/// <param name="color">color</param>
void BlowUpTeddies(TeddyColor color)
{

```

```
// blow up teddies of the given color
gameObjects.Clear();
gameObjects.AddRange(GameObject.FindGameObjectsWithTag(
    color.ToString()));
for (int i = gameObjects.Count - 1; i >= 0; i--)
{
    BlowUpTeddy(gameObjects[i]);
}
}
```

We use the `gameObjects` field here to again hold a list of game objects in the scene, but that list only contains the game objects with the tag we provide as the argument to the `GameObject.FindGameObjectsWithTag` method. We actually made sure our user-defined tags were identical to the `TeddyColor` enumeration values. If we hadn't, we could certainly have used if statements to figure out the appropriate tag to use for our argument, but this way we could just convert the `color` parameter to a string using the `ToString` method and pass the resulting string as our argument.

You'll actually notice in the Unity editor that the game runs without error at this point until you press the Play button to stop playing the game, at which point it says that you're trying to destroy a game object that's already been destroyed. Our guess is that pressing the button to stop the game destroys all the game objects in the scene but the code still responds to the left mouse press as though we're trying to blow up yellow teddy bears. This seems like a reasonable hypothesis, especially since we don't get the error message if we click the Maximize on Play button in the Game view, so we won't make any code changes to try to handle this.

## 10.9. Putting It All Together

Now let's go through the entire problem-solving process for a problem that's most effectively solved using the loops we've learned about in this chapter. Here's the problem description:

- Start with a `TeddyBear` game object, centered in the window, not moving
- On every right mouse click, add a `Pickup` game object where the mouse was clicked
- When the player left clicks the `TeddyBear`, the teddy starts collecting the pickups, starting with the closest `Pickup` and targeting the closest `Pickup` each time it collects the `Pickup` currently "targeted for collection"
- The `TeddyBear` collects a `Pickup` by colliding with it, but this only works for the `Pickup` the `Teddy` has currently "targeted for collection"
- Once the last `Pickup` has been collected, the `Teddy` stops moving
- If the player adds more `Pickups` while the `Teddy` is moving, the `Teddy` picks them up as well
- If the player adds more `Pickups` while the `Teddy` is stopped, the player has to left click on the `Teddy` again to start it collecting again

This is obviously our `Ted the Collector` game from Section 9.7., with the important change in the third bullet that the `TeddyBear` collects the closest pickup rather than the oldest pickup as it collects the pickups in the game.

## Understand the Problem

We've already solved a very similar problem, so the only question we have is what happens if the player places a new Pickup that's closer to the Teddy than the Pickup it's currently moving to collect. Should the Teddy change its target or should it keep heading toward the current targeted Pickup?

Having the Teddy keep heading toward the current targeted Pickup is the easier problem to solve – so let's change the target, because we love a good coding challenge! It will also be more fun to keep making the Teddy change course by placing new Pickups while it's collecting ...

## Design a Solution

We still don't need a `Pickup` script, but we do need to make some changes to our `TeddyBear` and `TedTheCollector` scripts. To figure out what those changes need to be, we need to think about what should happen when the player places a new Pickup in the scene.

As before, the `TedTheCollector` script will add the new Pickup to the list of pickups that class maintains. It's possible, though, that the new Pickup is closer to the Teddy Bear than the Pickup it currently has targeted for collection. In that case, the Teddy Bear should change course to collect the new Pickup instead.

The `TeddyBear` script is the appropriate class to decide whether or not the Teddy Bear should change course, so we need to add a `TeddyBear` `UpdateTarget` method the `TedTheCollector` class calls when a new Pickup is added to the scene. The new method has a parameter for the new Pickup game object so the method can calculate the distance to the new Pickup and the distance to the Pickup that's currently targeted for collection and decide whether or not to change course based on which Pickup is closer. Figure 10.3. shows the UML for the `TeddyBear` class.

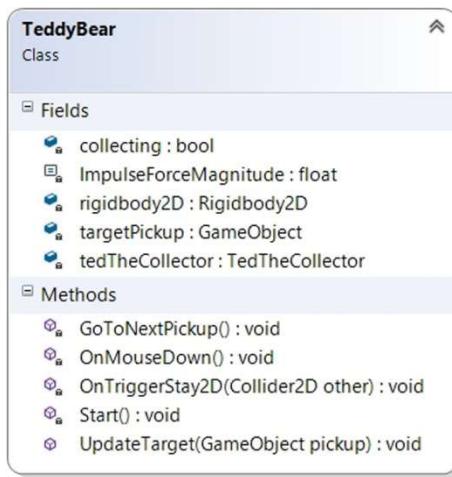


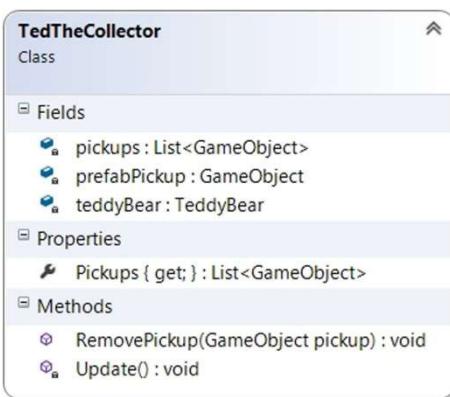
Figure 10.3. `TeddyBear` UML

We also need to change the fields and properties for the `TedTheCollector` class. Because the `TedTheCollector` class now needs to call a `TeddyBear` method, we'll save a reference to the `TeddyBear` class so we don't have to look up that reference every time we add a new Pickup.

We no longer need the `TargetPickup` property that returns the oldest Pickup in the game, because the Teddy Bear no longer cares which Pickup is the oldest. The Teddy Bear does need to figure out which Pickup should be its next target when it finishes collecting a Pickup, though. Because it selects the closest Pickup as its next target, the `TedTheCollector` class should expose a `Pickups` property that returns the list of the pickups in the game. The `TeddyBear` class can then walk that list to find the closest Pickup.

You might think that we should just make the `TedTheCollector` `pickups` field public and let the `TeddyBear` class access that field directly instead of implementing the new `Pickups` property. Although you might see some Unity developers making fields public (usually instead of marking them with `[SerializeField]` so they can be populated in the Inspector in the Unity editor), that's not the right choice. Keeping our field private and exposing a public property is the correct object-oriented approach to use.

The UML for the `TedTheCollector` class is shown in Figure 10.4.



**Figure 10.4. `TedTheCollector` UML**

### *Write Test Cases*

We can still do all our testing in a single test case as shown below.

#### **Test Case 1**

##### **Checking Game Behavior**

Step 1. Input: Right Click

Expected Result: Pickup placed at click location

Step 2. Input: Left Click on Teddy Bear

Expected Result: Teddy Bear collects Pickup and stops

Step 3. Input: Left Click on Teddy Bear

Expected Result: No response (there's no pickup to collect)

Step 4. Input: Right Click

Expected Result: Pickup placed at click location

Step 5. Input: Right Click

Expected Result: Pickup placed at click location

Step 6. Input: Left Click on Teddy Bear

Expected Result: Teddy Bear collects closest Pickup then moves toward next pickup

Step 7. Input: Right Click closer to the Teddy Bear than the currently targeted Pickup while Teddy Bear is moving toward Pickup

Expected Result: Pickup placed at click location, Teddy Bear changes course to collect new Pickup, then moves toward originally targeted Pickup

Step 8. Input: Right Click further away from the Teddy Bear than the currently targeted Pickup while Teddy Bear is moving toward Pickup

Expected Result: Pickup placed at click location, Teddy Bear collects originally targeted Pickup, then collects new Pickup, then stops

Notice that both Steps 7 and 8 have a timing constraint on the input so we can make sure Pickups that are added while the Teddy Bear is moving are collected properly.

### *Write the Code*

As we discussed in the Design a Solution step, we have a number of changes to make to both our classes. Let's start with the `TedTheCollector` `Pickups` property:

```
/// <summary>
/// Gets the pickups currently in the scene
/// </summary>
/// <value>pickups</value>
public List<GameObject> Pickups
{
    get { return pickups; }
}
```

Now we can move on to the `TeddyBear` `GoToNextPickup` method, where we change a single line of code from

```
targetPickup = tedTheCollector.TargetPickup;
```

to

```
targetPickup = GetClosestPickup();
```

That of course means we decided to write a new `GetClosestPickup` method that finds the pickup in the scene that's closest to the Teddy Bear:

```
/// <summary>
/// Gets the pickup in the scene that's closest to the teddy bear
/// If there are no pickups in the scene, returns null
/// </summary>
/// <returns>closest pickup</returns>
GameObject GetClosestPickup()
{
    // initial setup
    List<GameObject> pickups = tedTheCollector.Pickups;
    GameObject closestPickup;
    float closestDistance;
    if (pickups.Count == 0)
    {
        return null;
```

```

    }
else
{
    closestPickup = pickups[0];
    closestDistance = GetDistance(closestPickup);
}

```

The chunk of code above returns `null` if there aren't any pickups in the scene just like the `TedTheCollector` `TargetPickup` property used to do. The else body is only executed if there's at least one pickup in the scene, so it sets `closestPickup` to the first pickup in the list. It also sets `closestDistance` to the distance between that pickup and the teddy bear using a `GetDistance` method we wrote (we'll discuss that method when we're done with this one).

```

// find and return closest pickup
foreach (GameObject pickup in pickups)
{
    float distance = GetDistance(pickup);
    if (distance < closestDistance)
    {
        closestPickup = pickup;
        closestDistance = distance;
    }
}
return closestPickup;
}

```

The `foreach` loop iterates over all the pickups in the scene. The first thing we do in the body of the loop is calculate the distance from the teddy bear to the pickup we're currently processing. If the distance to that pickup is less than the distance to the closest pickup we've found so far, we just found a new closest pickup. In that case, we save the new `closestPickup` and `closestDistance`. When the `foreach` loop completes, we return the closest pickup we found in the scene.

You should note that some programmers take the reasonable position that every method should only have a single return in it; a return statement is where we exit the method and return to the caller of the method. Our method has two return statements instead: one if there are no pickups in the scene and one after we've searched the list of pickups in the scene for the closest one. There's certainly a way to write our method to have only a single return statement, but in our opinion that leads to less efficient code that's harder to understand, so we decided to opt for efficiency and code clarity instead.

Our `GetDistance` method is pretty simple:

```

/// <summary>
/// Gets the distance between the teddy bear and the
/// provided pickup
/// </summary>
/// <returns>distance</returns>
/// <param name="pickup">pickup</param>
float GetDistance(GameObject pickup)
{
    return Vector3.Distance(transform.position, pickup.transform.position);
}

```

This method uses the `Vector3` `Distance` method, which we found by reading the `Vector3` documentation, to calculate and return the distance between the teddy bear's location and the pickup's location. Although we could have simply used the `Vector3` `Distance` method in the body of our `GetClosestPickup` method instead, we felt that using this new method made that code easier to read (and we'll use it again before we're all done).

If you execute the test case at this point, everything works fine through Step 6 but Step 7 fails. This is understandable, since we haven't implemented or used the `TeddyBear` `UpdateTarget` method yet. The good news, though, is that our `GetClosestPickup` method is working properly.

Here's our new `TeddyBear` `UpdateTarget` method:

```
/// <summary>
/// Updates the pickup currently targeted for collection.
/// If the provided pickup is closer than the currently
/// targeted pickup, the provided pickup is set as the
/// new target. Otherwise, the targeted pickup isn't
/// changed.
/// </summary>
/// <param name="pickup">pickup</param>
public void UpdateTarget(GameObject pickup)
{
    float targetDistance = GetDistance(targetPickup);
    if (GetDistance(pickup) < targetDistance)
    {
        targetPickup = pickup;
        GoToTargetPickup();
    }
}
```

This code simply calculates the distance to our current target pickup and the distance to the pickup parameter. If the pickup parameter is closer, we set that as the new target pickup (by setting the `targetPickup` field) and start moving toward it. Notice that if the pickup parameter isn't closer we don't need to do anything, we just keep moving toward our current target pickup.

When we were getting ready to write the code to calculate the direction to the target pickup and add the impulse force to start the teddy bear moving in that direction, we realized that we already had code in the `GoToNextPickup` method that does that. We couldn't just call the `GoToNextPickup` method, though, because it also looks at all the pickups in the scene to find the closest one. That's definitely not the functionality we need here!

Our solution was to pull the code we need out into a new `GoToTargetPickup` method and call our new method from both the `GoToNextPickup` method and the `UpdateTarget` method:

```
/// <summary>
/// Starts the teddy bear moving toward the target pickup
/// </summary>
void GoToTargetPickup()
{
```

```
// calculate direction to target pickup and start moving toward it
Vector2 direction = new Vector2(
    targetPickup.transform.position.x - transform.position.x,
    targetPickup.transform.position.y - transform.position.y);
direction.Normalize();
rigidbody2D.velocity = Vector2.zero;
rigidbody2D.AddForce(direction * ImpulseForceMagnitude,
    ForceMode2D.Impulse);
}
```

Now we need to have the `TedTheCollector` script call the `TeddyBear` `UpdateTarget` method when a new Pickup is added to the scene. As we said in the Design a Solution step, we've added a `TeddyBear` field to our class for efficiency. We do need to do a little more work to populate that field.

Our general approach is to tag the `TeddyBear` with a new tag called `TeddyBear`, use the `GameObject` `FindGameObjectWithTag` method to find the `TeddyBear` game object, then find and save its `TeddyBear` component in our field. We'll do all the coding in the `Start` method, which we'll need to add back in to the `TedTheCollector` class because we removed it for the previous problem. Add the new `TeddyBear` tag and set the tag for the `TeddyBear` prefab to the new tag. Remember, changing the prefab changes all instances of that prefab in the scene, so the `TeddyBear` game object in the scene is also now tagged (select it in the Hierarchy window if you'd like to confirm that).

Here's the code for our `Start` method:

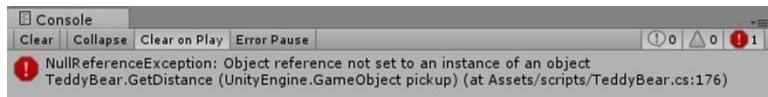
```
/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // save reference for efficiency
    GameObject teddyBearGameObject =
        GameObject.FindGameObjectWithTag("TeddyBear");
    teddyBear = teddyBearGameObject.GetComponent<TeddyBear>();
}
```

The last thing we need to do is add the following code to the `TedTheCollector` `Update` method right after we add a new pickup to the scene:

```
// have teddy bear update its target
teddyBear.UpdateTarget(pickup);
```

### *Test the Code*

The test case passes when we execute it, so we'd like to declare victory and move on. Unfortunately, we get the error message shown in Figure 10.5. whenever we add the first pickup to the scene (even after the teddy bear has collected pickups, then stopped).



**Figure 10.5. Error Message**

Even though the game doesn't crash, it would be really ugly to leave this error in our code, so let's fix it now.

### *Write the Code Again*

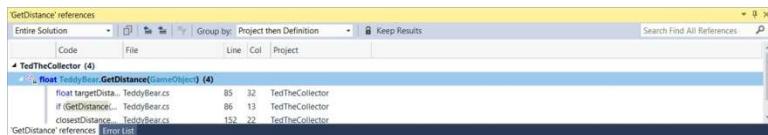
The error message tells us that the error occurs at line 176 in the `TeddyBear.cs` file; that line is

```
return Vector3.Distance(transform.position, pickup.transform.position);
```

in the `GetDistance` method. How do we know that's line 176? By default, Visual Studio shows us line numbers, but if they're not showing up for you, you can show them by selecting Tools > Options from the menu bar, expanding the Text Editor heading, selecting the All Languages heading, checking the Line Numbers checkbox in the pane on the right, and clicking OK.

The error message also tells us that the error is a `NullReferenceException`. We get this error when we try to access a field or property (or call a method) of a null object. We know that `transform.position` can't be `null` because our `TeddyBear` game object has a Transform component, so we can form a hypothesis that `pickup` is `null`.

To check our hypothesis, we need to look at the places where other code calls the `GetDistance` method. To do this, right-click on the `GetDistance` method header and select Find All References (about halfway down the popup). The resulting search results window is shown in Figure 10.6.



**Figure 10.6. GetDistance References**

The search results list the method header and every call to the method, including the line number for each line. We can even double-click one of the lines in the search results window and Visual Studio takes us to that line in the code; when we do that for the first result, we go to line 85, the first line of code in the `UpdateTarget` method body.

Aha! We know the error occurs when we're adding the first pickup in the scene, which means the `targetPickup` field is currently `null` because the Teddy Bear hasn't identified a target yet. That means we're passing `null` in as the argument to the `GetDistance` method here, which is exactly what's causing our problem. We can avoid this error by changing the `UpdateTarget` method to:

```
public void UpdateTarget(GameObject pickup)
{
    if (targetPickup == null)
    {
```

```
        targetPickup = pickup;
        GoToTargetPickup();
    }
}
else
{
    float targetDistance = GetDistance(targetPickup);
    if (GetDistance(pickup) < targetDistance)
    {
        targetPickup = pickup;
        GoToTargetPickup();
    }
}
```

## *Test the Code Again*

Now we confidently run the code again and ... ^&\$\*#\$&#\$!@\$\*@#

Now the Teddy Bear immediately starts collecting when we place the first Pickup! Why didn't we see this problem before? Because our `UpdateTarget` method was crashing before it got to the call to the `GoToTargetPickup` method. Now that we've fixed the crash, we get to that method call, which moves the Teddy Bear toward the target pickup even though it's not collecting yet.

Sigh.

## *Write the Code, Yet Again*

While we fix this problem, we can also clean up our `UpdateTarget` method a little. We have two places that contain identical lines of code:

```
targetPickup = pickup;  
GoToTargetPickup();
```

To fix this – because we know duplicated code is BAD – we'll pull these two lines of code into a separate `SetTarget` method and call that method from both places. This is also a good idea because we're about to make that code a little more complicated:

```
    /// <summary>
    /// Sets the target pickup to the provided pickup
    /// </summary>
    /// <param name="pickup">pickup</param>
    void SetTarget(GameObject pickup)
    {
        targetPickup = pickup;
        if (collecting)
        {
            GoToTargetPickup();
        }
    }
}
```

Now we always set the `targetPickup` field (as we should) but only start moving toward the (new) target pickup if the Teddy Bear is already collecting.

### *Test the Code, Yet Again*

Unfortunately, we've now made it so Step 7 fails. The Teddy Bear keeps moving toward its original targeted pickup, then keeps going past it without collecting it.

This might feel frustrating to you, but we WANT our test cases to expose errors in our code! It's much better for us to discover these problems during development than it is to ship the game and then discover them. It may not feel like it, but this is a good thing.

### *Write the Code (for the fourth time)*

We again form a debugging hypothesis before we start digging around in the code. We know that the Teddy Bear only collects a Pickup when it collides with it if that Pickup is set as its current target. The fact that the Teddy Bear goes past its original target implies that the target pickup was changed (as it should have been for Step 7 in the test case). We also know that the Teddy Bear didn't start moving toward the new (closer) target pickup, which it only does if `collecting` is `true`.

If we go to the code, right-click the `collecting` field, and select Find references, we discover that the field is never set in the code at all (except when it's initialized)! Let's fix that now.

First, we decide when `collecting` should be set to `true`. That should happen in the `OnMouseDown` method just before we call the `GoToNextPickup` method.

Next, we decide when `collecting` should be set to `false`. That should happen in the `GoToNextPickup` method if there are no more Pickups to be collected in the scene. We know that's the case if the `GetClosestPickup` method returns `null` when we call it.

So why didn't we see this problem in our solution in the previous chapter? Because `collecting` is initialized to `false` and the only Boolean expression that used `collecting` was checking if `collecting` was `false`, so that Boolean expression always evaluated to `true`. That check is in the `OnMouseDown` method, so it actually would send the Teddy Bear to the next Pickup even if it was currently collecting, but because the rule was to go to the oldest Pickup in the scene, that never changed where the Teddy Bear was going.

### *Test the Code (for the fourth time)*

Finally, the test case passes without errors!

## 10.10. Common Mistakes

### *Using the Wrong Condition in a For Loop*

If you forget that arrays and lists are zero-based rather than one-based, you'll end up using `<=` rather than `<` in your for loop condition when comparing to the size (`Count` for a list, usually `Length` for an array). This will make you try to access one too many elements at the end of the for loop, which will make your program blow up with an `IndexOutOfRangeException` exception. If that happens to you, go back and check your condition.

*Changing the Array or Collection You're Iterating Over With a Foreach Loop*

This isn't allowed, so your code will crash if you do this. If you need to remove elements from the array or collection you're iterating over while you're iterating you need to use a (back to front) for loop instead.

# Chapter 11. Iteration: While Loops

In the previous chapter we introduced for and foreach loops; in both those loops, we know how many times we expect the loop to iterate when we reach the loop during program execution. This chapter covers a different form of iteration where we don't know how many times the loop will iterate when we reach the loop. The decision about whether or not to keep looping for the loops discussed in this chapter is based on some condition that typically isn't checking whether a counter has reached a particular value.

## 11.1. Iteration Control Structure Revisited

We've already seen the value of for and foreach loops in the previous chapter, so let's focus our attention here on loops where we don't know how many times the loop will iterate. As usual, we'll start with an example.

### Example 11.1. Getting a Valid GPA

Problem Description: Write an algorithm that will ask for a GPA until a valid GPA is entered.

Algorithm: Here's one solution:

```
Prompt for and get GPA
While GPA is less than 0.0 or greater than 4.0
    Print error message
    Prompt for and get GPA
```

We use indentation to show what happens inside the loop, just as we used indentation to show what happened inside our other selection and iteration algorithms. In this example, the third and fourth steps are contained inside the loop (recall that the steps inside the loop are called the *loop body*).

Basically, our solution asks for a GPA. While the GPA is invalid (less than 0.0 or greater than 4.0), we print an error message and ask for a new GPA. Eventually, the user will enter a valid GPA, stopping the iteration of the last three steps in the algorithm. The associated CFG is shown in Figure 11.1.

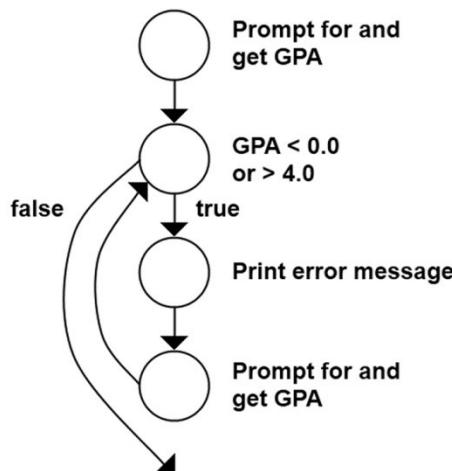


Figure 11.1. CFG for Example 11.1.

The first thing we do is prompt for and get a GPA from the user. We then move to the node that "tests" to see whether the GPA is invalid. Note that the test in this node is a Boolean expression like the one we saw for the selection control structure. If the GPA is valid, we take the branch marked false, skipping the loop body. If the GPA is invalid, however, we take the branch marked true, which lets us print an error message and read in a new GPA. Notice the edge from the bottom node back up to the test node; after we've read in the new GPA, we go back to the test node to check if the new GPA is invalid. Eventually, the user will enter a valid GPA, the Boolean expression at the "test" node will evaluate to false, and we'll take the left branch (the one marked false) out of the iteration control structure.

## 11.2. While Loops

The most common form of loop we use to implement these kinds of loops in C# (and many other languages) is the *while loop*. The syntax for while loops is shown below.

---

### SYNTAX: While Loop

```
while (Boolean expression)
{
    loop body
}
```

*Boolean expression*, this is just like the Boolean expressions we used in our if statements; it simply evaluates to `true` or `false`. The Boolean expression is evaluated each time we get to the `while` part of the loop. If it's `true`, we go into the loop and execute the loop body; if it's `false`, we exit the loop. The Boolean expression for a while loop should generally contain at least one variable *loop body*, the code that's executed each time through the loop

---

Let's implement our algorithm from Example 11.1 using a while loop. Our algorithm is as follows:

```
Prompt for and get GPA
While GPA is less than 0.0 or greater than 4.0
    Print error message
    Prompt for and get GPA
```

First, we'll add the `while` part of the loop (since that's our new concept), then we'll fill in the rest.

```
// prompt for and get GPA

// loop for a valid GPA
while (gpa < 0.0 || gpa > 4.0)
{
    // print error message and get new GPA
}
```

Can you see how the value of `gpa` determines whether we keep looping or exit the loop? If the `gpa` is out of range (either less than 0.0 or greater than 4.0), the Boolean expression evaluates to `true` and we execute the loop body. If the `gpa` is in range, the Boolean expression evaluates to `false` and we end the loop. Let's implement the rest of the algorithm:

```
// prompt for and get GPA
Console.WriteLine("Enter a GPA (0.0-4.0): ");
double gpa = double.Parse(Console.ReadLine());

// loop for a valid GPA
while (gpa < 0.0 || gpa > 4.0)
{
    // print error message and get new GPA
    Console.WriteLine("Invalid entry! GPA must be between 0.0 and 4.0.");
    Console.WriteLine();
    Console.Write("Enter a GPA (0.0-4.0): ");
    gpa = double.Parse(Console.ReadLine());
}
```

The first thing we do is get the GPA from the user, then while the GPA is invalid we print an error message and get a new GPA from the user. How many times does the body of the while loop execute? It depends on the user. If the user enters a valid GPA the first time, the loop body is never executed. If the user enters an invalid GPA first, then enters a valid GPA, the loop body executes once. You get the idea.

There are three things we need to do with the variables in the Boolean expression for every while loop we write, and we can remember those things with the acronym **ITM**. Before we get to the loop, we need to **Initialize** the variables contained in the Boolean expression so that they have values before we get to the line starting with `while`, which **Tests** the Boolean expression to see if the loop body should execute or not. Finally, within the loop body, we need to **Modify** at least one of the variables in the Boolean expression (give it a new value). So for every while loop, we need to make sure the variables in the Boolean expression are Initialized, Tested, and Modified – ITM.

So what happens if we forget about ITM? If we don't initialize the variables in the Boolean expression before the loop, the test in the `while` part is simply based on whatever happens to be in those memory locations (and the chances of those being exactly the values we want are pretty slim). If we don't properly test the Boolean expression, the decision about whether to loop or not won't be based on the condition we really want to check. And if we don't modify at least one of the variables in the Boolean expression inside the loop body, the loop will go on forever (commonly called an *infinite loop*)! Here's why: if we get into the loop body, the Boolean expression in the `while` part must have been `true`. If we don't change at least one of the variables inside the loop body, the Boolean expression is still `true`, so we loop again, and again, and again ...

In the loop above, we initialize `gpa` (the variable in the Boolean expression of the while loop) when we prompt for and get the GPA right before getting to the while loop. The GPA is then tested in the Boolean expression of the while loop to see if it's valid. If the GPA is valid we don't execute the loop body. If the GPA is invalid, though, we print an error message, modify the GPA by reading in a new value from the user, and loop back around to the start of the loop.

The while loop lets us implement algorithms in which we don't know how many times we need to loop. In fact, though, the while loop is so general that we can also use it to solve problems where we do know how many times we need to loop. Let's revisit our "printing the squares" example from the previous chapter, but this time we'll use a while loop to solve the problem instead of a for loop.

*Example 11.2. Printing Squares of the Integers from 1 to n*

Problem Description: Write a program that will print the squares of the integers from 1 to n, where n is provided by the user.

Algorithm: Here's one possible algorithm:

```
Prompt for and get n
Set i to 1
While i is less than or equal to n
    Print the square of that integer
    Increment i
```

The algorithm above loops through the body of the loop n times (from 1 to n), and each time it prints the square of the current integer.

When we turn this algorithm into code, we get

```
// get number of squares to print
Console.WriteLine("Enter the number of squares to print: ");
int n = int.Parse(Console.ReadLine());

// print the squares
int i = 1;
while (i <= n)
{
    Console.WriteLine("The square of {0} is {1}", i, i * i);
    i++;
}
```

You should be able to easily see how similar this is to the for loop we used in the previous chapter to solve the same problem. We still initialize `i` to 1 at the start of the loop, we still check if `i` is `<=` the number the user entered on each iteration, and we still add 1 to `i` on each iteration. Although it's far more common to use a for loop for this kind of problem, we did want to show you how a while loop can be used for this problem as well.

### 11.3. Testing While Loops

In the previous chapter, we stated that testing for and foreach loops is like testing sequential code. That's not true for while loops, however.

When we use a while loop in our program, "completely" testing the program becomes impossible. Think about a single loop – to really test it, you'd have to execute the loop body zero times, execute the loop body once, execute the loop body twice, execute the loop body three times, ... you see where this is going, right? When the program contains while loops, we need to compromise. The best way to test such programs is to execute each loop body zero times, one time, and multiple times. And of course we still need to test the boundary values for each Boolean expression just like we did for the selection control structure.

Let's talk about how we should test the while loop we used in Example 11.1 above. How do we execute the loop body (the nodes that print an error message and get a new GPA) zero times? By entering a valid

GPA the first time we're asked. To execute the loop body one time, we enter an invalid GPA followed by a valid one. To execute the loop body multiple times, we enter a few invalid GPAs followed by a valid one. To test the boundary values, we want to input the following values for GPA: -0.1, 0.0, 4.0, and 4.1. An example test plan is provided below.

### Test Case 1

**Loops: Loop Body 0 Times**

**Boundary Value: 0.0**

Step 1. Input: 0.0 for GPA

Expected Result: Exit Program

### Test Case 2

**Loops: Loop Body 1 Time**

**Boundary Values: -0.1 and 4.0**

Step 1. Input: -0.1 for GPA

Expected Result: Error message, reprompt

Step 2. Input: 4.0 for GPA

Expected Result: Exit Program

### Test Case 3

**Loops: Loop Body Multiple Times**

**Boundary Value: 4.1**

Step 1. Input: -0.1 for GPA

Expected Result: Error message, reprompt

Step 2. Input: 4.1 for GPA

Expected Result: Error message, reprompt

Step 3. Input: 4.0 for GPA

Expected Result: Exit Program

These three test cases cover the boundary values and executing the loop body 0, 1, and multiple times. Notice that the last step in each test cases says "Exit program." Remember, each test case is for a complete execution of a program. Our program is really quite useless at this point – it doesn't even do anything with the GPA once it has a valid one! – but it's still a program.

In addition, we're actually using information about the structure of our solution rather than a problem description to decide what the test cases should do. That makes these test cases white-box tests rather than black-box tests, which in turn makes them unit tests rather than functional tests.

## 11.4. Do-While Loops

Although a while loop can be used to implement any loop, some people prefer using a different loop structure called a *do-while loop*. The syntax for do-while loops is described below; note that the do-while loop is essentially a while loop where the test occurs after the loop body rather than before it. That means that the loop body of a do-while loop always executes at least once, but the loop body of a while loop might not be executed at all.

## SYNTAX: Do-While Loop

```
do
{
    loop body
} while (Boolean expression);
```

*Boolean expression*, the Boolean expression is evaluated each time we get to the `while` part of the loop. If it's `true`, we loop back around and execute the loop body again; if it's `false`, we exit the loop. The Boolean expression for a do-while loop should typically contain at least one variable *loop body*, the code that's executed each time through the loop

---

Let's solve the problem from Example 11.1. one more time, this time using a do-while loop. Our algorithm is:

```
Loop
    Prompt for and get GPA
    If the GPA is less than 0.0 or greater than 4.0
        Print error message
    While the GPA is less than 0.0 or greater than 4.0
```

The resulting C# code is as follows:

```
// loop for a valid GPA
double gpa;
do
{
    // prompt for and get GPA
    Console.WriteLine("Enter a GPA (0.0-4.0): ");
    gpa = double.Parse(Console.ReadLine());

    // print error message for invalid GPA
    if (gpa < 0.0 || gpa > 4.0)
    {
        Console.WriteLine("Invalid entry! GPA must be between 0.0 and 4.0");
        Console.WriteLine();
    }
} while (gpa < 0.0 || gpa > 4.0);
```

This is a bit more awkward than using the while loop for this example, because we need the if statement to make sure we print the error message only if the current GPA is invalid. Although there are examples where a do-while loop provides a more elegant solution than a while loop, we personally find that we almost never use do-while loops in practice. We did want to cover them here for completeness, though.

One quick comment about testing do-while loops. Because the loop body of a do-while loop is always executed at least once, we can't test 0 iterations of a do-while loop; we can only test the loop body executing 1 and multiple times.

## 11.5. Spawning Into a Collision-Free Location

Remember that in the Putting It All Together problem in Chapter 7, we said that we shouldn't spawn a teddy bear on top of a teddy bear that's already in the game, but that we needed to know about while loops first. Well, now we know about while loops, so we can solve this problem.

All our work will be in the `TeddyBearSpawner` script because we only need to handle this problem when we're spawning a new teddy bear. The big idea behind our approach is that we'll pick a random location for the teddy bear we're spawning, then check to see if the collider for the new teddy bear would collide with anything already in the game. If it wouldn't, we spawn the teddy bear at that location and we're done. If it would collide with something, though, we randomly generate a new location and do the check again. We keep doing this until we find a collision-free location for the new teddy bear or we decide to give up on this spawn attempt.

We start by adding five more fields:

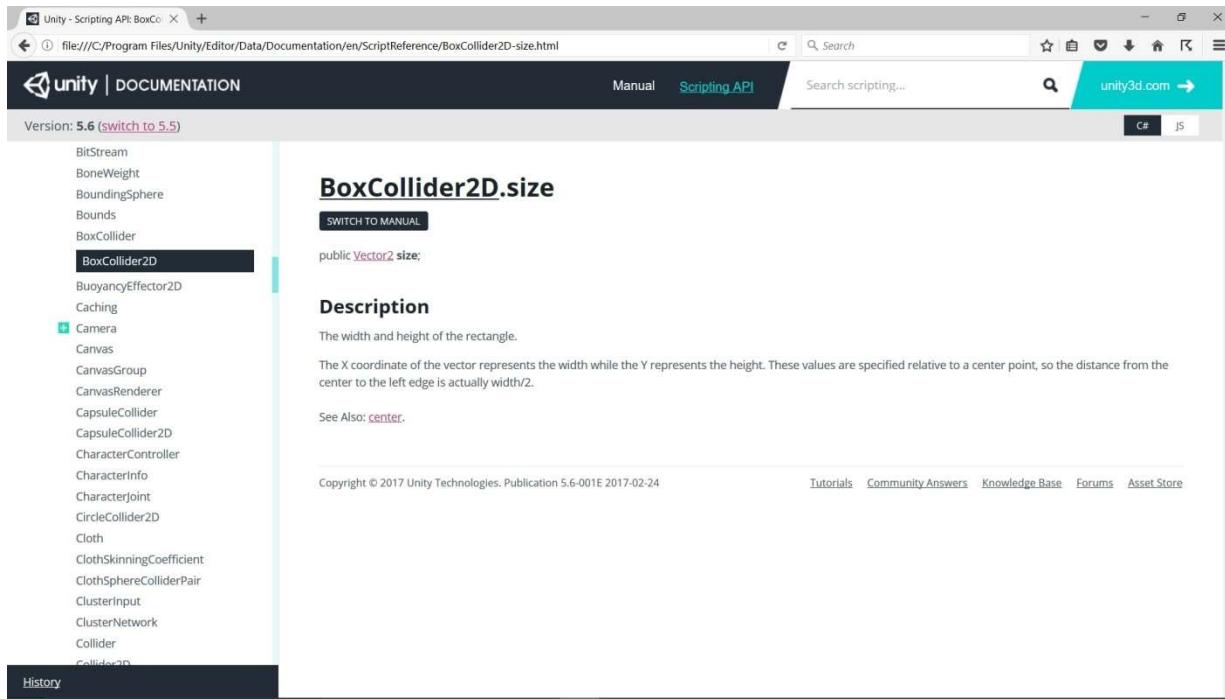
```
// collision-free spawn support
const int MaxSpawnTries = 20;
float teddyBearColliderHalfWidth;
float teddyBearColliderHalfHeight;
Vector2 min = new Vector2();
Vector2 max = new Vector2();
```

We use the constant to make sure we don't end up in an infinite loop as we spawn a new teddy bear; we'll discuss how we use that below. Because we need to check for a collision at least once, and possibly several, times when we want to spawn a new teddy bear, saving the dimensions we'll use to check for that collision saves us some processing time. We'll change the x and y components of the `min` and `max` fields when we check for collisions, which saves us from creating two new `Vector2` objects every time we spawn a new bear.

Next, we add code to the `TeddyBearSpawner` `Start` method to retrieve and save those dimensions:

```
// spawn and destroy a bear to cache collider values
GameObject tempBear = Instantiate(prefabTeddyBear) as GameObject;
BoxCollider2D collider = tempBear.GetComponent<BoxCollider2D>();
teddyBearColliderHalfWidth = collider.size.x / 2;
teddyBearColliderHalfHeight = collider.size.y / 2;
Destroy(tempBear);
```

The first two lines of code above create a new teddy bear object so we can retrieve the teddy bear collider. To generate the third and fourth lines of code, we found the `size` member in the `BoxCollider2D` documentation (see Figure 11.2) and used the width (the `x` component) and the height (the `y` component) to calculate the dimensions we need. The fifth line of code destroys the teddy bear we created because we're done with it.



**Figure 11.2. BoxCollider2D Size Documentation**

Our final change is to the `TeddyBearSpawner` `SpawnBear` method. We replace the code we originally had to generate a random location and create the new teddy bear object with the following code (with discussion embedded as appropriate):

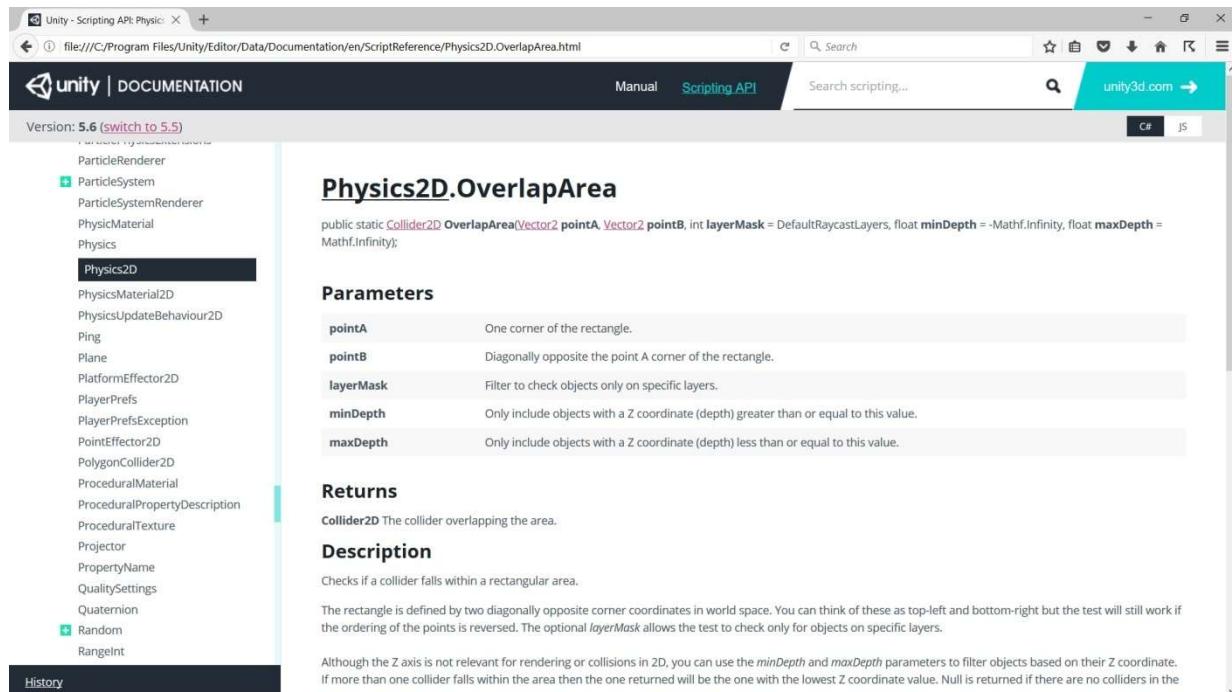
```
// generate random location and calculate teddy bear collision rectangle
location.x = Random.Range(minSpawnX, maxSpawnX);
location.y = Random.Range(minSpawnY, maxSpawnY);
location.z = -Camera.main.transform.position.z;
Vector3 worldLocation = Camera.main.ScreenToWorldPoint(location);
SetMinAndMax(worldLocation);
```

The first three lines of code select a random location in screen coordinates and the fourth line of code converts those screen coordinates into world coordinates (We declared `location` as a field so we don't need to create a new `Vector3` object for it on each spawn). The final line of code calls a `SetMinAndMax` method we wrote; we'll look at that method after we're done with this method, but the `SetMinAndMax` method sets the `min` field to the coordinates of the upper left corner of a collision rectangle for a teddy bear if it were placed at `worldLocation` and it sets the `max` field to the coordinates of the lower right corner of a collision rectangle for a teddy bear if it were placed at `worldLocation`.

```
// make sure we don't spawn into a collision
int spawnTries = 1;
while (Physics2D.OverlapArea(min, max) != null &&
spawnTries < MaxSpawnTries)
{
    // change location and calculate new rectangle points
    location.x = Random.Range(minSpawnX, maxSpawnX);
    location.y = Random.Range(minSpawnY, maxSpawnY);
    worldLocation = Camera.main.ScreenToWorldPoint(location);
    SetMinAndMax(worldLocation);
```

```
    spawnTries++;
}
```

We use the `spawnTries` variable to keep track of how many times we've generated a location to see if it's collision-free; that way, we can make sure we don't get stuck in an infinite loop if the game is crowded with lots of teddy bears. We built our Boolean expression so the while loop would execute while the current location would spawn the new teddy bear into a collision and we haven't tried the maximum number of spawn tries yet. Figure 11.3 shows the documentation for the `Physics2D.OverlapArea` method we use for the first part of that expression.



**Figure 11.3. Physics2D OverlapArea Documentation**

As the documentation states, the method returns `null` if there's no collision for the given rectangle corners, so if the method doesn't return `null` there is a collision for those corners. You might be confused by the fact that we called the method with 2 arguments even though the documentation lists 5 parameters for the method. It turns out that C# has a way to specify optional parameters; the `layerMask`, `minDepth`, and `maxDepth` parameters are optional for this method.

```
// create new bear if found collision-free location
if (Physics2D.OverlapArea(min, max) == null)
{
    GameObject teddyBear = Instantiate(prefabTeddyBear) as GameObject;
    teddyBear.transform.position = worldLocation;

    // set random sprite for new teddy bear
    SpriteRenderer spriteRenderer = teddyBear.GetComponent<SpriteRenderer>();
    int spriteNumber = Random.Range(0, 3);
    if (spriteNumber < 1)
    {
        spriteRenderer.sprite = teddyBearSprite0;
    }
}
```

```
    }
    else if (spriteNumber < 2)
    {
        spriteRenderer.sprite = teddyBearSprite1;
    }
    else
    {
        spriteRenderer.sprite = teddyBearSprite2;
    }
}
```

Most of the code above is identical to what we had in Chapter 7; the only difference is that we put it in an if statement to ensure we only spawn the teddy bear in a collision-free location. Remember that the while loop could have ended because we reached the max spawn tries, so we need to include the check above before spawning the teddy bear.

That's all the code in the `SpawnBear` method; here's the `SetMinAndMax` method:

```
    /// <summary>
    /// Sets min and max for a teddy bear collision rectangle
    /// </summary>
    /// <param name="location">location of the teddy bear</param>
    void SetMinAndMax(Vector3 location)
    {
        min.x = location.x - teddyBearColliderHalfWidth;
        min.y = location.y - teddyBearColliderHalfHeight;
        max.x = location.x + teddyBearColliderHalfWidth;
        max.y = location.y + teddyBearColliderHalfHeight;
    }
```

You should be able to see how the method sets the `min` field to the coordinates of the upper left corner of a collision rectangle for a teddy bear if it were placed at `location` by calculating the x and y values for that corner using the center of the teddy bear and the `teddyBearColliderHalfWidth` and `teddyBearColliderHalfHeight` values we calculated in the `Start` method. We do similar processing for the `max` field, which holds the coordinates of the lower right corner of a collision rectangle for a teddy bear if it were placed at that location.

## 11.6. Putting It All Together

Our problem here is different from our previous Putting It All Together problems in other chapters because for this problem we're just going to write (and test) a single method rather than developing a complete program. Don't worry, we'll go back to solving larger problems again in the coming chapters, but we wanted to focus on an interesting while loop here.

Here's the problem description:

Develop a method that starts at a particular location in a list of strings, searching for the next occurrence of "inactive". The search needs to examine every string in the list, returning the index of the next occurrence of "inactive" or -1 if "inactive" doesn't appear in the list.

This problem is actually based on a problem we needed to solve in one of our commercial games. In our game, we need to search for a weapon launcher with a particular characteristic for our weapon selection logic. Because we need to select the next launcher with that characteristic, starting from the currently selected launcher, this is essentially the same problem we're solving here.

### *Understand the Problem*

To make sure we understand the problem, let's think about all the possibilities for occurrences of "inactive" in the list (this will help us develop our test cases as well). We already know what to do if "inactive" doesn't appear in the list, since the problem description tells us to return -1 in this case. It also seems clear that if we find "inactive" later than the search start location in the list we should return its index, and we should do the same thing if we find "inactive" before the search start location.

What if the string at the search start location is the only occurrence of "inactive" in the list? The problem description says we need to look at every string in the list, so we need to return the search start location if that's the only occurrence of "inactive" in the list. By the way, this was exactly what we needed in our commercial game as well, since if the currently selected weapon launcher was the only launcher with the required characteristic, we wanted to keep it selected.

### *Design a Solution*

We're going to develop a single method to solve this problem, so we don't need to worry about how a set of objects will interact in our problem solution. We will, however, need to carefully figure out the steps we need to follow in the method, so we'll actually come up with an algorithm for the method before we implement the code. We'll do that at the beginning of our Write the Code step.

### *Write Test Cases*

Let's start by creating test cases for the four scenarios we thought about in the Understand the Problem section.

We already know what to do if "inactive" doesn't appear in the list, since the problem description tells us to return -1 in this case. It also seems clear that if we find "inactive" later than the search start location in the list we should return its index, and we should do the same thing if we find "inactive" before the search start location. We also realized as we worked to Understand the Problem that we need to return the search start location if that's the only occurrence of "inactive" in the list.

We should also include test cases where we start the search at the very beginning of the list and at the very end of the list. In some sense, these are like the boundary values we've tested in our selection and while loop test cases.

#### **Test Case 1**

##### **Inactive Not In List, Starting Search at Beginning of List**

Step 1. Input: None. Hard-coded search start at index of 0 with list of "hey", "hi", "yo"  
Expected Result: -1 for index

**Test Case 2****Inactive Later In List**

Step 1. Input: None. Hard-coded search start at index of 1 with list of "hey", "hi", "yo", "inactive"

Expected Result: 3 for index

**Test Case 3****Inactive Earlier In List**

Step 1. Input: None. Hard-coded search start at index of 1 with list of "inactive", "hey", "hi", "yo"

Expected Result: 0 for index

**Test Case 4****Only Inactive At Search Start Index**

Step 1. Input: None. Hard-coded search start at index of 1 with list of "hey", "inactive", "hi", "yo"

Expected Result: 1 for index

**Test Case 5****Starting Search at End Of List**

Step 1. Input: None. Hard-coded search start at index of 3 with list of "hey", "inactive", "hi", "yo"

Expected Result: 1 for index

*Write the Code*

Because this is a fairly complicated problem, we're going to develop a solid algorithm before we implement that algorithm in C#.

As always, there are a number of ways we can solve this problem. Here's one solution (that doesn't happen to use a while loop):

```

For each string from the search start index + 1 to the end of the list
    If the string is "inactive", return its index
For each string from the start of the list to the search start index - 1
    If the string is "inactive", return its index
If the string at the search start index is "inactive"
    Return the search start index
Otherwise
    Return -1

```

There are a couple problems with this solution. The largest problem is that we'll have 3 if statements that check for "inactive", which means we'll end up duplicating essentially the same code in multiple places in our method. Although that's not really a huge deal for this problem, it would be worse if the check was more complicated (like it is in our weapon selection problem). You should also be able to see that the above algorithm might have us break out of our for loops when we find "inactive" in the list, and some programmers object to this behavior in for loops. Let's come up with a better algorithm that actually uses a while loop.

We'll start by figuring out what condition should be `true` for us to keep looping. This is actually a little trickier than it might seem, because we should keep looping while we haven't found "inactive" in the list yet and while we haven't looked at all the strings in the list yet. Instead of trying to specify that in a Boolean expression, we'll use a variable (often called a *flag* in this context) to keep track of that for us. The start of our algorithm is therefore

```

Set search complete to false
While the search isn't complete
    <fill in more steps here>

```

We also need to know where to start our search. Because we only want to return the index of the string at the search start location if it's the only occurrence of "inactive" in the list, we should start our search with the string at the location just past the search start location.

What happens if we find an occurrence of "inactive" inside our while loop? We should save its index and set the search complete flag to `true` (since we just found what we were looking for). After the while loop, we can simply return the saved index. Here's our next cut at the algorithm:

```

Set inactive index to -1
Set i to search start index + 1
Set search complete to false
While the search isn't complete
    If the string at i is "inactive"
        Set inactive index to i
        Set search complete to true
    <fill in more steps here>
Return inactive index

```

You might be wondering why we initialized inactive index to -1. We did that so that if the body of the while loop never changes that variable (which will be the case if "inactive" never appears in the list) it will still be -1 when we return it at the end of the method.

The algorithm above actually has a bug that we'll fix in a moment. If the search start location provided to us is actually the last string in the list, we'll end up trying to check a string at a location `i` that's past the end of the list.

We still have a little more work to do on our algorithm, because we haven't made it so we move along the list on each loop iteration, nor have we added logic to check when we've looked at every string in the list. Let's add that logic (and the logic to fix our bug) now:

```

Set inactive index to -1
Set i to search start index + 1
If i is >= the number of strings in the list
    Set i to 0
Set search complete to false
While the search isn't complete
    If the string at i is "inactive"
        Set inactive index to i
        Set search complete to true
    If i is equal to the search start index
        Set search complete to true
    Otherwise
        Add 1 to i
        If i is >= the number of strings in the list
            Set i to 0
Return inactive index

```

The second if statement in our loop body sets our search complete flag to `true`; we know we started our search at the search start index + 1, so if we're looking at the string at the search start index, we just checked the last string in the list and our search is complete.

In the otherwise part, we know we should keep searching the list so we add 1 to the current index so we can look at the next string in the list on the next loop iteration. After adding 1 to `i`, we need to check to see if it's time to wrap around to the beginning of the list. We added that same wrapping code before the loop when we initialize `i` to fix the bug we discussed above.

To add those algorithm steps in multiple places, we copied and pasted the steps in the algorithm. Whenever you find yourself doing a copy and paste in an algorithm or in code, you should IMMEDIATELY stop and decide whether or not you should create a new method for the duplicated functionality. We haven't really discussed how to write our own methods yet (coming soon, we promise), so we'll leave it this way here. We don't like it, though!

You should work through the algorithm convincing yourself that it will work properly for the various scenarios we considered in the Understand the Problem section. You should also convince yourself there isn't a much easier solution that just uses the index in a test for our while loop rather than using the search complete flag; comparing the index to the search start location seems intuitive but isn't really much better. Here's what that algorithm could look like:

```

Set inactive index to -1
Set i to search start index + 1
If i is >= the number of strings in the list
    Set i to 0
While i isn't equal to search start index and inactive index is -1
    If the string at i is "inactive"
        Set inactive index to i
    If inactive index is -1
        Add 1 to i
        If i is >= the number of strings in the list
            Set i to 0
    If the string at search start index is "inactive"
        Set inactive index to i
Return inactive index

```

The above algorithm is actually a couple steps shorter than the algorithm we developed, but we find it less satisfying because the condition for the while loop is harder to understand and we have to duplicate the check for "inactive" for the string at the search start index after the loop is done.

As we said, there are many ways to solve problems we come across in game development, and you should certainly explore a different solution to this problem if you find an algorithm that seems more intuitive to you. For the rest of this chapter, though, we'll use the algorithm that uses the search complete flag.

Because we've done such a thorough job coming up with our algorithm, converting the algorithm to the required method is straightforward:

```

/// <summary>
/// Finds the index of the first occurrence of "inactive" in the list of
/// strings, starting at the provided index + 1. Checks all strings in the
/// list, wrapping around to the beginning of the list if necessary.
/// Returns -1 if "inactive" doesn't occur in the list
/// </summary>
/// <param name="startIndex">the start index</param>
/// <param name="strings">the list of strings</param>
/// <returns>the index of the "inactive" string or -1</returns>
static int FindInactiveStringLocation(int startIndex,
    List<string> strings)
{
    // initialize search variables
    int inactiveIndex = -1;
    int i = startIndex + 1;
    if (i >= strings.Count)
    {
        i = 0;
    }
    bool searchComplete = false;

    while (!searchComplete)
    {
        // check for and save inactive string info
        if (strings[i] == "inactive")
        {
            inactiveIndex = i;
            searchComplete = true;
        }

        // check for end of search
        if (i == startIndex)
        {
            searchComplete = true;
        }
        else
        {
            // move along list, wrapping if necessary
            i++;
            if (i >= strings.Count)
            {
                i = 0;
            }
        }
    }

    return inactiveIndex;
}

```

Now it's time to make sure the code works as we expected.

*Test the Code***Test Case 1****Inactive Not In List, Starting Search at Beginning of List**

Step 1. Input: None. Hard-coded search start at index of 0 with list of "hey", "hi", "yo"

Expected Result: -1 for index

**Test Case 2 (Original)****Inactive Later In List**

Step 1. Input: None. Hard-coded search start at index of 1 with list of "hey", "hi", "yo", "inactive"

Expected Result: 3 for index

**Test Case 3****Inactive Earlier In List**

Step 1. Input: None. Hard-coded search start at index of 1 with list of "inactive", "hey", "hi", "yo"

Expected Result: 0 for index

**Test Case 4****Only Inactive At Search Start Index**

Step 1. Input: None. Hard-coded search start at index of 1 with list of "hey", "inactive", "hi", "yo"

Expected Result: 1 for index

**Test Case 5****Starting Search at End Of List**

Step 1. Input: None. Hard-coded search start at index of 3 with list of "hey", "inactive", "hi", "yo"

Expected Result: 1 for index

Our test plan seems complete, but remember we said we should execute the bodies of while loops 0, 1, and multiple times in our testing. At this point all the test cases execute the loop body multiple times, so let's see if we can get test cases for 0 and 1 time.

To get the loop body to execute once, we can provide a list where "inactive" is in the location immediately following the start location. Although we could add an additional test case for this, each additional test case costs us time (and, in practice, money) to write and execute. Instead, let's modify the list we provide in Test Case 2 to cover this.

**Test Case 2****Inactive Later In List**

Step 1. Input: None. Hard-coded search start at index of 1 with list of "hey", "hi", "inactive"

Expected Result: 2 for index

How about a test case for making the loop body execute 0 times? It actually turns out that this is impossible. Because we set the search complete flag to `false` before the loop and then test it when we get to the loop, there's no way for us to come up with a test case that skips the loop body based on the search start index and list of strings we provide as input. So even though we should always try to execute our while loop bodies 0, 1, and multiple times, there are going to be times when that just isn't possible.

To test the code, we embedded the method we wrote into a console application and had the `Main` method in that application execute all the test cases for us. For example, here's the code we wrote to execute Test Case 1:

```
// Test Case 1
List<string> strings = new List<string>();
strings.Add("hey");
strings.Add("hi");
strings.Add("yo");
int index = FindInactiveStringLocation(0, strings);
if (index == -1)
{
    Console.WriteLine("Test Case 1 passed");
}
else
{
    Console.WriteLine("TEST CASE 1 FAILED!!");
}
```

Notice that the code for the test case provides the specific inputs for the test case, and also includes the expected result for the test case (for Test Case 1, our expected result is -1 for the index). By structuring our test code this way, we make it really easy to tell whether or not the test cases in the test plan passed without having the tester try to evaluate each of the actual results to determine whether or not it matches the expected result.

Go ahead and look at the code accompanying the book to see the code for the other test cases. When we run our entire test plan, we get the following results shown in Figure 11.4.

```
C:\WINDOWS\system32\cmd.exe
Test Case 1 passed
Test Case 2 passed
Test Case 3 passed
Test Case 4 passed
Test Case 5 passed

Press any key to continue . . .
```

**Figure 11.4. Test Case Results**

As you can see, all our test cases passed, so we're done solving this problem.

## 11.7. Common Mistakes

### *Forgetting ITM in a Loop*

The most common cause of infinite loops (loops that just keep going) is the programmer forgetting about ITM. We have to make sure all the variables contained in the Boolean expression for the loop are initialized so that they have values before we get to the test the first time. If we don't properly test the

Boolean expression, the decision about whether to loop or not won't be based on the condition we really want to check. We also need to make sure that at least one of the variables in the Boolean expression is modified in the loop before we get back to the test again. If (when) you write a program containing an infinite loop, check ITM.

*Using an If Statement Instead of a While Loop*

Some people have a tendency to use an if statement when they're trying to get valid user inputs. This would only work if we had a GUARANTEE that the user would never enter two invalid inputs in a row. We'd need that guarantee because there's no way for an if statement to "loop back", so we could never catch the second invalid input. A while (or do) loop is definitely the way to go when we're trying to get valid user inputs.

# Chapter 12. Class Design and Implementation

Up to this point in the book, we've been using lots of C# classes, both those that are provided in the C# and Unity namespaces and those that we've come up with ourselves. We've even written some code within our own classes, "finishing" those classes so they'd do what we need them to do. Now that you understand the basics of C# and the three control structures, it's time for us to look at how we really go about designing and implementing our own classes from scratch.

In this chapter, our focus will be on designing and implementing a specific class rather than an entire system of interacting classes. Developing complete systems is typically deferred to classes following the first programming class in most game development and computer science programs, so we won't tackle formally designing and implementing full systems in this book<sup>25</sup>.

## 12.1. High-level Class Design

Well, that section heading seems pretty vague, doesn't it? Our main goal in this section is to create a UML class diagram for the class we're trying to design. As we develop more of our own classes, we'll use this process during the Design a Solution step in our problem-solving process.

To make our discussions throughout this chapter more concrete, let's work through the design and implementation of a new class. Here's a problem description:

Design and implement a class for a deck of cards. The class should implement standard operations that we perform on a card deck.

Well, we're probably going to need to figure out what's required in a little more detail (Understand the Problem) before we can start on our design.

For example, is this a standard deck of 52 cards with the typical ranks and suits? For this problem, we'll assume the answer is yes. Our bigger problem is figuring out what the standard operations are.

So what can we do with a deck of cards? We certainly need to be able to shuffle the deck, and we should also provide the capability to cut the deck. What about dealing cards from the deck? This gets a little trickier, because the way we deal cards is often dependent on the game we're playing. Rather than trying to make the deck use that information, we can simplify the deck and even keep it more general by instead providing a method to take the top card from the deck. That makes the deck general enough to use for any card game using the standard 52 card deck; other classes using the `Deck` class can handle the details of how many cards get dealt to each player and so on.

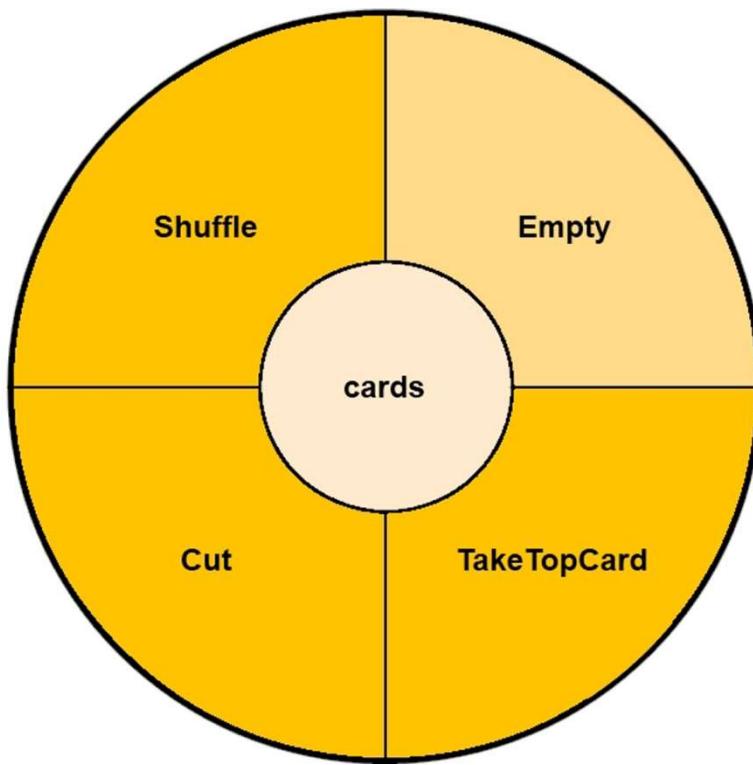
Is that everything? Almost. Let's think about how a deck is used in games in which the deck is used to deal cards to each player, then gets used as the draw pile in the game. We can still take the top card from the deck to draw a card from the pile, but at some point someone may take the last card from the draw pile (making the deck empty). Would the next player try to get the top card from the empty draw pile? We hope not (if you're playing with someone who would try that, you should be playing for money)! In

---

<sup>25</sup> The final chapter, though, does implement a complete game with multiple classes. We take a somewhat informal approach to developing that system and defer more formal techniques (like use cases and UML sequence diagrams) to later development efforts.

real life we can tell the pile is empty by looking at it; we're therefore going to need to provide a property for our `Deck` class that tells if the deck is empty.

That leads us to the pictorial representation of the `Deck` class shown in Figure 12.1. Remember from our discussion about classes and objects way back in Chapter 4 that we don't let consumers of the class see "inside" the object to use it. Instead, the consumer of the class uses object properties and methods to get it to do what they need it to do without worrying about what fields are inside the object or even how the properties and methods work. In other words, consumers of the class will be able to use everything that's provided on the outer part of the diagram below.



**Figure 12.1. The Deck Class**

We're now well on our way to our UML diagram. We've already identified the properties and methods we're going to want for our class, but we also need to know the fields (variables and constants used throughout the class) we're going to use, the data types of those fields, and the data types returned by our properties and methods. Let's start with our fields.

In our case, we only have one field: `cards`. Now, we could actually use 52 fields – one for each of the cards in the deck – but that would be a very awkward approach to use, especially when we had to process those fields with the properties and methods. What we'd really like is some object that could hold all 52 cards. We're obviously not the first C# programmers to need a class we could use for such an object; in fact, we already know of several classes we could use. Specifically, either an array of `Card` objects or a `List` of `Card` objects should work well for this. Let's use a `List`, since the contents of the deck change as we take cards from the deck.

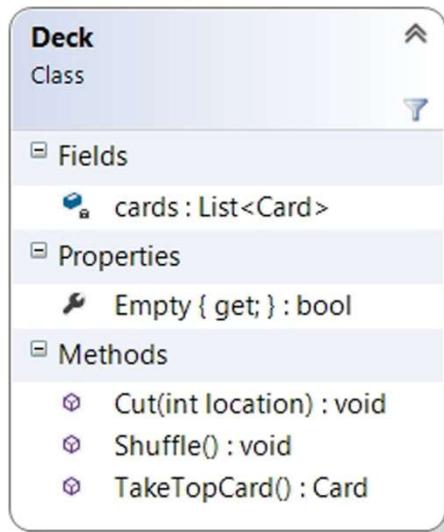
Next we'll look at our `Empty` property. The most reasonable thing to do is to have the property return `true` if the deck is empty and `false` otherwise. We'll worry about how to make that happen when we Write the Code.

Now let's look at our methods, starting with the `Cut` method. We're going to need a single piece of information from the code calling the method; specifically, the method will need to know where to cut the deck. Let's say that the place is specified as a number, where 0 would mean to cut at the top card, 1 would mean to cut at the second card, and so on. Should the method return anything? No, because only the order of the cards in the deck is changed and the consumer of the class doesn't (and shouldn't) know about the order of the cards in the deck.

What about our `Shuffle` method? It doesn't need any information from the code that calls the method because the deck just shuffles itself. Does the method need to return anything to the code that calls the method? No, because all that happens is that the cards in the deck get shuffled into a different order, so there's nothing we need to return.

The `TakeTopCard` method won't need any parameters either, because we're simply going to take the top card from the deck. The method does have to return something, though: the `Card` object from the top of the deck.

Well, it looks like we've done all the design we need to create the UML diagram; that diagram is provided in Figure 12.2.



**Figure 12.2. Deck UML Diagram**

We still have to Write the Code for our `Deck` class, of course, but we'll defer that step until later. Just for reference, Figure 12.3. shows what the documentation for the `Deck` class will look like when we're all done.

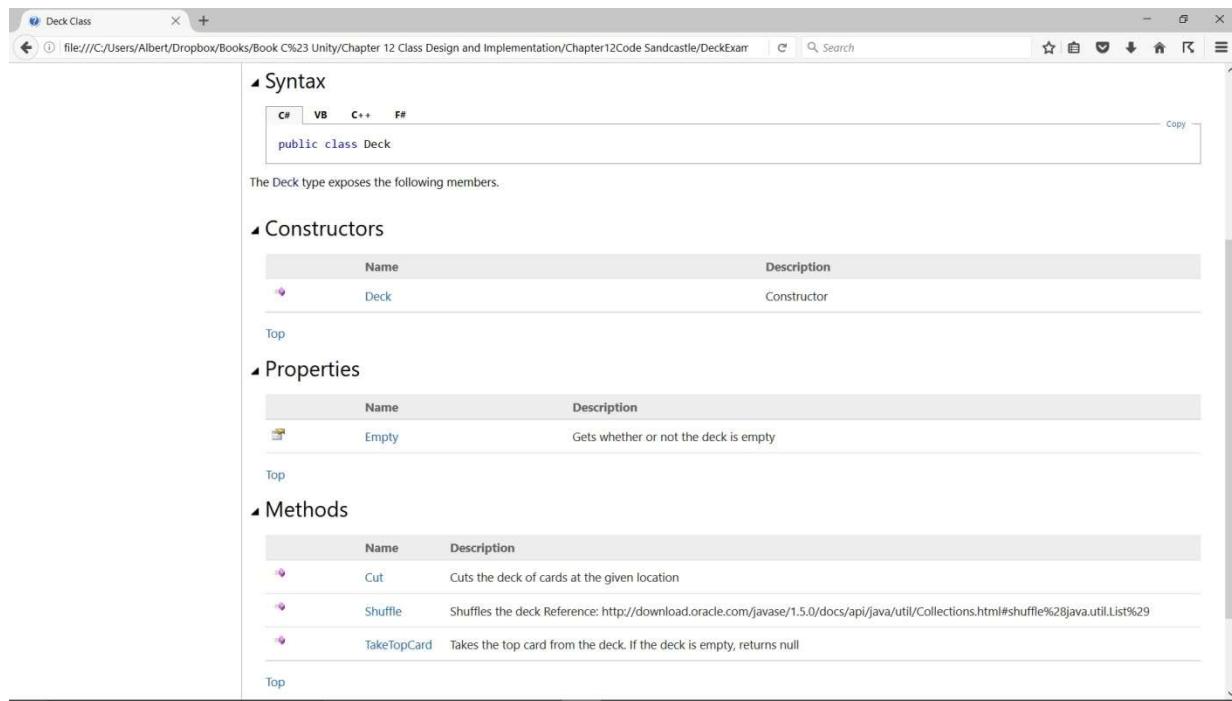


Figure 12.3. Deck Class Documentation

## 12.2. Implementing Classes in C#: Fields

Before we implement our `Deck` class in C#, there's a handy syntax description below showing how a class is defined in C#.

### SYNTAX: Class Definition

```
using directives for namespaces

namespace NamespaceName
{
    documentation comment
    [access modifier] [static] class ClassName
    {
        fields

        constructors

        properties

        methods
    }
}
```

`NamespaceName`, the namespace the class belongs to  
`documentation comment`, comment describing the class  
`access modifier`, optional modifier that tells the visibility of the class  
`static`, optional modifier to identify a static class

*ClassName*, the name of the class being defined

*fields*, variables and constants used by the class or by objects created from the class

*constructors*, used when an object is created from the class

*properties*, properties exposed or used by the class or by objects created from the class

*methods*, methods exposed or used by the class or by objects created from the class

Note: The square brackets ([ and ]) above are not part of the C# code, they're just used to indicate which parts of the given syntax are optional

---

We'll discuss access modifiers in greater detail below, but for now, note that if we're writing a class for our programs or others to use, we use `public` as our access modifier. For the examples in this book, the only other kind of class we'll define is the application class, which doesn't need an access modifier at all.

Using the provided syntax, let's work on implementing our `Deck` class in C#. Before we do that, though, how do we actually add a new class to our project within the IDE? Right click the project name (it's in bold, near the top) in the Solution Explorer window and select Add > New Item... Now select Class in the middle pane of the dialog, change the Name of the class to something reasonable (Deck for our example), then click the Add button. You can see in the Solution Explorer window that the new class has been added to your project.

Open the new `Deck` class by double-clicking it in the Solution Explorer window.

The namespace will be set to whatever we called our project when we created it (we called ours `DeckExample`). We also need to include a class documentation comment, as we'll always do. We're going to want to make this class accessible to others (`public`), so here's how we start our class definition:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DeckExample
{
    /// <summary>
    /// A deck of cards
    /// </summary>
    public class Deck
    {
```

Next, recall that a class has four parts: fields, constructors, properties, and methods. Let's take a look at fields first, then we'll look at the other parts. Fields include both variables and constants. What are fields for? They hold the *state* of the object.

Declaring the fields for our class is very similar to the way we've been declaring variables and constants up to this point. We'll discuss the details for fields that are variables, but the same concepts apply to constants as well. For fields, there are two things we need to worry about that we haven't had to deal with for our variables and constants so far. Specifically, we need to decide on the *visibility* of the

variable, and we need to decide if the variable is a *static variable* or an *instance variable*. Here's the syntax for declaring a field (that's a variable) in a class:

---

#### SYNTAX: Field (Variable) Declaration in a Class

```
[access modifier] [static] dataType variableName;
```

*access modifier*, optional modifier that tells the visibility of the variable  
*static*, optional modifier distinguishing between static and instance variables  
*dataType*, the data type for the variable  
*variableName*, the name of the variable

---

You'll notice, of course, that the last two parts of the variable declaration – the data type and variable name – are identical to the way we declared variables in the past. If the variable is in fact an object rather than a value type, then we of course use a class name for the data type for that object.

The first new thing we see is the optional *access modifier* for each variable. This tells C# how “accessible” the variable is; in other words, who can see and use it. We'll have five choices for this: `public`, `private` (the default in classes we define), `protected`, `internal`, and `protected internal`.

In this book, we'll almost always make our fields `private`, and we suggest that you do the same. A `private` variable is one that can only be looked at and modified by properties and methods contained in the class that declares the variable. Making our variables `private` helps us keep our variables hidden inside our objects so that objects can only interact with one another through the object properties and methods. Because making our variable `private` is such a good idea, if we don't provide an access modifier at all the default for the variable in a class is `private`, so you only need to add an access modifier if you want something other than `private` for that variable.

By the way, that's why we've been making the (`private` by default) fields in our Unity scripts visible in the Inspector by marking them with `[SerializeField]`. You'll probably see some Unity developers making the fields `public` instead of using `[SerializeField]` to make those fields show up in the Inspector, but that's a really bad approach to use because it breaks our information hiding.

Public fields can be accessed directly by methods and objects outside the class; we'll try to avoid `public` variables like the plague, but `public` constants are fine because consumers of the class can't mess them up. Protected fields can be accessed within the class and by subclasses of this class (when we want to use inheritance, which we'll discuss later in the book); we'll only use `protected` fields if we're using inheritance in our problem solution. The last two access modifiers, `internal` and `protected internal`, won't be required for the problems we solve in this book (nor do we use them in any of the commercial games we develop).

The next thing we need to decide for our fields is whether we want them to be static variables or instance variables. If we want a static variable, we include the `static` keyword after the access modifier; if we want an instance variable, we leave it off. So what's the difference?

Let's talk about instance variables first, since they'll be the ones we use most commonly in this book. Whenever we create an object from a class, that object gets its own copies of the instance variables, and that object is the only one that can modify those instance variables (as long as we made them `private`). If we create three `Deck` objects, for example, each of those objects will have its own list of the cards in the deck.

We definitely want the `cards` variable in our deck to be `private` – you don't want someone to be able to change the order of the cards in the deck without shuffling or cutting the deck, right? We also want this to be an instance variable, since each deck needs to have its own list of cards. So here's what we include in our class definition:

```
List<Card> cards = new List<Card>();
```

So what about static variables? A static variable is a single variable that gets “shared” by every object that's been created for that class. Say we want to give each deck of cards a unique ID. Not only is this a classic example for static variables, we've actually had this precise need in one of our commercial games for our game objects. The highest ID that's been used so far isn't information that really belongs in one particular object of the class; it's really class-wide information, since it applies to all objects of that class. If we use a static variable, we can assign each new deck object a unique ID when we create it (that ID would be stored in an instance variable for the new object), then increment the static variable to make sure the next deck object we create gets a different ID. To make a variable a static variable instead of an instance variable, all we have to do is to include `static` in our variable declaration:

```
static int nextId;
```

And that's all you need to know to declare fields in your classes.

### 12.3. Implementing Classes in C#: Properties

Although we actually usually implement the constructors in our classes right after declaring the fields, we're going to defer the constructor discussion until the following section. In this section, we'll look at how we go about defining properties.

Properties are typically used to provide access to the fields in the class; in other words, to provide access to the state of the object. The three kinds of access we can provide for a property are read access, write access, and read-write access. We read a property using a `get` accessor and we write a property using a `set` accessor. If we want to provide both read and write access (typically called read-write access) to the property, we simply provide both `get` and `set` accessors for the property. The syntax for defining properties is provided below.

#### SYNTAX: Property Definition in a Class

```
documentation comment
[access modifier] [static] dataTypePropertyName
{
    get { executable statements }
    set { executable statements }
}
```

*documentation comment*, comment describing the property  
*access modifier*, optional modifier that tells the visibility of the property  
*static*, optional modifier distinguishing between static and instance properties  
*dataType*, the data type read and/or written for the property  
*PropertyName*, the name of the property  
*executable statements*, code to execute when reading or writing the property

---

Let's start with a simple example that's not related to the `Deck` class. Let's say we have a `Weapon` class that inflicts a certain amount of damage when the weapon is used in an attack. Consumers of the `Weapon` class will need to be able to read the damage value to know how much damage to inflict. In addition, our game may provide power-ups that actually increase the damage inflicted by the weapon either temporarily or permanently. In either case, consumers of the class will need to be able to write the damage value to make the change. The best way to handle this is to provide a `Damage` property as follows (assuming we have an `int` `damage` field already declared):

```
/// <summary>
/// Gets and sets the damage inflicted by the weapon
/// </summary>
public int Damage
{
    get { return damage; }
    set { damage = value; }
}
```

Before we discuss how the property works, you've probably noticed that for the `get` and `set` accessors we're breaking our rule that our open curly braces are always on a new line. Because it's very common for those accessors to be a single line of code, it's common C# programming style to put the entire accessor on a single line of code. For more complicated accessors (which we'll see soon), we follow our standard curly brace placement style.

The `get` accessor returns the value of the `damage` field and the `set` accessor sets the `damage` field to the `value` that's provided. We provide the `value` by putting it on the right-hand side of the assignment statement that's setting the property; this is like passing a single argument into the property using special syntax. Specifically, here's the syntax we use to set a property:

---

### SYNTAX: Set (Write) a Property

`objectName.PropertyName = value;`

*objectName*, the name of the object  
*PropertyName*, the name of the property we're setting  
*value*, the value to set the property to

---

So if we set the `Damage` property of a `Weapon` object (let's call it `someWeapon`) using

```
someWeapon.Damage = 42;
```

`value` is 42 inside the `set` accessor in the property code shown above and the internal `damage` field inside `someWeapon` gets set to 42.

So why not just make our fields `public` instead of `private` – then we won’t need to use properties at all? There are several reasons. First, when we make a field `public`, any consumer of the class can both read and write the field. That might be fine in some cases, but in other cases (like the `Empty` property in our `Deck` class) we don’t want to let them both read and write. It’s therefore better – and easier – to always make our fields `private` and provide the properties necessary to let consumers access them as appropriate.

Second, we can provide error-checking in the `set` accessor to make sure the consumer setting the property isn’t setting it to an invalid value. For example, we can change our `Damage` property to make sure we clamp the property to 0 if the consumer tries to set the `Damage` property to a negative number:

```
/// <summary>
/// Gets and sets the damage inflicted by the weapon
/// </summary>
public int Damage
{
    get { return damage; }
    set
    {
        if (value >= 0)
        {
            damage = value;
        }
        else
        {
            damage = 0;
        }
    }
}
```

That’s not something we could do with a `public` field.

Finally, we can expose properties that are intuitive to the consumer of the class but require a little extra processing “under the hood.” Let’s say we wanted to provide the upper left corner of the collider for a `TeddyBear` object:

```
/// <summary>
/// Gets the upper left corner for the collider
/// </summary>
public Vector2 ColliderUpperLeftCorner
{
    get
    {
        BoxCollider2D collider = GetComponent<BoxCollider2D>();
        return new Vector2(
            collider.transform.position.x - collider.size.x / 2,
            collider.transform.position.y - collider.size.y / 2);
    }
}
```

```
}
```

By including some extra processing logic in our `get` accessor, we can provide an intuitive property to consumers of the class without exposing them to the details of the implementation. This is actually how the `Empty` property in our `Deck` class will work also, so we'll implement that property in a moment.

In some cases we might want the accessors of a read-write property to have different access modifiers; for example, we might want the `get` accessor to be `public` and the `set` accessor to be `private`. In this case, we set the access modifier for the entire property to `public` and add `private` before the `set` keyword for the `set` accessor.

Okay, how can we tell if our deck is empty? Our deck is empty when it doesn't have any more cards in it; in other words, when the list of cards in the deck is empty. That observation leads us to the following implementation of the property:

```
/// <summary>
/// Gets whether or not the deck is empty
/// </summary>
public bool Empty
{
    get { return cards.Count == 0; }
}
```

A consumer of this property might use this property in the following way:

```
if (myDeck.Empty)
```

That means the `get` accessor needs to return a value to the consumer; we use `return` to make the accessor return a value. The Boolean expression `cards.Count == 0` evaluates to `true` if the list is empty and `false` otherwise, so we can simply return the value of that expression as the value of the property.

As discussed above, this property does some extra processing to make the result useful to the consumer of the property while hiding the internal implementation details of the class. You should also note that this property is read-only. We certainly want consumers of the class to be able to find out whether or not the deck is empty, but we don't want them to be able to make the deck empty by using write access to the property.

## 12.4. Implementing Classes in C#: Constructors and Other Methods

The final parts of our class are the constructors and other methods exposed by the class. As you know, constructors are used to create new objects, where each object has its own *identity*. Methods implement the *behavior* of the object. So the combination of fields, properties, methods, and constructors gives each object the state, behavior, and identity we discussed in Chapter 4.

Let's take a look at the syntax we use to define a method.

## SYNTAX: Method Definition in a Class

```
documentation comment
[access modifier] [static] returnType MethodName (
    dataType parameterName,
    dataType parameterName,
    . . .
)
{
    method body
}
```

*documentation comment*, comment describing the method

*access modifier*, optional modifier that tells the visibility of the method

*static*, optional modifier distinguishing between static and instance methods

*returnType*, the data type returned by the method

*MethodName*, the name of the method

*dataType*, the data type of a parameter

*parameterName*, the name of a parameter

*method body*, code the method executes when called

The first part of the method (the part starting with the access modifier and ending just before the open curly brace) is called the *method header*. Let's take a closer look.

The access modifier for a method works exactly the same way as an access modifier for the fields and properties in the class; we choose between `public`, `protected`, `private`, `internal`, and `protected internal` to determine who can see and use this method. Remember that we said we should make all the variables in our class `private`? Well, for methods, we're going to want to make them `public` if consumers of the class need to call them and `private` if they're only used internally in our class (we've already done this in our examples in previous chapters).

The next thing we need to decide for our methods is whether we want them to be static methods or instance methods; this is the same idea as static variables and instance variables. If we decide that the method is an instance method – the most common kind in this book and the most common kind you'll use in general – then each object will have its own copy of this method. If we decide that a method is a static method, we include the word `static` in our method declaration. One quick caution – static methods are NOT allowed to access instance variables. So if you need to get at an instance variable, you need to use an instance method to do that.

Static methods can be very useful at times. For example, recall that when we use `Console.WriteLine`, we're using the static `WriteLine` method, so we don't have to create a console object. However, static methods generally represent a very small percent of the methods you'll write, so think carefully before deciding to make a method static.

The third thing we see in the method header is the return type for the method. This tells us which data type will be returned by the method. For example, if we're taking the top card from the deck, we want our method to return a `Card`. If our method was calculating a speed for a bicycle, we'd want the method

to return a `float`. There will be lots of times, however, when we don't want to return anything at all (we're shuffling or cutting the deck, for example). In those cases, we pick `void` as our return type. This simply means that this particular method doesn't return anything.

Next, we pick our method name, which should be descriptive (just like our field and property names). The final part we need to worry about in the method header is the list of parameters for this method. Parameters seem to be one of the most confusing parts of programming for beginning programmers, but there's a straightforward way to think about them. If you understand that parameters are used to pass information between the code that calls the method and the method itself you're well on your way to mastering them. By the way, that's how we've been using parameters all along!

Finally, we include the *method body*, the part of the method that does the actual work for the method when the method is called. The method body contains any *local variables* we need in the method. When we have variables or constants that we only need in a particular method, we declare those variables and constants at the beginning of the method body. These are called local variables (and constants) because they can only be used inside the method in which they're declared. Of course, the method body also contains the executable statements that we want the method to carry out.

There's actually a computer science term to describe where we can use a particular variable; it's called the variable's *scope*. Variables that we declare as fields in a class are usable throughout the class, while variables we declare within a method are only usable within the method. In fact, the loop control variable we declare in a for loop is only usable within the body of the for loop, and in a foreach loop the variable name we declare in the foreach part is only visible within the loop body.

Now, you'll really have two kinds of methods in the classes you define: *constructors* and all the other methods you need. Let's look at constructors first.

Constructors are used when you create a new object from this class; whenever we've created objects in our programs up to this point, we've been using the constructor to do so. For example, we should provide a constructor for any objects we want to create from the `Deck` class. The easiest kind of constructor to write is one that doesn't do anything other than create the object:

```
/// <summary>
/// Constructor
/// </summary>
public Deck()
{
}
```

You should notice a few things here. We've made our constructor `public` because we want consumers of the class to be able to create objects from this class; unless it's a static class, it's pretty useless if they can't create an instance of the class! We haven't included a return type for this method, despite our syntax description for methods above, because constructors are a special kind of method that doesn't require a return type. You can think of the constructor as returning a new object of the class, though. The method name HAS to be the same as the class name for constructors; that's how C# knows that it's a constructor rather than some other method. Finally, note that our parameter list above is empty. That means we don't have any arguments to pass when we call this constructor, though we can of course use parameters for constructors when we need them.

But we probably wouldn't want to have our constructor do nothing. Why not? Well, what if we created a new deck object, then shuffled the deck and took the top card? What would we get for the top card? We couldn't take the top card, because the deck still doesn't have any cards in it! We should definitely put the cards in the deck when we create the deck object. Using the code we used in Chapter 10 to use nested loops to fill a deck of cards, we can change our constructor to the following:

```
/// <summary>
/// Constructor
/// </summary>
public Deck()
{
    // fill the deck with cards
    foreach (Suit suit in Enum.GetValues(typeof(Suit)))
    {
        foreach (Rank rank in Enum.GetValues(typeof(Rank)))
        {
            cards.Add(new Card(rank, suit));
        }
    }
}
```

Recall that we already declared our `cards` variable as a field and created an empty list to hold the cards when we declared the variable. The code above puts all the cards we need into that list.

As you know, our class can have multiple constructors. Why do we think you already know this? Because we've already talked about method overloading, and since a constructor is just a special form of method we can overload constructors also. The compiler figures out which constructor we want to use based on the arguments that we provide in the call to the constructor.

You could argue that our constructor should also shuffle the deck, but have you ever noticed how the cards are arranged in a new (physical) deck of cards you open? They're actually arranged in order by suit and then rank – it's your job to shuffle the new deck before using it. We're following the same approach here.

One final comment about constructors before we move on. We've decided not to include the constructors in our console apps in our UML diagrams because we'll always want to write at least one constructor for each class we define, so there's no need to add them to our diagrams. For our Unity games, the default is that our scripts don't include constructors (remember, we use `Instantiate` instead of a constructor to create new game objects), so we don't include constructors in our UML diagrams for them either.

Well, we've spent a lot of time talking about constructors, but there are a whole lot of other methods too: all the other methods to do things with the class or the object. However, Figure 12.4. shows the code as it currently stands.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DeckExample
{
```

```

/// <summary>
/// A deck of cards
/// </summary>
public class Deck
{
    #region Fields

    List<Card> cards = new List<Card>();

    #endregion

    #region Constructors

    /// <summary>
    /// Constructor
    /// </summary>
    public Deck()
    {
        // fill the deck with cards
        foreach (Suit suit in Enum.GetValues(typeof(Suit)))
        {
            foreach (Rank rank in Enum.GetValues(typeof(Rank)))
            {
                cards.Add(new Card(rank, suit));
            }
        }
    }

    #endregion

    #region Properties

    /// <summary>
    /// Gets whether or not the deck is empty
    /// </summary>
    public bool Empty
    {
        get { return cards.Count == 0; }
    }

    #endregion
}
}

```

**Figure 12.4. Deck.cs**

Wait a minute, you say, after all that talk about making fields private why didn't we make our `cards` field `private`? As we said above, because it's such good coding practice to do this, our fields automatically default to being `private`; you'll have to explicitly provide a different access modifier to make them something else. Was that discussion a waste of time then? Not really, because you do need to understand how the access modifiers work so you can use them properly.

Okay, let's start working on our methods; recall that we have to implement the `Cut`, `Shuffle`, and `TakeTopCard` methods. Let's start with the easiest method – the `TakeTopCard` method. Here's the code:

```

/// <summary>
/// Takes the top card from the deck. If the deck is empty, returns null
/// </summary>
/// <returns>the top card</returns>
public Card TakeTopCard()
{
    if (!Empty)
    {
        Card topCard = cards[cards.Count - 1];
        cards.RemoveAt(cards.Count - 1);
        return topCard;
    }
    else
    {
        return null;
    }
}

```

First, we check to make sure the consumer of the class isn't trying to take the top card from an empty deck. If they are, we return a null card. As you can see, we're using the `Empty` property to check to see if the deck is empty. We'll actually talk about an alternative to returning null in this case when we get to the chapter that discusses exceptions.

If the deck isn't empty, we save the last card in the list – we're thinking of the end of the list as the top of the deck – into a local variable so we can return it from the method. We then remove the card from the list of cards in the deck, since after someone takes the card from the deck it's no longer in the deck. Then we return the card we saved to the code that called the method. Note that we use `return` to indicate what the method returns to the code that called the method just like we did in our `Empty` property.

You might be thinking that it would be more intuitive to think of the front of the list of cards as the top of the deck, since then we could use `0` in the above code instead of `cards.Count - 1` to access and remove the top card. The problem is that when we remove something from the front of the list, all the other elements in the list get shifted down one place. When we remove something from the end of the list, nothing else in the list needs to be moved. That may not seem like that big a deal to you, but think about it this way. If we use the front of the list as the top of the deck and remove the top cards one at a time until the deck is empty, we'll have to do  $51 * 50 * 49 * \dots * 1$  shifts (that's  $51!$ , called 51 factorial). If we use the back of the list as the top of the deck, we won't have to do any shifts. Why spend all those CPU cycles to do shifts when we don't have to?

Especially in game development, we always want to be careful to be as efficient as possible. This is one case where we can get much better efficiency by thinking carefully about how we approach our problem solution.

Okay, let's look at our first cut <grin> at the `Cut` method next:

```

/// <summary>
/// Cuts the deck of cards at the given location
/// </summary>
/// <param name="location">the location at which to cut the deck</param>
public void Cut(int location)
{

```

```

int cutIndex = cards.Count - location;
List<Card> newCards = new List<Card>(cards.Count);
for (int i = cutIndex; i < cards.Count; i++)
{
    newCards.Add(cards[i]);
}
for (int i = 0; i < cutIndex; i++)
{
    newCards.Add(cards[i]);
}
cards = newCards;
}

```

Before we look at the details in the method body, let's think of the problem this way. What we're really trying to do is move the stack of cards above the cut location to be below the stack of cards from the cut location to the bottom of the deck. Now we're getting somewhere – all we have to do is move the cards above the cut location to the bottom of the deck.

We pass in the location at which we want to cut the deck as an argument in the method call since that's information the method needs to do its job. We'll end up using lots of parameters for the methods we write, so let's talk a little bit more about how they actually work.

Let's say we call the `Cut` method as shown below:

```
myDeck.Cut(5);
```

What really happens inside the method? The `location` parameter is matched up with the 5 when the method is called, so any time `location` is referred to inside the method, it has the value 5. One of the great things about methods is that we can reuse them as many times as we want, especially by using different arguments when we call the method. So if we call the `Cut` method with this method call instead

```
myDeck.Cut(7);
```

we've reused the same code to cut the deck in a different place. Inside the method for this call, any time `location` is referred to inside the method, it has the value 7.

We've been calling methods throughout the book, so you already know that we can use variables or expressions as our arguments in the method calls. That means that both

```

int cutLocation = 13;
myDeck.Cut(cutLocation);

```

and

```
myDeck.Cut(7 + 8);
```

are also valid ways to call our `Cut` method.

Okay, back to the details of our method. Because the given location is interpreted as the number of cards down from the top of the deck but the top of the deck is at the end of the list in our implementation, we can't use the provided location directly. Instead, we calculate the `cutIndex` to give us the list index that

corresponds to the given cut location. The next thing we do is create a new list of cards that will hold the cards after they've been cut. We're using one of the overloaded constructors for the [List](#) class; with this constructor, we provide the capacity of the list. We know lists will actually grow in size as we need them to – that's one of the key benefits of lists over arrays – so why do we do that? It turns out that it actually costs extra CPU time to grow a list when we need to, so providing the capacity of the list when we construct it lets us avoid that CPU cost.

The next thing we do is copy the cards in the original list of cards from the cut index to the end of the list (e.g., the top of the deck) into the beginning part (e.g., the bottom of the deck) of the new list of cards. Then we copy the cards from the beginning of the original list of cards up to, but not including, the cut index into the end of the new list of cards, essentially putting the bottom part of the old deck of cards on top of the top part of the old deck of cards we've saved in the new list.

The last line of code sets our `cards` list to the `newCards` list we created and filled with the cut deck. That's what we want to do, of course, but what happens to the old list of cards that the `cards` variable referred to before this assignment? It gets garbage collected since nothing refers to it any more.

Before moving on to our final method, you should realize that there's a more intuitive way to implement the `Cut` method. To figure out how to implement it, we need to read the [List](#) documentation to see if the [List](#) class exposes any methods that could help us do this. Of course it does, so here's the revised method:

```
/// <summary>
/// Cuts the deck of cards at the given location
/// </summary>
/// <param name="location">the location at which to cut the deck</param>
public void Cut(int location)
{
    int cutIndex = cards.Count - location;
    Card[] newCards = new Card[cards.Count];
    cards.CopyTo(cutIndex, newCards, 0, location);
    cards.CopyTo(0, newCards, location, cutIndex);
    cards.Clear();
    cards.InsertRange(0, newCards);
}
```

You should look at the [List](#) documentation, specifically for the `CopyTo` and `InsertRange` methods, to make sure you understand how the revised method works.

How did we know to do it this way? By reading the documentation! We read the description for each of the [List](#) methods to see if there were any that could help us, and sure enough, there were. There are lots of people who believe that reading documentation is for n00bs and that real programmers just whack the code together as quickly as possible. That philosophy is dead wrong, though; we read documentation all the time to make sure we're writing clean, easy-to-understand code, and you should too.

On to our `Shuffle` method. Before we actually write the code, let's come up with an approach we could use to shuffle the deck. As you probably know, the point of shuffling a deck of cards is to randomize the order of the cards in the deck. One approach we could use would be to really try to simulate a shuffle, where we'd split the deck in half and interleave the cards from each half to re-form the deck. It's been mathematically shown that shuffling 7 times is reasonable to randomize the cards in a deck.

Alternatively, since our goal is to get the cards into a random order, we can come up with a different way to randomize the order of the cards.

Take a look at the following code. The general idea is to work our way through the list of cards backwards, swapping each card with the card at a random location from the beginning of the list up to the current location, inclusive. By the time we've worked our way through the entire list of cards, we have a very good randomization of the card order. Notice the reference in the comment for the method; this is the way Java implements a random shuffle of a list.

```
/// <summary>
/// Shuffles the deck
///
/// Reference:
/// http://download.oracle.com/javase/1.5.0/docs/api/java/util/
/// Collections.html#shuffle%28java.util.List%29
/// </summary>
public void Shuffle()
{
    Random rand = new Random();
    for (int i = cards.Count - 1; i > 0; i--)
    {
        int randomIndex = rand.Next(i + 1);
        Card tempCard = cards[i];
        cards[i] = cards[randomIndex];
        cards[randomIndex] = tempCard;
    }
}
```

And that finishes our work on the `Deck` class. Now let's apply what we've learned about class design to solve a complete problem.

If you download the code from the web site, you'll see that we actually added a `Print` method to the `Deck` class and wrote a small test program to test the constructor, property, and methods. You'll often find that your classes evolve, both to support unit testing those classes and because consumers of the class need additional class functionality.

## 12.5. Putting It All Together

Here's the problem description:

Design and implement a class to represent a single die. You also need to provide the standard die operations for the class.

### *Understand the Problem*

A couple questions immediately come to mind. First of all, how many sides should the die have? A standard die has 6 sides, but you can purchase dice of many different numbers of sides (especially if you want to roll to find out the damage the evil Orc has just inflicted <grin>). Let's make the default die a six-sided die, but also provide the capability to create a die of any integer number of sides.

We also need to ask what the standard die operations are. We should obviously be able to roll the die (presumably, randomly selecting which side is on top), and we'll also have to be able to find out which

side of the die is on top. Anything else? What about letting someone weight the die so some rolls are more common than others? You're kidding! We're not going to let anyone cheat that way! Really, rolling a die and seeing which side is on top are the only common die operations.

We will add one more operation, though, that tells how many sides the die has. In real life, you hardly ever need to remind yourself how many sides the die has (especially with a standard die!), but in code if the die has lots of sides you might need to remind yourself.

Okay, we're ready to move on to our design.

### *Design a Solution*

Because we're going to allow an arbitrary number of sides for a die, we'll need an instance variable in the `Die` class to "remember" how many sides the die has. Because the number of sides is an integer, we'll use an `int` for that variable. We're also going to need to remember which side is currently on top; an `int` will work for that variable as well.

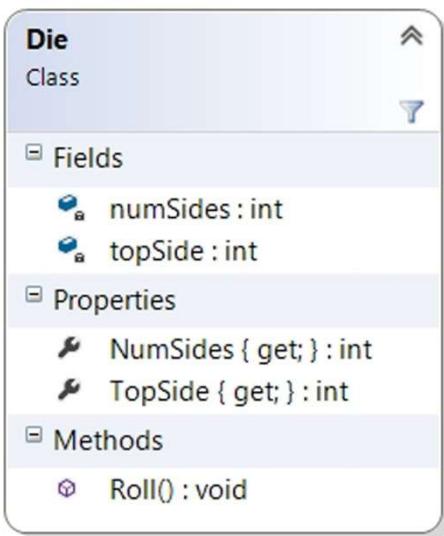
What about our properties and methods? Even though we don't include constructors in our UML diagram, we should probably realize that we need two constructors for the class. We'll have one constructor that doesn't have any parameters and sets the number of sides to 6, and we'll need another constructor that lets the user specify with an argument how many sides the die will have. This makes it easier for consumers of the class who are creating a standard die, while also providing the flexibility to create a die with an arbitrary number of sides.

So which operations should be properties and which should be methods? Remember, we typically use properties to provide access to the state of the object. That means we should expose a `TopSide` property to let a consumer find out which side is on top and a `NumSides` property to let the consumer find out how many sides the die has. You should immediately realize that both these properties should only provide read (not write) access; if not, stop and think about that for a moment.

The `TopSide` `get` accessor will return an `int` to tell the consumer which side of the die is on top. Similarly, the `NumSides` `get` accessor will return an `int` for the number of sides.

Because rolling the die is more complicated and really represents having the die do something (rather than directly accessing object state), we'll use a method to roll the die. The `Roll` method won't need any parameters, since all the information we need (the number of sides) is contained in the object. Should the method return anything? There's a tradeoff here. On one hand, rolling the die shouldn't return anything to the user, since it just changes the internal state of the die object. On the other hand, if we roll the die, we're probably interested in which side ends up on top, so we could return that information. We're going to design this so the `Roll` method doesn't return anything; a consumer of the class would therefore roll the die using the `Roll` method, then access the `TopSide` property to find out which side is on top. This is actually consistent with how rolling a die works in the real world – first we roll the die, then when it comes to rest we see which side is on top.

Given this design, we have the UML diagram shown in Figure 12.5.

**Figure 12.5. Die UML Diagram**

Since we've now finished our UML diagram, we're ready to move on to our test plan.

#### *Write Test Cases*

We need to make sure our test plan covers both constructors, the `TopSide` and `NumSides` properties, and the `Roll` method. We don't have any selection or iteration constructs in any of this, so we could do all the testing in a single test case, but let's do one test case for the default six-sided die and another for a 256-sided die. For each of the test cases, we'll get the top side and the number of sides right after creating the die, then roll the die 3 times and print the resulting top side of the die after each roll. These test cases are really unit test cases rather than functional test cases since we're testing a single class.

#### **Test Case 1**

##### **Checking 6-Sided Die**

Step 1. Input: None. Hard-coded steps in the test case code

Expected Result:

```

Top Side: 1
Num Sides: 6
Top Side: appropriate
Top Side: appropriate
Top Side: appropriate

```

Notice that we have a new problem here because we're trying to test a program that uses random numbers. How are we going to tell whether or not the top side is correct after each roll? This is actually a really hard problem to solve. At this point, we'll just confirm that the rolls "look random", though we recognize this is an imperfect approach. We'll discuss this further at the end of the chapter.

Here's the other test case:

**Test Case 2****Checking 256-Sided Die**

Step 1. Input: None. Hard-coded steps in the test case code

Expected Result:

```
Top Side: 1
Num Sides: 256
Top Side: appropriate
Top Side: appropriate
Top Side: appropriate
```

*Write the Code*

Let's write the constructor that doesn't have any parameters and the `TopSide` and `NumSides` properties and test those out before writing the `Roll` method. Our initial code is in Figure 12.6.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DieProblem
{
    /// <summary>
    /// A die
    /// </summary>
    public class Die
    {
        #region Fields

        int topSide;
        int numSides;

        #endregion

        #region Constructors

        /// <summary>
        /// Constructor for six-sided die
        /// </summary>
        public Die()
        {
            numSides = 6;
            topSide = 1;
        }

        #endregion

        #region Properties

        /// <summary>
        /// Gets the current top side of the die
        /// </summary>
        public int TopSide
        {
            get { return topSide; }
        }
    }
}
```

```

    }

    /// <summary>
    /// Gets the number of sides the die has
    /// </summary>
    public int NumSides
    {
        get { return numSides; }
    }

    #endregion
}

}

```

**Figure 12.6. Die.cs**

Although the initial value we select for `topSide` is really arbitrary, we should still pick an initial value to use in the constructor (`ints` actually default to 0 for their initial value, which would be invalid for a standard die); we just happened to pick 1.

### *Test the Code*

We need to write an application class for our test cases; we'll start by simply adding the portions of Test Case 1 that we can currently execute. Specifically, we won't call the `Roll` method in the test case yet because we haven't written it yet!

Assuming you created the project in the IDE as a console application, we can simply add our test cases to the `Program` class that was automatically generated by the IDE. Figure 12.7 show that class with Test Case 1 partially implemented.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DieProblem
{
    /// <summary>
    /// Tests the Die class
    /// </summary>
    class Program
    {
        /// <summary>
        /// Executes test cases for the Die class
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            #region Test Case 1

            Die die1 = new Die();
            Console.WriteLine("Test Case 1");
            Console.WriteLine("-----");

```

```

        Console.WriteLine("Top Side: " + die1.TopSide);
        Console.WriteLine("Num Sides: " + die1.NumSides);
        Console.WriteLine();

        #endregion

        Console.WriteLine();
    }
}
}
}

```

**Figure 12.7. Program.cs**

When we run the code, we get the output shown in Figure 12.8, which is exactly what we expected.

**Figure 12.8. Test Case 1 Partial Output**

### *Write the Code, Part 2*

Next, we add the constructor that takes in the number of sides as a parameter; that code is provided below.

```

/// <summary>
/// Constructor for a die with the given number of sides
/// </summary>
/// <param name="numSides">the number of sides</param>
public Die(int numSides)
{
    this.numSides = numSides;
    topSide = 1;
}

```

There's actually something here that we only discussed briefly earlier in the book. You might think that we could just use

```
numSides = numSides;
```

to set the instance variable to the value of the parameter. Although this is syntactically correct, it doesn't actually do what you hoped it would. In fact, it sets the parameter's value to the parameter's value, which is obviously a waste of time.

So how do we say we want to set the value of an instance variable with the value of a parameter that has the same name? By preceding the instance variable's name with the `this` keyword like this:

```
this.numSides = numSides;
```

The problem is that the parameter named `numSides` "hides" the instance variable named `numSides` inside the constructor. By preceding the instance variable's name with `this`, we're indicating that we mean the instance variable for this object, not the parameter. We won't have to use the `this` keyword very often – most commonly in our constructors – but it's important that you understand how it works.

### *Test the Code, Part 2*

Now we can add code to our `Program` class to run the first part of Test Case 2. When we do that and run the program, we get the output shown in Figure 12.9.



```
C:\WINDOWS\system32\cmd.exe
Test Case 1
-----
Top Side: 1
Num Sides: 6

Test Case 2
-----
Top Side: 1
Num Sides: 256

Press any key to continue . . .
```

**Figure 12.9. Test Cases 1 and 2 Partial Output**

Our code seems to be working fine so far, but let's work on the constructors a little more before moving on.

### *Write the Code, Part 3*

Here's what our 2 constructors look like:

```
/// <summary>
/// Constructor for six-sided die
/// </summary>
public Die()
{
    numSides = 6;
    topSide = 1;
}
```

```

/// <summary>
/// Constructor for a die with the given number of sides
/// </summary>
/// <param name="numSides">the number of sides</param>
public Die(int numSides)
{
    this.numSides = numSides;
    topSide = 1;
}

```

This isn't very satisfying, because the code in the constructors looks remarkably similar (setting the `topSide` field is even identical). We should be able to figure out a way to consolidate the constructor logic in one place and reuse it for both the six-sided die and general die constructors. After all, the ability to reuse methods – including constructors – is one of the reasons we use methods in the first place!

We can, in fact, do this in a better way. Check out the new constructors:

```

/// <summary>
/// Constructor for six-sided die
/// </summary>
public Die(): this(6)
{

/// <summary>
/// Constructor for a die with the given number of sides
/// </summary>
/// <param name="numSides">the number of sides</param>
public Die(int numSides)
{
    this.numSides = numSides;
    topSide = 1;
}

```

How does the constructor for the default six-sided die work now? It actually calls the other constructor with 6 as the argument, creating a die with 6 sides just as we want. Remember, we've used `this` before to reference our instance variables; our modification simply shows another use of `this` to refer to (call) another constructor for the class. Why is this a good idea? Because we've shortened the code and put all the initialization that's done at instantiation time into a single constructor. If we ever need to change that code, we can do it in one place rather than in multiple places.

### Test the Code, Part 3

Before moving on, we need to re-test our code to make sure we haven't broken anything by changing the constructor for the six-sided die. Testing code after you've made internal changes that shouldn't be visible to consumers of the code is typically called *regression testing* because we're making sure our code hasn't regressed – moved backward to a worse condition – as a result of our changes. Running our test cases again shows that the code still works as expected, so we can move on.

*Write the Code, Part 4*

Finally, we add the `Roll` method, which we can implement as follows:

```
/// <summary>
/// Rolls the die
/// </summary>
public void Roll()
{
    Random rand = new Random();
    topSide = rand.Next(numSides) + 1;
}
```

The first line of code creates a new `Random` object we can use to generate random numbers. The second line of code generates a random number between 0 and `numSides` - 1, then adds 1 to that number to get the correct number for a face on the die. We don't want a six-sided die to have its sides numbered from 0 to 5, do we? That's why we need to add 1 to the random number – to shift the numbers so that the sides are actually 1 to 6 as we'd expect.

That completes our `Die` class, so let's move on to our complete test cases.

*Test the Code, Part 4*

We added code to our `Program` class to roll the die 3 times in each of our test cases and print the results; the output is shown in Figure 12.10.

```
C:\WINDOWS\system32\cmd.exe
Test Case 1
-----
Top Side: 1
Num Sides: 6
Top Side After Roll: 1
Top Side After Roll: 1
Top Side After Roll: 1

Test Case 2
-----
Top Side: 1
Num Sides: 256
Top Side After Roll: 22
Top Side After Roll: 22
Top Side After Roll: 22

Press any key to continue . . .
```

**Figure 12.10. Test Cases 1 and 2 Output**

Houston, we have a problem! It sure doesn't look like the die is getting rolled randomly. We better go back to the code (again).

*Write the Code, Part 5*

What's going on here? To find the answer, we need to look at the documentation for the `Random` class. Our first step is to examine a little more closely how the `Random` constructor that we're using works; the documentation says that the constructor

"Initializes a new instance of the `Random` class, using a time-dependent default seed value." and, in the Remarks section, "The default seed value is derived from the system clock and has finite resolution."

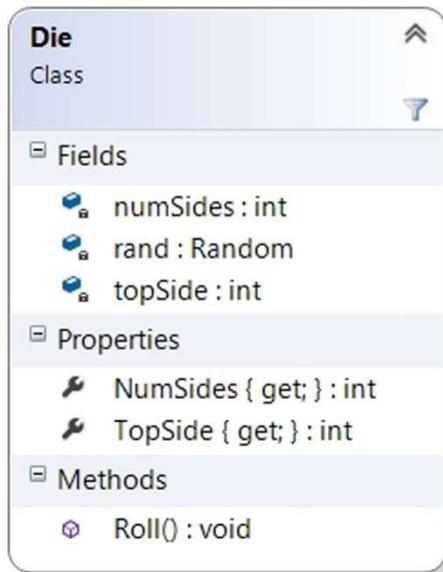
This is actually fairly common. Random number generators have some base number that they use to generate the sequence of numbers; this base number is called a *seed*. Using the current system time is also a common approach, but why are we getting the same number every time?

If we keep reading the Remarks section of documentation we find the following statement:

"As a result, different `Random` objects that are created in close succession by a call to the default constructor will have identical default seed values and, therefore, will produce identical sets of random numbers. This problem can be avoided by using a single `Random` object to generate all random numbers."

It feels like we're getting closer. We're creating a new instance of `Random` each time we call the `Roll` method; because computers are screamingly fast these days, the documentation tells us that we're probably using the same seed each time. Using the same seed yields the same sequence of numbers, so we've found our problem. Now all we have to do is fix it!

One solution (the one suggested by the documentation, and therefore the one we'll use here) is to use a single random number generator as an instance variable rather than creating a new random number generator each time we call the `Roll` method. We'll create that object when we declare the variable that holds it. Our new UML diagram is shown in Figure 12.11.



**Figure 12.11. Revised Die UML Diagram**

The complete (and revised) `Die` class code is provided in Figure 12.12.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
  
```

```
namespace DieProblem
{
    /// <summary>
    /// A die
    /// </summary>
    public class Die
    {
        #region Fields

        int topside;
        int numSides;
        Random rand = new Random();

        #endregion

        #region Constructors

        /// <summary>
        /// Constructor for six-sided die
        /// </summary>
        public Die(): this(6)
        {
        }

        /// <summary>
        /// Constructor for a die with the given number of sides
        /// </summary>
        /// <param name="numSides">the number of sides</param>
        public Die(int numSides)
        {
            this.numSides = numSides;
            topside = 1;
        }

        #endregion

        #region Properties

        /// <summary>
        /// Gets the current top side of the die
        /// </summary>
        public int TopSide
        {
            get { return topside; }
        }

        /// <summary>
        /// Gets the number of sides the die has
        /// </summary>
        public int NumSides
        {
            get { return numSides; }
        }

        #endregion
    }
}
```

```

#region Public methods

/// <summary>
/// Rolls the die
/// </summary>
public void Roll()
{
    topSide = rand.Next(numSides) + 1;
}

#endregion
}
}

```

**Figure 12.12. Die.cs***Test the Code, Part 5*

Now we run our test cases again; the results can be found in Figure 12.13.

```

C:\WINDOWS\system32\cmd.exe
Test Case 1
-----
Top Side: 1
Num Sides: 6
Top Side After Roll: 2
Top Side After Roll: 1
Top Side After Roll: 3

Test Case 2
-----
Top Side: 1
Num Sides: 256
Top Side After Roll: 170
Top Side After Roll: 183
Top Side After Roll: 133

Press any key to continue . . .

```

**Figure 12.13. Test Cases 1 and 2 Output**

Whew, it looks like we solved our problem! We should run the test cases a few more times, though, to make sure we don't keep getting the same sequence of numbers (we don't).

There's actually a reasonable way around our informal "see if the rolls look random" approach to testing here. As discussed above, the `Random` class uses a seed as the starting point for the sequence of random numbers it generates. If we create the `Random` class with the same seed every time we'll get the same sequence of random numbers; that's what caused our problem above. To be precise, we actually get pseudo-random numbers in the sequence – if the sequence were truly random, we wouldn't be able to predict the next number in the sequence, but if we know the sequence of numbers previously generated using the same seed we can perfectly predict the numbers in the sequence. Why do we care in this case? Because we could run the sequence once with a given seed to see what the sequence of numbers is, then in our test case code if we use the same seed we'll know exactly what the roll results should be. We won't bother with that approach here, but we wanted you to be aware of it.

## 12.6. Common Mistakes

### *Trying to Return Something from a Constructor*

It might seem intuitive that you should return something from a constructor. We are creating a new object, after all! Constructors do not, however, have return types (actually, their return type is `void`, which is implicit). If we add a return type to our constructors, the compiler views them as “normal” methods rather than constructors, and they won’t be called when you try to create an instance of the class.

### *Forgetting to Use this When Assigning to an Instance Variable*

This is really only an issue when the instance variable has the same name as a parameter, but that happens a lot, especially with constructors. If you forget the `this` the code will compile and run, it just won’t do what you want it to do!

### *Trying to Access an Instance Variable from a Static Method*

Remember, static methods are only allowed to access static variables, not instance variables. If you find you really need to do this, you need to change your class design to either provide a static variable to hold the information you need or change the static method to an instance method.

# Chapter 13. More Class Design and Methods

In Chapter 12, we started covering the basics of class design. We discussed high-level class design and the instance and static variables we can use in a class, we discussed how properties are designed and implemented, and we also provided some introductory material about methods. In this chapter, we look at methods in much greater detail.

## 13.1. A Brief Review

We've already either completed or generated a number of methods in previous chapters, but one of the questions we didn't really consider very closely is how we actually decide which methods we should use.

In Chapter 12 we said that classes have constructors and other methods. Recall that our constructors don't provide a return type, but all our other methods do. If the method doesn't return anything we set the return type to `void`, but if the method does need to return a value to the code that called the method we set the return type to the data type of that value.

The methods we wrote in the previous chapter were all `public` since we wanted the consumers of the classes to be able to use those methods. It's often the case, however, that it also makes sense to have `private` methods that do internal processing for the class. Private methods are useful inside the classes that contain them, but there's no reason for a consumer of the class to call those methods directly. That's why we make them `private`.

## 13.2. Deciding Which Methods to Use

One of our key jobs in the Design a Solution step of our process is developing the UML diagram for each of the classes in our solution. That UML diagram for a class includes a variety of important information, including information about the methods the class exposes to consumers of the class.

So how do we decide which methods to provide? If we were designing a system of interacting classes, there are very useful Object-Oriented Analysis and Design techniques we can use to determine the required methods. However, we'll continue using a more informal process in this book.

Specifically, we'll decide which methods to include in a class by thinking about the behavior that class needs to provide. One of the things that really helps us do this is the realization that many of the classes we design are actually used to model entities in the real world. Given that observation, we can think about how the real world entity behaves and decide which of those behaviors will be useful in the software class. We used this approach for our `Deck` class in the previous chapter, exposing `Shuffle`, `Cut`, and `TakeTopCard` methods because shuffling, cutting, and taking the top card are things we do with real decks of cards. That same approach led us to provide a `Roll` method for our `Die` class.

We'll also regularly find that we need `private` methods to help with the internal processing required by the class. We generally won't realize we need those methods when we think about the behavior of the real world object we're modeling; instead, we'll usually discover them when we Write the Code.

### 13.3. Figuring Out Information Flow

Once we've decided which methods we're going to write, we need to decide what information flows in to and out of each method. This really isn't as hard as it sounds! We've actually been doing this for a long time, although we've been discussing it informally. Any time we've talked about what information needs to come in to a method through parameters or what information needs to be returned by the method, we've been discussing information flow.

Why do we need to figure out the information flow for our methods anyway? Because we're going to need to use parameters for any information that comes into the method and we're going to need to return whatever information needs to come out of the method<sup>26</sup>. Before we can actually write our methods, we have to know what parameters and return types we need. Let's revisit the methods from our `Deck` class.

The constructor doesn't require any information to come in, since it just builds a standard deck of 52 cards with 4 suits and 13 ranks within each suit. We could come up with a more general constructor that lets the code calling the method specify how many suits and ranks there should be, but we'd then also need a new `Card` class that set the ranks and suits properly. We'll leave the constructor with no information coming in through parameters. Like all constructors, this method returns an object of the class.

How about the `shuffle` method? We don't need any information to come in to the method because the method just randomly shuffles the cards. We also don't pass any information out of the method, so this method doesn't have any information flow at all. No parameters, and the return type is `void`.

The `cut` method does need information to be passed in through a parameter; specifically, the method needs to know the location at which to cut the deck. It doesn't pass anything out of the method, though; it simply changes the internal state of the deck by cutting it, so the return type should be `void`.

Finally, the `TakeTopCard` method doesn't need any information coming in to the method because the deck knows the code calling the method is trying to remove the top card from the deck (rather than some other card). It does need to return that card to the code calling the method, though. So there are no parameters for this method, but the return type for the method is `Card`.

This figuring out the information flow stuff isn't new; in fact, we already had the above discussion in Chapter 12 as we were developing our class diagram for the `Deck` class. We included it again here because it's an important step we take when we design the methods in our class.

### 13.4. Creating the Method Header

Now that we've decided what methods to use and what information flows in to and out of each one, we're ready to develop our method headers. A method header is simply the part of our method that gives the access modifier, the name of the method, and details about information flow in to and out of the method. We already discussed method headers in Chapter 12.

The method header for a method without any information flow is pretty straightforward – here's the header for the `Shuffle` method:

---

<sup>26</sup> In fact, if our parameters are objects we can also return information through those parameters. We'll address this when we talk about passing objects as parameters.

```
public void Shuffle()
```

Methods that do have information flow require more complicated method headers. For example, the `Cut` method has a method header that looks like

```
public void Cut(int location)
```

Because this method needs the location coming in to the method as an integer, we pass that information with an `int` parameter. Whenever we have information flow into a method, we use a parameter for each piece of information (in this case, one parameter for the location at which we should cut the deck).

Each parameter has a data type and a name. Both the name and data type for a parameter shouldn't really require explanation, since they're very similar to the names and data types of variables and constants.

The `TakeTopCard` method doesn't need any information to come in to the method, but it does return the top card from the method. That method header looks like

```
public Card TakeTopCard()
```

And that takes care of all the methods we have for the `Deck` class.

But how do we know how many parameters each method should have? And how do we know whether or not the method needs to return a value? Believe it or not, we've already solved that problem! When we did our information flow analysis, we captured the answer in the characteristics of each method in our class diagram. Look again at the method headers we just generated, and you'll see how we easily went from our class diagram to our method header. Don't skip the class diagram because you think your problem solving will go faster without it; you're going to have to figure out the method headers no matter what, and the class diagram is a great tool to help you do that.

Now that we have the method headers figured out, let's revisit the method bodies.

### 13.5. The Method Body

So far in this chapter, we've really only defined how other code will interact with each of our methods; we still need to write the rest of the code for each method, which is called the *method body*. Let's review the method bodies for each of the methods in the `Deck` class.

Here's the code for the `TakeTopCard` method:

```
/// <summary>
/// Takes the top card from the deck. If the deck is empty, returns null
/// </summary>
/// <returns>the top card</returns>
public Card TakeTopCard()
{
    if (!Empty)
    {
        Card topCard = cards[cards.Count - 1];
        cards.RemoveAt(cards.Count - 1);
        return topCard;
    }
}
```

```
        else
    {
        return null;
    }
}
```

First, we check to make sure the consumer of the class isn't trying to take the top card from an empty deck. If they are, we return null because there is no top card. If the deck isn't empty, we save the last card in the list – we're thinking of the end of the list as the top of the deck – into a local variable so we can return it from the method. We then remove the card from the list of cards in the deck, since after someone takes the card from the deck it's no longer in the deck. Then we return the card we saved to the code that called the method.

For the Cut method, we have:

```
/// <summary>
/// Cuts the deck of cards at the given location
/// </summary>
/// <param name="location">the location at which to cut the deck</param>
public void Cut(int location)
{
    int cutIndex = cards.Count - location;
    Card[] newCards = new Card[cards.Count];
    cards.CopyTo(cutIndex, newCards, 0, location);
    cards.CopyTo(0, newCards, location, cutIndex);
    cards.Clear();
    cards.InsertRange(0, newCards);
}
```

From the [List](#) documentation, we find that the `CopyTo` method “Copies a range of elements from the list to a compatible one-dimensional array, starting at the specified index of the target array.” In other words, our first call to the `CopyTo` method copies all the cards in the deck from the cut index to the end (top) of the deck into the beginning of the `newCards` array. The first argument in the method call tells where to start copying from and the last argument in the method call tells the number of elements to copy. The second argument is the target array and the third argument is the starting index in the target array. Our second call to the `CopyTo` method copies all the cards in the deck from the beginning (bottom) of the deck up to the cut index into the `newCards` array starting just after the cards we already added to that array. The last line of code in the method body simply copies all the cards in the array – which now contains the cut deck – into our `cards` field, replacing the cards that used to be in that field.

And finally, our `shuffle` method:

```
/// <summary>
/// Shuffles the deck
///
/// Reference:
/// http://download.oracle.com/javase/1.5.0/docs/api/java/util/
/// Collections.html#shuffle%28java.util.List%29
/// </summary>
public void Shuffle()
{
    Random rand = new Random();
    for (int i = cards.Count - 1; i > 0; i--)
```

```

    {
        int randomIndex = rand.Next(i + 1);
        Card tempCard = cards[i];
        cards[i] = cards[randomIndex];
        cards[randomIndex] = tempCard;
    }
}

```

There's nothing particularly complicated in this method body, though it is an interesting way to randomize the order of the elements in the collection. That concludes our review of the methods in the `Deck` class.

## 13.6. Parameters and How They Work

Although we've been using parameters and arguments for some time, and we talked about them in previous chapters, parameters seem to be a very difficult concept for programmers learning about methods to really nail down. This section takes a closer look at how they work.

Perhaps the easiest way to understand how parameters work is to think of them this way: whenever we associate an argument in the method call with a parameter in the method header, the value of that argument is copied into a temporary variable with the parameter name inside the method. This approach is called *pass by value*. Note that this only applies to passing value types as parameters; we'll talk about passing objects as parameters in the following section.

Let's illustrate this idea by looking at a code fragment that simply cuts a deck in two different locations using the `Cut` method. Here's the code:

```

// set cut locations
int firstCutLocation = 26;
int secondCutLocation = 17;

// cut the deck twice
deck.Cut(firstCutLocation);
deck.Cut(secondCutLocation);

```

Recall that the `Cut` method has a single parameter called `location`. The first time we call the method, any time `location` is referenced inside the method, we're really talking about the value of the `firstCutLocation` variable. Why? Because in the method call, we said the parameter `location` corresponds to the argument `firstCutLocation` (remember, it matches number, order, and type for parameters), so the value of `firstCutLocation` gets copied into `location`. After the method finishes, the deck will have been cut at the location given in `firstCutLocation`. Now we call the method again, this time associating the `location` parameter with the `secondCutLocation` argument. That means that the value of `secondCutLocation` gets copied into `location`. After the method completes this time, the deck will have been cut at the location given in `secondCutLocation`.

We typically write methods to complete general kinds of actions on objects (e.g., cutting a deck). We can then reuse these methods as many times as we want, and by using different arguments and different variables on the left of the `=` sign each time (for methods that return something), we can make those methods act on as many different values as we want. The ability to use the same action (method) many

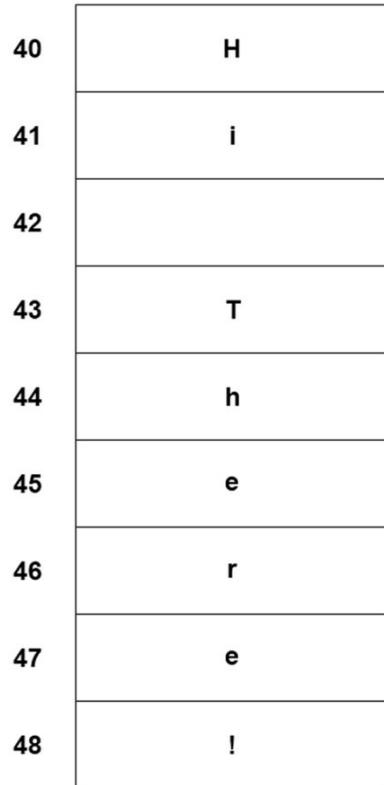
times is one of the things that makes methods so useful, and parameters are an essential part of making this reuse possible.

### 13.7. Passing Objects as Parameters

In the previous section, our discussion of parameter passing was limited to passing value types as parameters. But there's another critical "thing" we might pass as a parameter – you guessed it, an object. We can't really pass an object by its value, because the compiler needs to "set aside space" for parameters that are passed by value, and the compiler can't know before we run the program exactly how big the object will be as we run the program. For example, think of passing a `StringBuilder` object into a method. How many characters long will the `StringBuilder` be when we pass it into the method? In most cases, we don't know, so we can't pass the entire object as a value when we're passing objects as parameters.

So how does C# get around this problem? It turns out that, for objects, C# doesn't actually pass the entire object as a value – it passes a reference to the object as the value instead. Think of this as passing the memory address of the object rather than the actual object.

For example, say we have a `StringBuilder` object called `message`, with the value `"Hi There!"` that's stored in memory starting in memory location 40 as shown in Figure 13.1.



**Figure 13.1. StringBuilder Object in Memory**

When the `StringBuilder` is passed as an argument, it's actually the address of the `StringBuilder` object (40) that's passed rather than the object itself. The compiler can handle this without any trouble, because it simply needs to set aside enough space for an address rather than for the object itself.

We get one additional benefit when we pass objects as arguments to a method. Within the method, we can change the contents of the object (changing the contents of our `StringBuilder` object, for example) without changing the address, so we can actually change a parameter within a method if it's an object. It's kind of like changing the furniture in a house; the contents of the house change, but the address stays the same. Say we had the following method (comments are omitted for the sake of brevity), which simply replaces all occurrences of '`e`' in the string builder object with '`o`' instead:

```
void ChangeString(StringBuilder theString)
{
    theString.Replace('e', 'o');
}
```

We pass our `StringBuilder` object in as an argument using

```
ChangeString(message);
```

and after the method runs, our `message` will now be "`Hi Thoro!`". That's because we passed the address of the object as an argument rather than the object itself.

When we change an object that was passed as an argument to a method inside that method, this change is called a *side effect*. It's called a side effect because the fact that the object might be changed inside the method isn't obvious from the return type of the method or the list of parameters. There are absolutely many times when we want to change object parameters in a method, but be careful when you do this. Errors caused by side effects are very difficult to find, so whenever you implement changes to object parameters inside your method, do so with care.<sup>27</sup>

We said earlier that we could use literals rather than variables for arguments that were value types. We can't do that for objects, of course, because there's no way to just use a literal value for an object (unless we use `null`, which isn't usually useful). We can, however, create a new object for the argument if we don't happen to have an object already created. For example, in the constructor for our `Deck` class, we used

```
cards.Add(new Card(rank, suit));
```

to create new cards as arguments for the `Add` method in the `List` class.

Remember, when we pass objects as arguments, we really only pass a reference to the object rather than the object itself. This also lets us change the object inside the method.

### 13.8. Putting It All Together

Let's continue building our class design expertise by designing and implementing a complete class following our problem solving steps. Here's the problem description:

Design and implement a class to represent a hand of cards. You also need to provide the standard hand operations for the class.

---

<sup>27</sup> There are ways to mark the parameters that could be changed through side effects to make it clearer that those side effects may occur, but we don't need them for the problems in this book.

### Understand the Problem

We seem to be seeing a trend in the problem descriptions: they keep saying we have to provide standard operations for the class, and we have to figure out what those are! In lots of cases, we'd actually get a better description of what we need to provide for the class, but this gives us good practice really analyzing what's required.

We're obviously going to have to create `Hand` objects. We'll also need to provide some capability to access the cards in the hand. Why would we want to do that? Because in reality, people typically look at the cards in their hand to decide what to do. What else will we need? We're going to want to be able to add a card to a hand because a card might be dealt into the hand or the player could draw a card into the hand. We'll want the ability to remove a card from the hand, because players either play or discard cards from their hand. Finally, we'll want to be able to tell whether or not the hand is empty.

### Design a Solution

We're definitely going to need an instance variable in the `Hand` class to store the cards that are currently in the hand; just as for the `Deck` class, a `List` will be quite useful for this variable.

Are there any properties that our hand should provide? What kinds of information would a consumer of the `Hand` class want to know about the current state of a hand? At the very least, they'd probably want to know how many cards are in the hand. Let's therefore provide a `Count` property that provides that information.<sup>28</sup> Because consumers of the class shouldn't be able to directly change the size of the hand through the property, we'll provide read (not write or read-write) access for this property.

We also said that a consumer should be able to tell whether or not the hand is empty. We know the consumer could compare the `Count` property to 0 to determine whether or not the hand is empty, but as a convenience we'll also provide an `Empty` property that returns `true` if the hand is empty and `false` otherwise. We'll only provide read access for this property.

What about our methods? Our constructor won't need any parameters, because we'll simply have it create a hand with no cards in it. We could also provide another constructor that takes in a list of cards as a parameter and adds all those cards to the hand, but let's not do that. It's more intuitive to think of a hand as starting empty, then we add one card at a time to build up the hand.

Providing access to each of the cards in the hand could be a little tricky. There's a very common pattern that people use to "iterate" over things like a hand of cards; not surprisingly, it's called an iterator. Providing an iterator would really require that we implement something called an *interface* in C#, though, and implementing interfaces is a topic that we won't cover in this book.

Instead, let's provide a method that will let someone using the `Hand` class access each of the cards in the hand. This method, called `GetCard`, will let the user access a particular card in the hand (without removing it from the hand). The code calling this method will need to provide the location of the card to get as an `int` (starting with 0), and the method will return the card at that location.

---

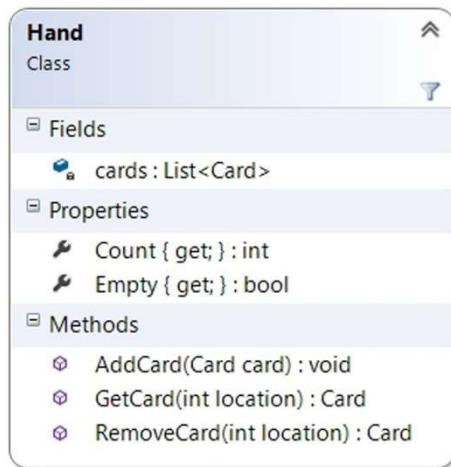
<sup>28</sup> You might think a property called `Size` would be better for the number of cards in the hand. Because the collection classes expose the `Count` property to tell how many elements are in the collection, we opt to use the more standard name here.

Providing this method (and the `Count` property) will make it very easy for someone to look at the cards in the hand using a for loop. They can simply set up the loop to go from 0 to the size of the hand, then get each of the cards inside the loop. By the way, do we need to add a new instance variable to store the size of the hand? No! The list holding the cards in the hand gives us access to the `Count` property of that list, so that list implicitly keeps track of the size of the hand for us. Cool.

We need to able to add a card to the hand, so let's figure out the information flow for an `AddCard` method. We'll need a single piece of information – the card to be added to the hand – as a parameter for this method. The method doesn't return anything, so we'll pick a return type of `void` for the method.

Finally, we need to be able to remove a card from the hand. Our `RemoveCard` method will need a single parameter telling the location of the card we want to remove. Like the `TakeTopCard` method in the `Deck` class, this method will also return a value: the card being removed.

We now have the class diagram shown in Figure 13.2.



**Figure 13.2. Hand Class Diagram**

Now that we have our design completed, we're ready to move on to our test plan.

### *Write Test Cases*

We need to make sure our test plan covers all the methods in our `Hand` class. The `GetCard` and `RemoveCard` methods will both have selection constructs (making sure a valid location has been provided), and we'll need to make sure we test both branches in those methods. We could do all our testing in a single test case, but let's build a number of test cases that we can use to test our class as we build it. Here's a reasonable set of test cases:

#### **Test Case 1**

##### **Checking Constructor and Empty and Count Properties**

Step 1. Input: None. Hard-coded steps in the test case code

Expected Result:

The hand is empty

Size: 0

This test case simply checks that the object is created properly and that the `Empty` and `Count` properties work properly when the hand is empty.

### Test Case 2

#### **Checking AddCard and GetCard methods and Empty and Count Properties**

##### **Branch: true branch in GetCard method**

Step 1. Input: None. Hard-coded steps in the test case code

Expected Result:

Ace of Spades  
Queen of Hearts  
The hand is not empty  
Size: 2

This test case makes sure the `AddCard` and `GetCard` methods work properly by adding an Ace of Spades and Queen of Hearts to the hand and that the `Empty` and `Count` properties work properly when the hand isn't empty.

### Test Case 3

#### **Checking GetCard method**

##### **Branch: false branch in GetCard method**

Step 1. Input: None. Hard-coded steps in the test case code

Expected Result:

Location for getting a card was invalid

This test case makes sure the `GetCard` method returns `null` when we try to get a card at location 0 from an empty hand.

### Test Case 4

#### **Checking RemoveCard method**

##### **Branches: true branch in RemoveCard method, false branch in RemoveCard method**

Step 1. Input: None. Hard-coded steps in the test case code

Expected Result:

Jack of Clubs  
Ace of Spades  
Queen of Hearts  
Location for removing a card was invalid

This test case makes sure the `RemoveCard` method works properly with both valid and invalid locations. We add an Ace of Spades, Queen of Hearts, and Jack of Clubs to the hand, then remove the card at location 2, then location 0, then location 0, then location 0 (which tries to remove a card from an empty hand).

### *Write the Code*

First, we'll add our single field to the class:

```
List<Card> cards = new List<Card>();
```

Next, we'll write the constructor:

```
/// <summary>
/// Constructor
/// </summary>
public Hand()
{
}
```

Our constructor doesn't actually do anything, but we know we need a constructor for the classes we write<sup>29</sup>. It actually turns out that if we don't provide a constructor for a class, C# will automatically generate a no-parameter constructor that doesn't do anything.

So why did we explicitly write a no-parameter constructor that doesn't do anything if C# will automatically provide one? This is really a matter of taste; some programmers like to do this and others don't. We like to make the constructor explicit in the code, though, because we believe it makes the code easier to understand.

Let's write our properties next, because then we can run Test Case 1 to see how we're doing with our implementation. The `Count` property simply returns the number of elements in our `cards` list:

```
/// <summary>
/// Gets the number of cards in the hand
/// </summary>
public int Count
{
    get { return cards.Count; }
}
```

and the `Empty` property tells whether or not the hand contains any cards (the same way we did in the `Deck` class):

```
/// <summary>
/// Gets whether or not the hand is empty
/// </summary>
public bool Empty
{
    get { return cards.Count == 0; }
}
```

### *Test the Code*

Now we'll add driver code for Test Case 1 to the `Program` class in our project; the code is provided in Figure 13.3.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

---

<sup>29</sup> We'll actually sometimes write `static` classes that don't have a constructor, but those are the rare exception, not the rule.

```

namespace HandProblem
{
    /// <summary>
    /// Tests the Hand class
    /// </summary>
    class Program
    {
        /// <summary>
        /// Tests the Hand class
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            Hand hand;

            // Test Case 1
            Console.WriteLine("Test Case 1");
            Console.WriteLine("-----");
            hand = new Hand();
            if (hand.Empty)
            {
                Console.WriteLine("The hand is empty");
            }
            else
            {
                Console.WriteLine("The hand is not empty");
            }
            Console.WriteLine("Size: " + hand.Count);
            Console.WriteLine();
        }
    }
}

```

**Figure 13.3. Test Case 1 Code**

When we run this code, we get the results shown in Figure 13.4.

**Figure 13.4. Test Case 1 Results**

The results are as we expected, so let's move on.

## Write the Code, Part 2

Here's the code for the `AddCard` method:

```
/// <summary>
/// Adds the given card to the hand
/// </summary>
/// <param name="card">the card to add</param>
public void AddCard(Card card)
{
    cards.Add(card);
}
```

This is a very straightforward method that just adds the given card to the list of cards in the hand, so we can proceed to the next method.

For the `GetCard` method, we need to handle the case where the code calling the method tries to get a card at an invalid location: one that's less than 0 or one that's past the location of the last card in the hand. In that case, the method returns `null`. This leads to the following code:

```
/// <summary>
/// Gets the card at the given location (leaving the card in the hand)
/// </summary>
/// <param name="location">the 0-based location of the card</param>
/// <returns>the card or null if the location is invalid</returns>
public Card GetCard(int location)
{
    // check for valid location
    if (location >= 0 && location < cards.Count)
    {
        return cards[location];
    }
    else
    {
        // invalid location
        return null;
    }
}
```

## Test the Code, Part 2

Now that we have the `AddCard` and `GetCard` methods implemented, we can add Test Cases 2 and 3 to our test driver code and run those test cases. Here's the code we add for Test Case 2:

```
// Test Case 2
Console.WriteLine("Test Case 2");
Console.WriteLine("-----");
hand = new Hand();
hand.AddCard(new Card(Rank.Ace, Suit.Spades));
hand.AddCard(new Card(Rank.Queen, Suit.Hearts));
for (int i = 0; i < hand.Count; i++)
{
    hand.GetCard(i).Print();
}
```

```
PrintEmpty(hand);
Console.WriteLine("Size: " + hand.Count);
Console.WriteLine();
```

A couple comments before we look at the code for Test Case 3. We used a for loop to print each of the cards in the hand. As we claimed when we were designing the class, the `GetCard` method and `Count` property let us easily access each card in the hand.

You should also notice that we're now calling a `PrintEmpty` method to print whether or not the hand is empty. When we only had a single test case, we used the following code for that:

```
if (hand.Empty)
{
    Console.WriteLine("The hand is empty");
}
else
{
    Console.WriteLine("The hand is not empty");
}
```

When we started writing the driver code for Test Case 2, we realized we needed the exact same code for this test case. One of the great benefits of methods is that they let us reuse code, and that benefit applies to test driver code as well as the code we'd actually deliver for our game. So, we wrote the following method in our `Program` class:

```
/// <summary>
/// Prints whether or not the given hand is empty
/// </summary>
/// <param name="hand">the hand to check</param>
static void PrintEmpty(Hand hand)
{
    if (hand.Empty)
    {
        Console.WriteLine("The hand is empty");
    }
    else
    {
        Console.WriteLine("The hand is not empty");
    }
}
```

We can now use this method for any of the test cases that need it without duplicating the code (notice that we had to make this method `static` so our `Main` method, which is `static`, could call it). Our driver code for Test Case 3 is:

```
// Test Case 3
Console.WriteLine("Test Case 3");
Console.WriteLine("-----");
hand = new Hand();
Card card = hand.GetCard(0);
PrintLocationValidity(card, "getting a card");
Console.WriteLine();
```

Notice that we're using a `PrintLocationValidity` method to print a message about the location. Take a look at the code for this chapter to see how that method is implemented if you're interested.

When we run all our test cases up to this point, we get the results shown in Figure 13.5.

```
C:\WINDOWS\system32\cmd.exe
Test Case 1
-----
The hand is empty
Size: 0

Test Case 2
-----
Ace of Spades
Queen of Hearts
The hand is not empty
Size: 2

Test Case 3
-----
Location for getting a card was invalid
Press any key to continue . . .
```

**Figure 13.5. Test Cases 1-3 Results**

### *Write the Code, Part 3*

Finally, we write the code for our `RemoveCard` method. Remember when we had to handle someone trying to take the top card from an empty deck? Well, we also need to handle someone trying to remove a card from an empty hand or trying to remove a card from a location outside the range of locations for cards in the hand. It's not that bad, though; we can handle this the same way we did in the `GetCard` method:

```
/// <summary>
/// Removes the card from the given location in the hand
/// </summary>
/// <param name="location">the 0-based location of the card</param>
/// <returns>the card or null if the location is invalid</returns>
public Card RemoveCard(int location)
{
    // check for valid location
    if (ValidLocation(location))
    {
        Card card = cards[location];
        cards.RemoveAt(location);
        return card;
    }
    else
    {
        // invalid location
        return null;
    }
}
```

Remember we said there will be times when we write `private` methods for internal processing in the class? We actually realized as we were writing the `RemoveCard` method that we were writing the exact same Boolean expression for the `if` statement as the one we used in the `GetCard` method. Rather than

having that duplicated code, we decided to move it into a new `ValidLocation` method that returns `true` if the location is valid and `false` otherwise. That method is:

```
/// <summary>
/// Tells whether or not the given location is valid
/// </summary>
/// <param name="location">the location</param>
/// <returns>true if the location is valid, false otherwise</returns>
bool ValidLocation(int location)
{
    return location >= 0 && location < cards.Count;
}
```

We call the `ValidLocation` method from the `RemoveCard` method as shown above, and we also changed the `GetCard` method to call the method as well. Not only does that put the validation logic in a single place, it also makes it more obvious what the Boolean expression is checking in the if statements in each of the methods.

### *Test the Code, Part 3*

Now that we have the `RemoveCard` method implemented, we can add Test Case 4 to our test driver code and run that test case. Here's the code we add for Test Case 4:

```
// Test Case 4
Console.WriteLine("Test Case 4");
Console.WriteLine("-----");
hand = new Hand();
hand.AddCard(new Card(Rank.Ace, Suit.Spades));
hand.AddCard(new Card(Rank.Queen, Suit.Hearts));
hand.AddCard(new Card(Rank.Jack, Suit.Clubs));
hand.RemoveCard(2).Print();
hand.RemoveCard(0).Print();
hand.RemoveCard(0).Print();
card = hand.RemoveCard(0);
PrintLocationValidity(card, "removing a card");
Console.WriteLine();
```

When we run this code, we get the results shown in Figure 13.6, so we're done solving this problem.

```
C:\WINDOWS\system32\cmd.exe
Test Case 1
-----
The hand is empty
Size: 0

Test Case 2
-----
Ace of Spades
Queen of Hearts
The hand is not empty
Size: 2

Test Case 3
-----
Location for getting a card was invalid

Test Case 4
-----
Jack of Clubs
Ace of Spades
Queen of Hearts
Location for removing a card was invalid

Press any key to continue . . .
```

**Figure 13.6. All Test Case Results**

## 13.9. Common Mistakes

### *Not Including an Argument for Each Parameter*

When we write our method header, we define the parameters that method requires. Any code that calls this method has to provide the required information, providing exactly one argument for each parameter in the method header. Trying to call the method with too few or too many arguments results in a compilation error.

### *Mismatched Data Types Between Arguments and Parameters*

The data type of an argument must match (be compatible with, to be more precise, but we won't worry about that distinction here) the data type of the parameter for which it's being provided. A type mismatch between the arguments and parameters results in a compilation error.

### *Mixing up the Order of Arguments and Parameters*

This one occurs if you call a method with arguments that meet the number and type restrictions, but you mix up the order. Your program could compile – it just won't work correctly! For example, if you call a method to print out someone's height and weight (with the parameters in that order) with arguments of weight and height (note the incorrect order), the method will print the height as the weight and the weight as the height. While the problem in this example would be pretty easy to discover and correct, it's a lot harder to identify this kind of problem with methods that do calculations.

### *Method Never Runs*

Remember, to make a method run, you actually have to call it from another method (either in the class containing the method or in a different class). If you've written a method that never runs when you execute your program, you probably forgot to add the code that calls the method.

# Chapter 14. Unity Text IO

We already know how to do keyboard input in a console application using the `Console` class and we also know how to use mice, keyboards, and controllers to control game objects in our Unity games.

That gives us some of the input part of a user interface (UI) we'd want to provide for a game, but it doesn't give the user any text output (like for their current score) and it doesn't let the user provide text input (like their Gamertag). In this chapter, we'll learn how to implement those important user interface components. As the title of this chapter suggests, text input and output is often referred to as text IO.

Of course, another important part of a game's UI lets the user interact with the menu system. That piece of the UI requires some more advanced C# knowledge, though, so we'll put that off until we get to Chapter 17.

## 14.1. A Big Idea

Before we start covering the details of Unity Text IO, we want to (briefly) discuss an important idea in Unity UI: the **Canvas** component. As the Unity manual says, "The **Canvas** component represents the abstract space in which the UI is laid out and rendered." Think of a Canvas as a place we put our UI elements on and you have the basic concept. Sometimes we'll have a single canvas in our scene and sometimes we'll have multiple canvases, but every UI element we include in the scene will be a child of a **GameObject** that has a **Canvas** component attached to it.

## 14.2. Text Output

One of the things that we've been sorely missing in our games so far is the ability to share textual information with the player. For example, games regularly have a score, and though we could certainly keep track of a player's score using a variable, we don't currently have a way to tell the player what their score is! That's all about to change.

Let's add a score display to the fish game we built in Chapter 8, where the player drives a fish around eating teddy bears. First, we need to add a **Text** component to our scene. Right click in the Hierarchy window and select **UI > Text**. As you can see, you actually end up with a number of new components, including a **Canvas** that the text is drawn on in the game. Change the name of the **Text** component to **ScoreText**.

Select **ScoreText** in the Hierarchy window and change the **Pos X** and **Pos Y** values in the **RectTransform** component to move the text to be near the upper left corner of the screen (we used -300 and 160 for these values). In the **Text (Script)** component, change the **Font Style** to **Bold**, the **Font Size** to 24, and the **Color** to white.

If you run the game now, you'll see that you can use the keyboard to drive the fish around eating teddy bears, but the score display always says **New Text**. Let's fix that now.

Before we do that, though, we need to decide who should update the score display. Should we implement another script that we attach to the main camera to do this for us? Should we have a teddy bear update the score display when it's destroyed? Should we have the fish update the score display when it destroys a teddy bear? As you can see, we have lots of options here.

We're going to add fields and processing to the `Fish` script to have it keep track of the current score and to update the score display when it destroys a teddy bear. We decided to pursue this option because the fish is the player's avatar (how often do you get to say that?), and it makes sense to have the player keep track of their own score. Having each player keep track of their own score makes even more sense when we implement multiplayer games, so it's reasonable to use the same approach in single player games as well.

We'll start by adding three fields to the `Fish` script: a `score` field to keep track of the current score, a `bearPoints` field that tells how many points each bear is worth, and a `scoreText` field that saves a reference to the `ScoreText` component for efficiency. We mark the `bearPoints` and `scoreText` fields with `[SerializeField]` so we can set them in the Inspector:

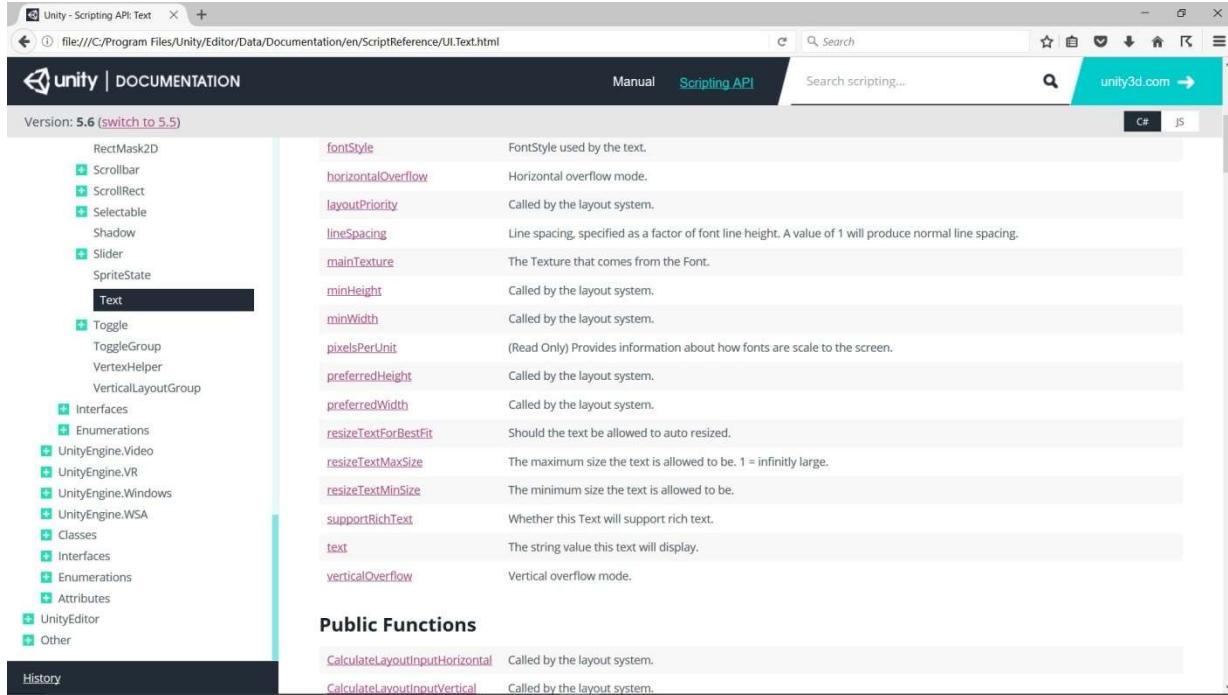
```
// score support
int score;
[SerializeField]
int bearPoints;
[SerializeField]
Text scoreText;
```

The `Text` class is actually contained in the `UnityEngine.UI` namespace, so we need to include a `using` directive for that namespace to get our changes to compile.

In the Unity editor, select the Fish game object in the Hierarchy window. Set the Bear Points value in the Inspector to 10 and drag the Score Text component from the Hierarchy window onto the Score Text field in the Inspector. Click the Prefab Apply button near the top of the Inspector.

If you select the Fish prefab in the Project window, you'll see that the Bear Points value has been saved in the prefab but the Score Text field still says None (Text). That's because prefabs can only refer to objects contained in the Project window and our Score Text component is only in the Hierarchy window. That doesn't cause us any trouble at all for this particular problem, so we'll leave our project as it is.

Next, we need to set the score text properly when the game starts; the appropriate place to do that is in the `Fish` Start method, but before we can do that we need to know how to interact with a `Text` object to change the text it displays. See the documentation below (which we retrieved by searching for `Text` in the Unity Scripting Reference and scrolling down in the pane on the right).



**Figure 14.1. Text Class Documentation**

As you can see, we can simply set the `text` variable to the string we want to display. Given that, we add the following code to the `Fish` Start method:

```
// set initial score text
scoreText.text = "Score: " + score;
```

When we run the game now, we get the output shown in Figure 14.2.



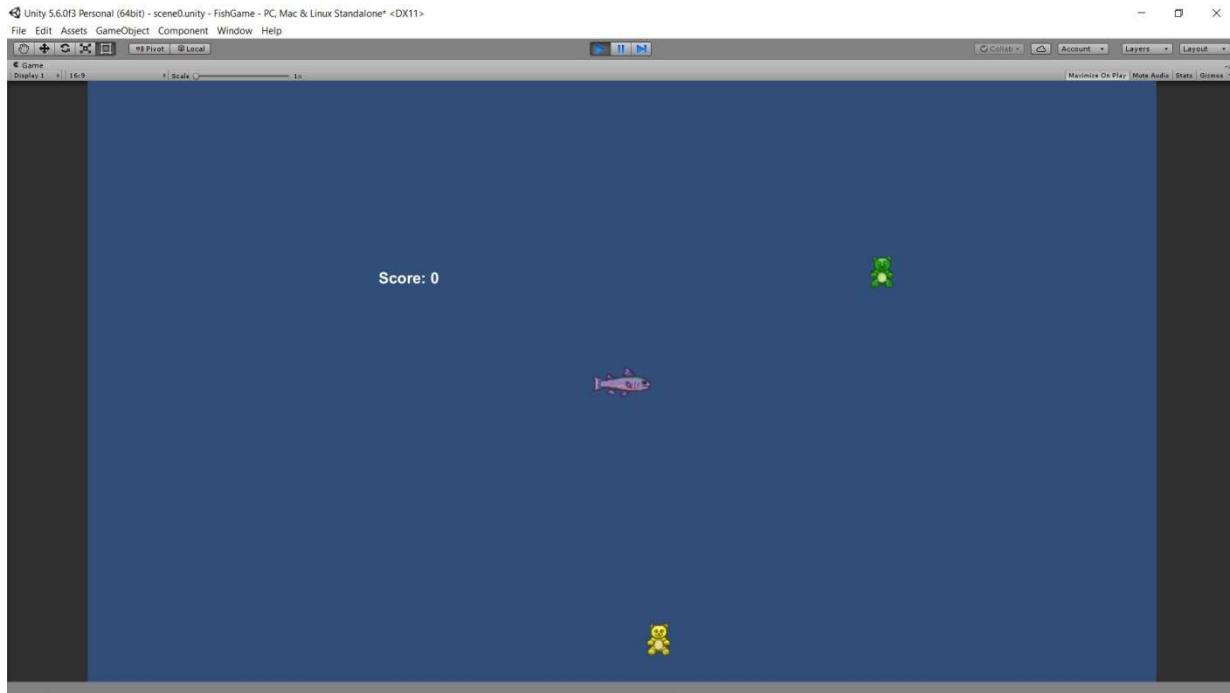
**Figure 14.2. Initial Score Output**

This is nicer than seeing "New Text" for our score display, but unfortunately our score stays at 0 no matter how many fish we eat. The good news is that we only need to add two more lines of code, this time in the `Fish` `OnCollisionEnter2D` method when we detect a collision between the fish's head and a teddy bear:

```
// update score
score += bearPoints;
scoreText.text = "Score: " + score;
```

Run the game one more time and you'll see the score increase as you eat teddy bears. Awesome.

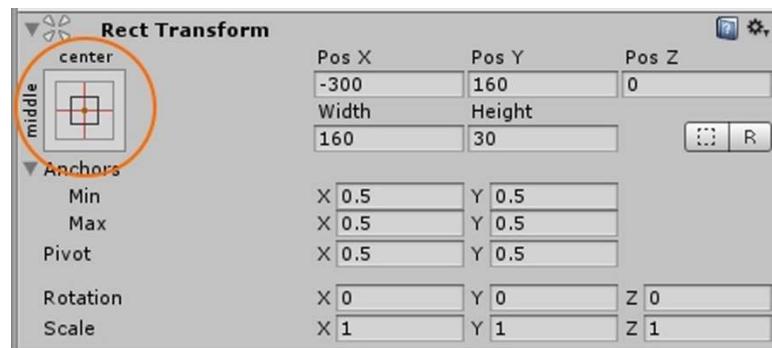
Unfortunately, although you might think we're done, check out Figure 14.3 to see what happens if we run the game with Maximize on Play selected.



**Figure 14.3. Maximize on Play Score Output**

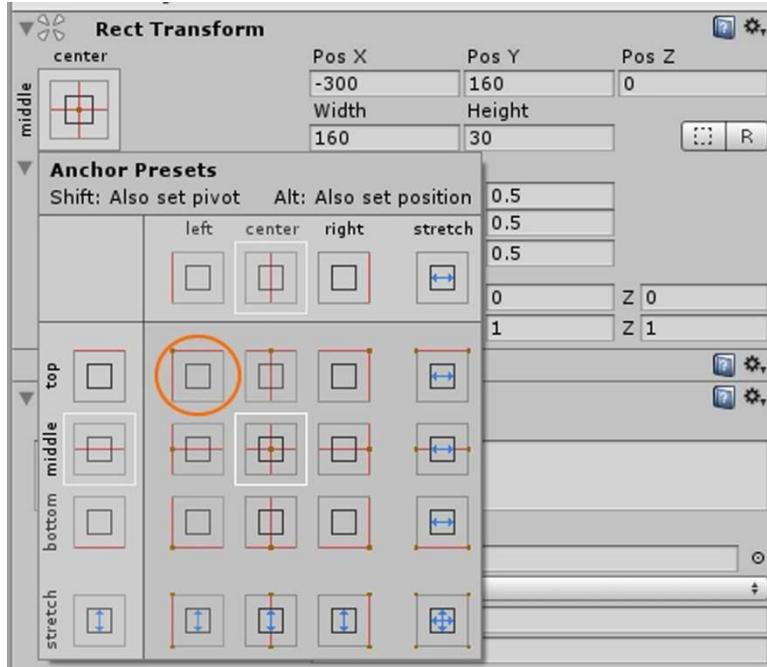
The score is definitely not in the correct place any more. Stop running the game to return to the editor.

Luckily, this is an easy problem to solve. Select the ScoreText game object in the Hierarchy window. In the Inspector, click the circled area in the Rect Transform component as shown in Figure 14.4.



**Figure 14.4. Anchor Presets Button**

Clicking this button gives us access to a set of presets for anchoring our Text component to a particular position on the Canvas. Select the circled preset as shown in Figure 14.5, which will anchor the Text component to the upper left corner of the Canvas.



**Figure 14.5. Upper Left Anchor Preset**

Click in the Inspector to close the anchor presets popup. Notice that this step changed the Pos X and Pos Y values for the Rect Transform; we then "tweaked" those to be 150 and -80.

Run the game again and you'll see that the score text appears in the upper left corner whether or not we have Maximize on Play selected. NOW we're done with this section!

### 14.3. Text Input

Now we know how to provide text output in our games, but there might also be times when we need to get text input from the player; for example, we might want to know their Gamertag.

To get text input, we use another user interface component called an Input Field. The Input Field isn't a stand-alone component, though; we need to add it to a Text component. By the way, Input Field is called an "Interaction Component" in Unity because it's a component the player can interact with.

Start by creating a new Unity Project, add a Text component to the scene, and rename the Text component GamertagText. Select the Text component in the Hierarchy window and change the Pos X and Pos Y values in the Rect Transform component to 0. In the Text (Script) component, change the Color to white.

At this point, the text is displayed when we run the game (just like the score was in the previous section), but the player can't actually interact with the text to change it. Let's fix that now.

Click the Add Component button in the Inspector and select UI > Input Field. Now drag the GamertagText game object from the Hierarchy window onto the Text Component value of the new Input Field (Script) component. You may get a warning that says "Using Rich Text with input is unsupported". If you do, uncheck the Rich Text checkbox under the Character heading in the Text component.

You should have noticed that the New Text we used to see displayed has disappeared from the Game view (and the Text value in the Text component has been cleared out). That's because the text that should be displayed will now be included in the Input Field component instead. Change the Text field in that component to say Enter Gamertag ... To actually center the text in the game window, change the Alignment value under the Paragraph heading in the Text component to centered.

At this point, the player can actually click on the text and change it using the keyboard, but there's really no way for them to know that! Let's change some visual characteristics of the Input Field to make this more obvious to the player.

First, let's make the text change color when the player hovers the mouse over it so they understand they can do something. To do this, change the Highlighted Color of the Input Field to be different from the Normal Color. If you run the game now, you'll see the text color change appropriately.

Once the player sees the text color change when they mouse over it, they're likely to actually click the text. At that point, the Input Field is selected; this is obvious because we now have a blue background under the text (you can of course change that color as well by changing the Selection Color value). The player can now type in their Gamertag.

You can see in the Inspector that the Input Field has a Character Limit value that defaults to 0, which means there's no limit on the number of characters the player can enter. If they enter more characters than will fit in the Text object, characters scroll off to the left (but are still included as part of the player's input). You can limit the number of characters the player can input by changing the Character Limit value.

Okay, so the player has entered their Gamertag into the Input Field. How do we actually get the text they entered? The best way to do this would be to grab that text once they finish by either pressing Enter or by clicking somewhere else in the game window, but doing that requires some ideas we won't learn about until Chapter 17. Instead, we'll demonstrate using a less efficient way that only uses concepts we've already learned, but you should definitely use the Chapter 17 approach in your actual games. Think of this demonstration as testing code rather than "ship it in the game" code.

Start by adding a new scripts folder in the Project window and by adding a new `PrintGamertag` C# Script to that folder. Open up the new script in Visual Studio and change it to the following:

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

We need to add a using directive for the `UnityEngine.UI` namespace because the `Text` and `InputField` classes are in that namespace.

```

/// <summary>
/// Prints the gamertag every second
/// </summary>
public class PrintGamertag : MonoBehaviour
{
    // make visible in Inspector
    [SerializeField]
    Text gamertagText;

    // saved for efficiency
    InputField gamertag;

```

We mark the `gamertagText` field with `[SerializeField]` so we can populate it in the Inspector and we save a reference to the `InputField` for the Gamertag so we don't have to retrieve it from the `gamertagText` field every time we need to output its contents.

```

// output gamertag once per second support
float secondsSinceLastOutput = 0;

```

We use the `secondsSinceLastOutput` field to implement a simple timer that "goes off" approximately every second.

```

/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    gamertag = gamertagText.GetComponent<InputField>();
}

```

We save the reference to the `InputField` here so we don't have to retrieve it from the `gamertagText` field every second.

```

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    // output gamertag every second
    secondsSinceLastOutput += Time.deltaTime;
    if (secondsSinceLastOutput > 1)
    {
        secondsSinceLastOutput = 0;
        print(gamertag.text);
    }
}

```

The first line of code adds the time it took for the previous frame to execute to the `secondsSinceLastOutput` field. The if statement checks to see if it's been over a second since the last output. If it has been, the body of the if statement resets the timer to 0 and outputs the current contents of the `InputField` object by accessing the `text` field of that object.

Attach the script to the Main Camera in the scene and populate the Gamertag Text field by dragging the GamertagText component from the Hierarchy window onto the field. If you run the game, you'll see the current value of the Input Field displayed in the Console window approximately every second, and you can see that value change as the player edits the contents of that Input Field.

There are many ways to make player text input more visually appealing and to process that input more efficiently using Chapter 17 concepts, but this example should give you a good understanding of how we can get text input from the player.

#### **14.4. Putting It All Together: The Circle Problem in Unity Revisited**

Let's solve the same problem we solved in Chapter 4, but this time we'll have each circle print their information as text on the screen instead of in the Console window. Here's the problem description:

Display circles with integer radii from 1 to 5 in the game window, with each circle displaying its radius and area.

##### *Understand the Problem*

As before, the problem description says to display the circles "in the game window", but it doesn't say where, so we'll place those circles as we see fit. Also, the problem description doesn't say where we should have each circle display its radius and area, so we'll just have each circle display that information below itself.

##### *Design a Solution*

Because each circle will be responsible for displaying itself and its information once it's been placed in the scene, it makes sense to modify the `Circle` script and implement the required functionality in the `Start` method in that script.

##### *Write Test Cases*

Unfortunately, once we have graphical (rather than textual) output, it becomes harder to exactly specify our expected results. For example, we don't know precisely where each of the circles will be placed in the scene, so we won't indicate in our expected results where each circle will appear. We do know, however, that there should be 5 circles with radii from 1 to 5, and we also know the area that should be displayed for each circle. Here's our test case:

##### **Test Case 1: Checking Radius and Area Info**

Step 1. Input: None.

Expected Result: Five different circles, with radii 1, 2, 3, 4, and 5. Radius and area displayed below each circle, with the following values:

Circle with radius 1, area 3.141593

Circle with radius 2, area 12.56637

Circle with radius 3, area 28.27433

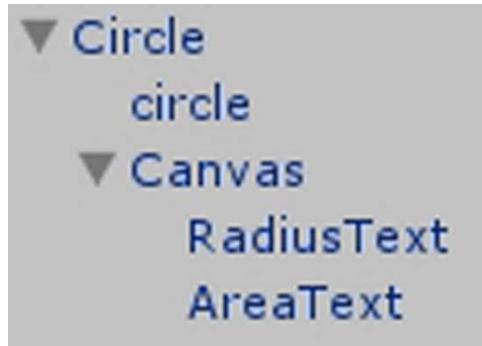
Circle with radius 4, area 50.26548

Circle with radius 5, area 78.53982

### Write the Code

We'll include our modified `Circle` script below, and interleave our comments where things are different.

Before we get to the code, though, let's look at the structure of our new Circle game object in the Unity editor. We zoomed in on a Circle game object in the Hierarchy window to generate Figure 14.6.



**Figure 14.6. Circle Object in Hierarchy window**

The Circle game object has a number of child game objects. The ones we'll work with are the circle sprite (so we can scale the sprite based on the radius) and the RadiusText and AreaText objects (so we can place them under the circle and set the displayed radius and area values). As you can see, for our solution to this problem, each Circle game object has its own Canvas.

In case you're wondering how we built this object, we started by right clicking in the Hierarchy window and selecting Create Empty. Next, we dragged the circle from the sprites folder onto our new game object, which made the sprite a child of that game object. Next, we right clicked our game object, selected UI > Text, and named the new Text component RadiusText. We then repeated those steps to add the AreaText component. Finally, we dragged our game object into the prefabs folder in the Project window and renamed the prefab Circle.

Okay, let's work our way through the `Circle` script, which we left attached to the Circle game object:

```

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

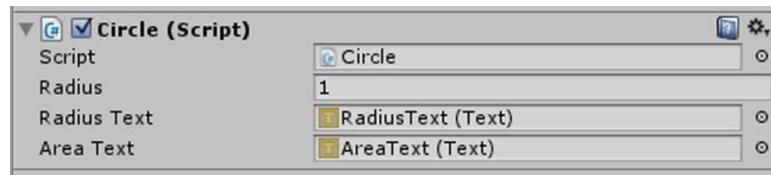
We need to add a using directive for the `UnityEngine.UI` namespace because the `Text` class is in that namespace, and we use `Text` objects to display the radius and area information for each circle.

```

/// <summary>
/// A circle
/// </summary>
public class Circle : MonoBehaviour
{
    // make visible in the Inspector
    [SerializeField]
    int radius;
  
```

```
[SerializeField]
Text radiusText;
[SerializeField]
Text areaText;
```

As the comment above states, by marking the above three fields (variables) with `[SerializeField]`, we make them visible in the Inspector; see Figure 14.7.



**Figure 14.7. Circle Script in Inspector**

Doing it this way is helpful because we can just change the `radius` field in the Inspector when we place a circle game object. To populate the `radiusText` field, we drag the `RadiusText` object from the Hierarchy window (Figure 14.6) onto the box for the field in the Inspector; populating the `areaText` field uses the same drag-and-drop process.

You might wonder why we didn't use properties here instead. As a reminder, properties don't appear in the Inspector, while fields marked with `[SerializeField]` do. Because we really wanted access to these fields in the Inspector, we couldn't use properties to provide access to them.

```
float area;

// for placing text below circle
const int RadiusTextBaseYOffset = -30;
const int AreaTextBaseYOffset = -50;
const int TextYOffsetScaleIncrement = -25;
```

The `area` field holds the calculated area for the circle; this field isn't marked with `[SerializeField]` because we don't need (and shouldn't have) access to it in the Inspector.

We use the first two constants above to offset the radius and area text from the center of the circle sprite when the circle sprite is scaled to actual size (the scale is 1 in that case). Because that sprite is scaled based on the radius, we use the third constant above to shift the radius and area text based on the scaled size of the circle sprite.

```
/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // calculate area
    // note that we're using UnityEngine Mathf instead of System Math
    area = Mathf.PI * Mathf.Pow(radius, 2);

    // scale circle sprite based on radius
    SpriteRenderer spriteRenderer =
        GetComponentInChildren<SpriteRenderer>();
```

To understand the second line of code above, remember that Unity uses a component-based system. The circle sprite has a `SpriteRenderer` component, the circle sprite is a child of the Circle game object, and the `Circle` script is attached to the Circle game object. The line of code looks at all the children of the Circle game object, trying to find a `SpriteRenderer` component; when it finds it (in the circle sprite), it puts that component into the `spriteRenderer` variable. We need this reference to scale the circle sprite based on the Circle's radius.

```
Vector3 scale = spriteRenderer.transform.localScale;
scale.x *= radius;
scale.y *= radius;
spriteRenderer.transform.localScale = scale;
```

The code above is almost identical to the scaling code from Chapter 4. The only difference is that we're manipulating the Transform component for the Sprite Renderer component we received above rather than for the Circle object as we did in Chapter 4. We need to do it this way because we only want to scale the sprite for the Circle object, we don't want to scale the text.

```
// calculate center of circle as offset from center of screen
// in screen coordinates
Vector3 circleCenterScreenCoordinates =
    Camera.main.WorldToScreenPoint(transform.position);
Vector3 circleOffsetFromCenter =
    new Vector3(circleCenterScreenCoordinates.x - Screen.width / 2,
                circleCenterScreenCoordinates.y - Screen.height / 2,
                circleCenterScreenCoordinates.z);
```

This code looks a little complicated, so let's think about the big picture before examining the details. We need to know where the circle is located on the screen so we can shift the text to appear below the circle. When we built our Circle game object, we placed our text appropriately so it would be in the correct location for a circle (of radius 1) placed in the center of the screen. To move the text to the correct location for a circle that's NOT centered in the screen, we need to know how much the circle is offset in x and y from the center of the screen. The code above calculates that offset for us.

The tricky part to this is that transforms specify the position, rotation, and scale of game objects in the coordinate system for the world, but the text positions are specified in the coordinate system for the screen. That makes perfect sense, because game objects "live" in the game world, but in Unity text objects are part of the user interface (UI), so they "live" on the screen.

We actually need to use the Main Camera in the scene to do our coordinate conversions because the location and other characteristics of the camera determine where something in the world is shown on the screen. If this seems strange to you, think of zooming in or out with a digital camera. The "thing" you're aiming at with the camera is at a specific location in the world, but changing the zoom settings on the camera changes where that "thing" appears on the camera preview screen. The `Camera.main` part of the first line of code gives us access to the Main Camera in the scene.

The rest of the first line of code calls the `Camera` `WorldToScreenPoint` method to convert the world coordinates of our Circle game object to screen coordinates; remember, this is why we needed a reference to our Main Camera. The argument we pass to the method is the position of the Circle game object, which we get to by accessing the `position` field of the `transform` field of the Circle game

object. The `WorldToScreenPoint` method returns a `Vector3` object (we learned that by reading the documentation), so we store the result in a `Vector3` variable called `circleCenterScreenCoordinates`.

This should all feel a little familiar to you, of course. Back in Chapter 8, we retrieved the mouse position in screen coordinates and had to convert that position to world coordinates so we could use the position in world coordinates in our game. This is exactly the same idea, just going in the other direction.

Now that we have the location of the Circle game object converted to screen coordinates, we can calculate offsets from the center of the screen in the x and y directions; that's what the second line of code above does. The code calls the `Vector3` constructor to create a new `Vector3` object containing this information and puts the new object into the `circleOffsetFromCenter` variable. For the constructor we're calling, we provide x, y, and z components for the new object as arguments to the constructor.

Let's look at the first argument, the x component of the new vector. Unity gives us a very handy `Screen` class that gives us access to information about the current display, including the width and height of the display (in pixels). That helps us here because the screen coordinates are 0, 0 at the lower left corner of the screen, so we know that the horizontal center of the screen is at `Screen.width / 2`. The fields in the `Screen` class are static, so we access them using the class name rather than a specific `Screen` object. To calculate the actual offset of the circle in the x direction, we subtract the horizontal center from the horizontal center of the circle in screen coordinates. We do the subtraction in this order so that if the circle is left of center the x offset is negative and if it's right of center the x offset is positive. We do a similar calculation for the y offset, and of course we don't change the z component of the screen coordinates when we're working in 2D.

```
// shift text display based on circle position and scaling
Vector3 radiusTextLocalPosition =
    new Vector3(circleOffsetFromCenter.x,
                circleOffsetFromCenter.y + RadiusTextBaseYOffset +
                    radius * TextYOffsetScaleIncrement,
                0);
radiusText.rectTransform.localPosition = radiusTextLocalPosition;
Vector3 areaTextLocalPosition =
    new Vector3(circleOffsetFromCenter.x,
                circleOffsetFromCenter.y + AreaTextBaseYOffset +
                    radius * TextYOffsetScaleIncrement,
                0);
areaText.rectTransform.localPosition = areaTextLocalPosition;
```

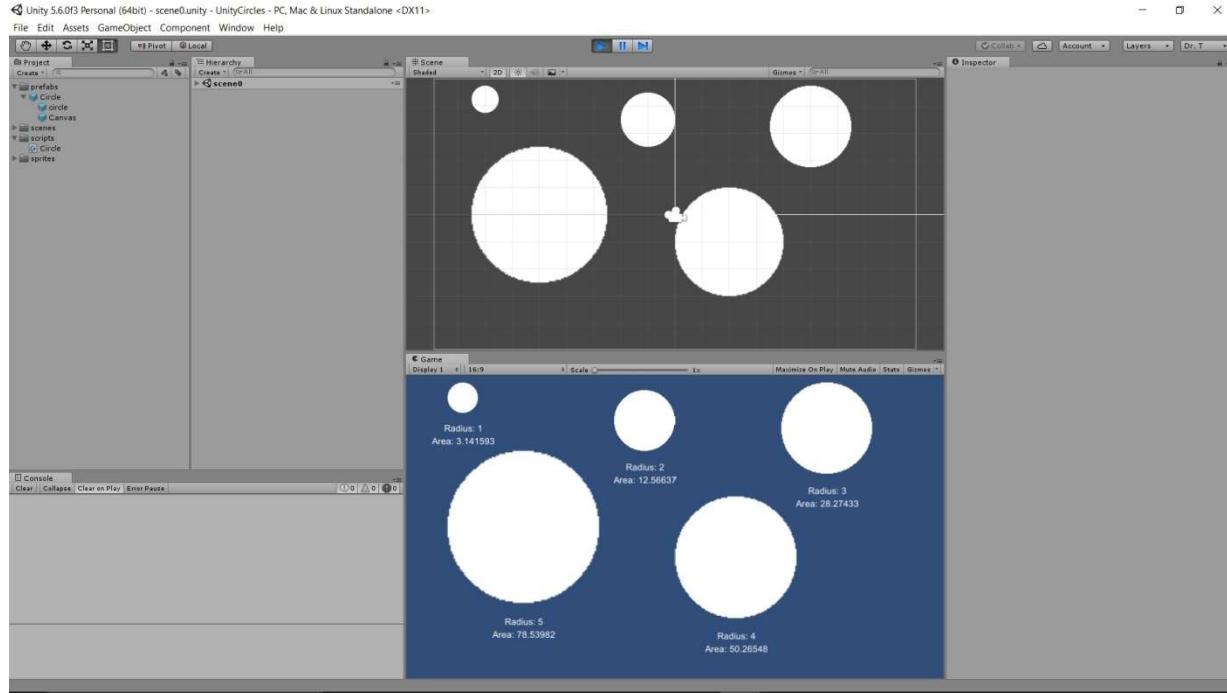
The first line of code above calculates the correct location for the radius text based on the circle offset from the center of the screen, the offset constants from the beginning of the script, and the radius of the circle; this is like what we did to calculate the appropriate scale for the circle sprite, though the math is a little more complicated. The second line of code then sets the actual position of the text, like we set the `localScale` field for the sprite renderer to actually scale the circle sprite. The third and fourth lines of code do the same for the area text.

```
// change text display for actual radius and area
radiusText.text = "Radius: " + radius;
areaText.text = "Area: " + area;
}
```

Finally, the two lines of code above seem pretty clear! As we learned earlier in this chapter, we can change the string that's displayed by a `Text` object by accessing the `text` field of that object; that's what those lines of code do.

### Test the Code

The actual results from running our Unity game are provided in Figure 14.8. As you can see, our actual results match our expected results. Whew!



**Figure 14.8. Test Case 1: Checking Radius and Area Info Results**

## Chapter 15. Unity Audio

At this point, we've included lots of the elements we need in good games. We know how to display images and animations, and we know how to respond to user input so the player can interact with the game world. There is, however, one more critical piece we need to include to provide an immersive play experience for the player. That critical piece is audio – both music and sound effects – so this chapter explains how to add audio to your games.

In this chapter, we're going to add sound to our Fish and Bears game from the previous chapter. The backgroundMusic sound will be used as background music in our game and the eat sound will be used when the fish eats a bear. The remaining two sounds – bounce1 and bounce2 – will be played when a bear bounces off the fish, but we want the game to randomly pick which one to play<sup>30</sup>. All the sounds are included in the solution on the Burning Teddy web site.

You should know that Unity has a very robust audio system; we're just scratching the surface in this chapter. You should explore the Audio section in the Unity manual to find out about all the other cool stuff you can do with audio in your Unity games.

### 15.1. Adding Audio Content

The first thing you need to do to add audio to your game is to actually create the audio assets. You can do that in a number of different ways, including using a microphone to record sounds or voices, converting music files into a usable format, and so on. The actual creation of those audio assets is beyond the scope of this book, so you'll have to explore how to create those on your own. It's important for you to know, however, that those assets need to be saved in a format supported by Unity. The good news is that Unity supports most common audio formats.

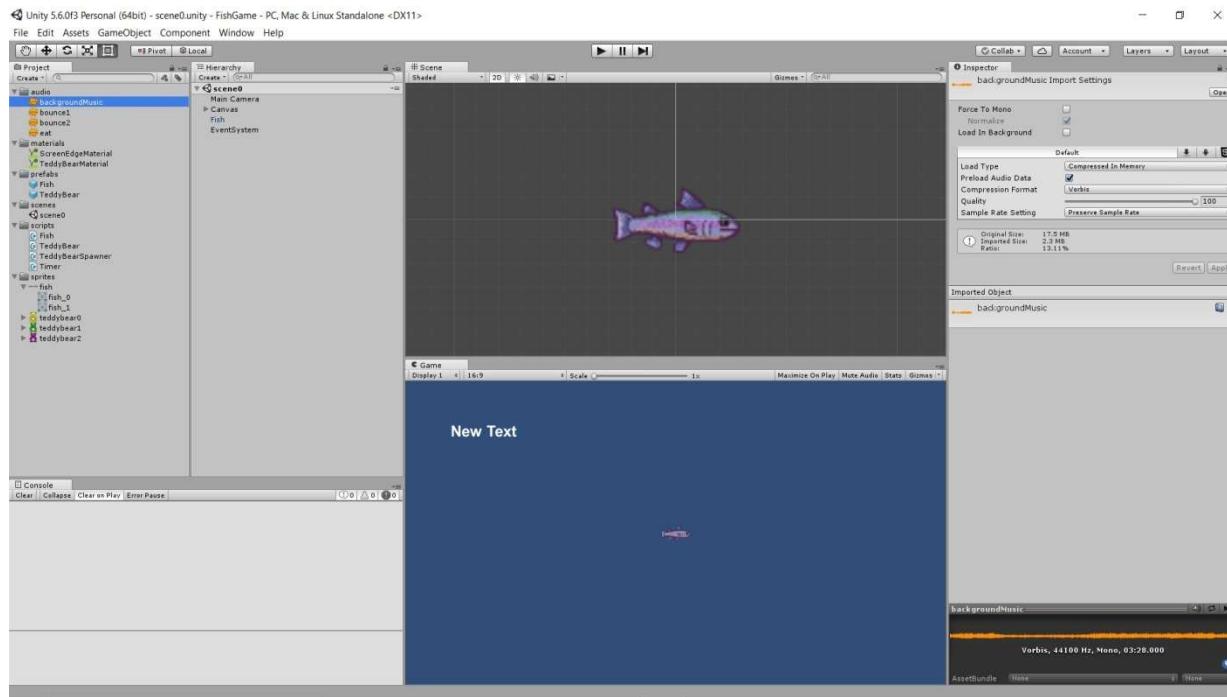
Adding the audio files to the Unity project is just as easy as adding sprites; all we need to do is copy the files into our project. We like to store our audio in a separate folder, so right click in the Project window and create another new folder called audio. Now go to your operating system and copy the audio files into the audio folder (which you'll find under the Assets folder wherever you saved your Unity project). When you copy the audio files into the audio folder, Unity automatically imports them as Audio Clips in your project.

Select backgroundMusic in the audio folder in the Project window; as you can see from the Inspector, we have a variety of characteristics we can set for each Audio Clip.

The Load Type tells how Unity loads the audio asset at runtime. The general rule of thumb is to use Decompress On Load (the default) for small files, Compressed In Memory for larger files, and Streaming to decode on the fly as the file is read from the disk. We'll change our backgroundMusic Audio Clip to use Compressed In Memory since this file is much larger than our sound effect files. Click the check box next to "Override for PC, Mac & Linux Standalone", click the Load Type dropdown, select Compressed in Memory, and click the Apply button. You should have something similar to Figure 15.1 at this point.

---

<sup>30</sup> The sound effects (but not the music) are actually from a commercial game our company was working on. Weird, we know.



**Figure 15.1. Audio Clip Import**

The Compression Format controls how compressed the files are and, of course, the quality of those sounds as well. Unity suggests using Vorbis (the default) for medium-length sound effects and music, so we'll leave that alone here. For short sound effects, PCM or ADPCM provide much better compression than Vorbis. Unity recommends ADPCM for sound effects that need to be played in large quantities; that doesn't really apply to our eat and bounce sound effects, so we'll change those to use PCM.

## 15.2. Audio Sources and the Audio Listener

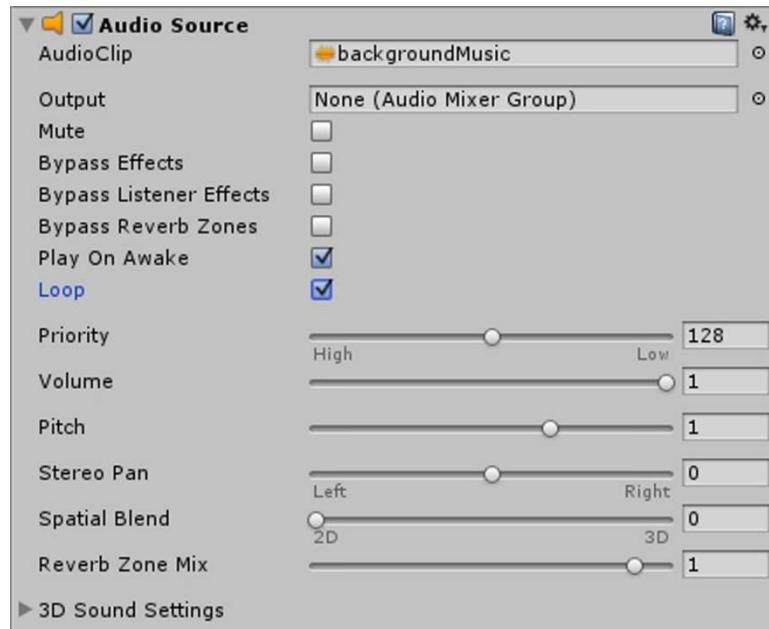
The general idea behind audio in Unity is that there are one or more Audio Sources (e.g., things making noise) and one Audio Listener (e.g., the thing that hears the noise). Not to get too philosophical, but if you have a bunch of Audio Sources making tree falling in a forest sounds but no Audio Listener, you won't hear any of those sounds in your game. To answer the age-old question, though – yes, of course the Audio Sources are making the sounds even if no one hears them!

By default, the Main Camera has an Audio Listener component attached to it. If you're doing fancy stuff like having the direction a sound is coming from important to player immersion, you might decide to remove the Audio Listener from the Main Camera and attach an Audio Listener to a different object (most commonly, the player) instead. For our 2D game here, though, leaving the Audio Listener on the Main Camera will work fine.

Now we need to add Audio Source components to the game objects that will play the audio in our game. We'll start with the background music; it makes sense for the Main Camera to just play the background music for the game. Yes, the Main Camera can make sounds and hear them as well. To see why this makes sense, make a sound right now. Did you hear it? See, it works!

Select the Main Camera in the Hierarchy window, click the Add Component button at the bottom of the Inspector, and select Audio > Audio Source. As you can see, there are lots of settings here to let you

tune the way audio works in your game! We only need to tweak a couple things for our game. First, drag `backgroundMusic` from the audio folder in the Project window and drop it on the `AudioClip` field of the `Audio Source` component in the Inspector. Click the check box for the `Loop` field so the background music loops in the game. The `Audio Component` in the Inspector should look like Figure 15.2.



**Figure 15.2. Main Camera Audio Source Component**

If you play the game now, you'll hear the background music. Sweet.

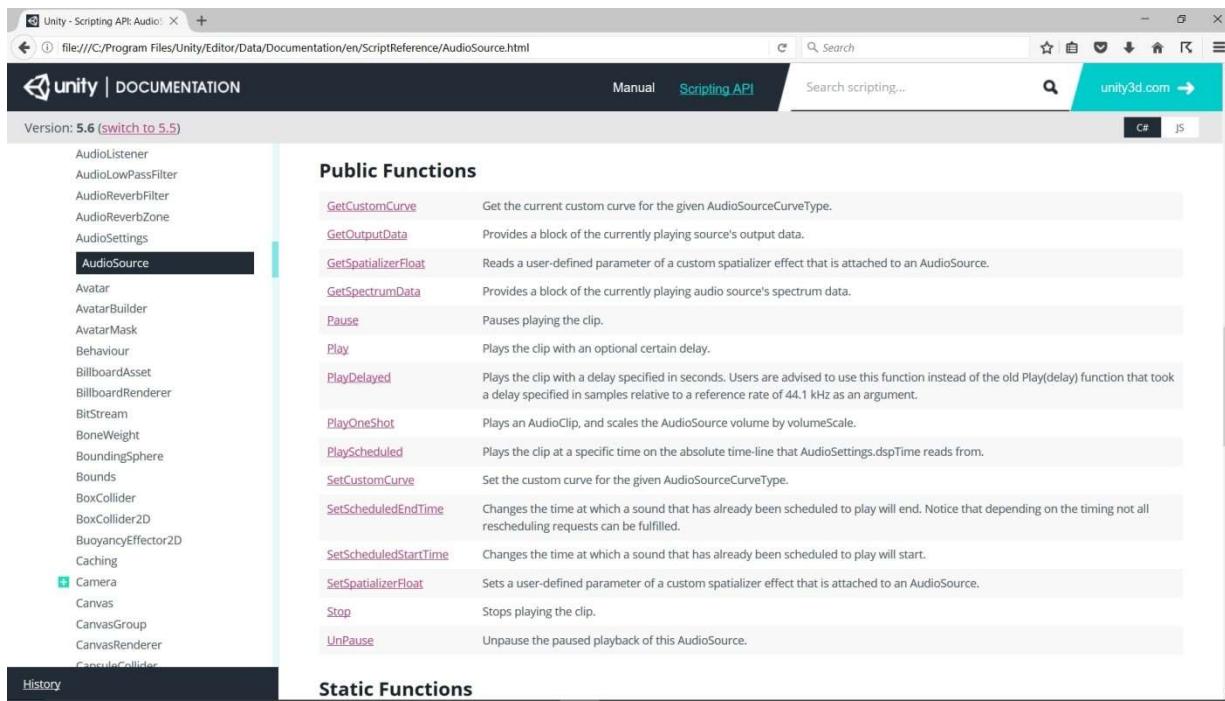
You should know that we've sometimes experienced strange Unity editor behavior where the audio works, but if we leave the editor open for a while and then play the game again the audio no longer works. Closing the Unity editor and then opening it up again consistently fixes this problem for us.

### 15.3. Playing Audio Clips from Scripts

The next thing we'll do is add the eat sound effect, since we (the authors) know that's probably the strangest sound effect in the game. Since we detect when the fish eats a teddy bear in the `Fish` `OnCollisionEnter2D` method, we'll add the `Audio Source` for this sound effect to the `Fish` prefab.

Select the `Fish` prefab in the `prefabs` folder in the Project window, click the `Add Component` button at the bottom of the Inspector, and select `Audio > Audio Source`. Drag `eat` from the audio folder in the Project window and drop it on the `AudioClip` field of the `Audio Source` component in the Inspector. Uncheck the check box for the `Play On Awake` field; we don't want the sound effect to play when we add the `Fish` to the scene (which is what `Play On Awake` does), we want to play the sound effect from the `Fish` script when the fish eats a teddy bear.

Okay, so how do we make an `Audio Source` play the clip that's associated with it? Check out Figure 15.3.



**Figure 15.3. AudioSource Documentation**

The  `AudioSource` class exposes a  `Play` method that lets us do exactly what we need. The next question, then, is how do we get access to the  `AudioSource` component(s) that are attached to the  `Fish` game object that the  `Fish` script is attached to?

It turns out that we can look them up (we'll show how soon), but for efficiency we don't actually want to look them up every time we need to play a sound. Instead, we'll start by adding a field to the  `Fish` script:

```
// sound effect support


```

We can now populate that field in the  `Fish Start` method using the following code:

```
// save audio sources


```

The first line of code returns all the  `AudioSource` components that are attached to the  `Fish` game object. At this point, there's only one  `AudioSource` component attached to the  `Fish` game object, so we could have used the  `GetComponent` method here instead, but looking forward we know we'll actually be adding  `AudioSource` components for the two bounce sounds as well, so we'll structure our code with that in mind.

The foreach loop walks through all the Audio Source components. The if statement checks if the name of the clip for the current audio source is the clip for the eat sound. If it is, we save the audio source into the field we declared above.

The last thing we need to do is actually play the eat sound when the fish eats a teddy bear. We already do a number of actions in the `Fish` `OnCollisionEnter2D` method for that case (destroying the teddy bear and adding points to the score), so we'll play the sound there as well:

```
// play eat sound
eatSound.Play();
```

If you play the game now, you'll hear the eat sound when the fish eats a teddy bear.

The last sound effect(s) we'll add to our game is for when a teddy bear bounces off the fish. Recall that we said that we want the game to randomly pick between the `bounce1` and `bounce2` sound effects when this happens.

Add two more Audio Source components to the Fish prefab, one for `bounce1` and one for `bounce2`. Be sure to uncheck the check box for the Play On Awake field for both components.

The next thing we'll do is add one more field to the `Fish` script (since we'll be playing these sound effects in that script):

```
List<AudioSource> bounce = new List<AudioSource>();
```

Next, we'll add code to the `Fish` `Start` method to add the Audio Sources for the `bounce1` and `bounce2` clips to our new `bounce` field. Here's the complete code for the audio source section of that method:

```
// save audio sources
AudioSource[] audioSources = gameObject.GetComponents<AudioSource>();
foreach (AudioSource audioSource in audioSources)
{
    if (audioSource.clip.name == "eat")
    {
        eatSound = audioSource;
    }
    else if (audioSource.clip.name == "bounce1" ||
              audioSource.clip.name == "bounce2")
    {
        bounce.Add(audioSource);
    }
}
```

Finally, we need to change the `Fish` `OnCollisionEnter2D` method to play one of the bounce sound effects when we've detected a collision between the fish and a teddy bear that's not at the front of the fish. We'll add an else body to the if statement that checks for a collision at the head of the fish, with the following code in the else body:

```
// play bouncing sound
bounce[Random.Range(0, bounce.Count)].Play();
```

For our index, we call the `Random` `Range` method, which will return either 0 or 1 (`bounce.Count` – which we know is 2 in this case – is the exclusive upper bound of the random number that gets generated). We then simply play the list element corresponding to the random index.

If you play the game now, you'll hear that sometimes `bounce1` plays when a teddy bear bounces off the fish and sometimes `bounce2` plays when that happens.

That's it for this chapter! We've added background music and sound effects to our game, which makes it a much more enjoyable experience for our players.

# Chapter 16. Inheritance and Polymorphism

One of the defining characteristics that distinguishes object-oriented programming languages like C# from other kinds of programming languages is the ability to use *inheritance*. Inheritance lets us structure our programs so that we can reuse code that others have written, even when we're developing new classes. Polymorphism is a related concept that's made possible through the use of inheritance. This chapter discusses both of these important object-oriented concepts and shows how we can use them in C#.

Up to this point in the book, we've used composition to build classes that contain objects of other classes. This is called a *has-a relationship*, since one class has (contains) objects of another class. In contrast, we'll use inheritance to express an *is-a relationship*, where one class is a specialized version of a more general class.

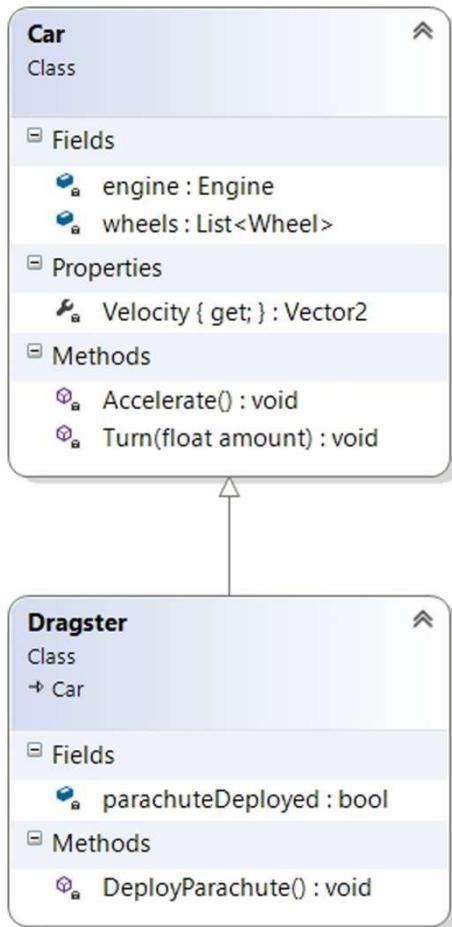
## 16.1. Inheritance Concepts

When we use inheritance in C# (and in other languages), we're really developing a *class hierarchy* that shows how our classes are related to each other. Typically, we start out by developing a general class, then we create other, more specialized, classes from that general class. For example, we might build a `Car` class that represents a general car object that has wheels and an engine, and can be started, stopped, and driven. But there are more specialized kinds of cars as well; for example, dragsters that require additional capabilities not provided by the `Car` class such as the ability to deploy the parachute. No one expects all cars to have this capability, so we'd have a more specialized `Dragster` class that's still a car as well.

The core idea behind inheritance is that we can structure our system of classes as a set of *parent* and *child* classes; these are also often called *superclasses* and *subclasses*. Why does this help us? Because a child class *inherits* all the fields, properties, and behavior from the parent class. If that was all that happens, we wouldn't be so excited (yes, we know you're excited, so don't try to hide it)! But there's more, much more! You also get a free set of knives ... oh, wait. There is more, but that's not it. In addition to inheriting everything from its parent class, a child class can both add fields and behavior that aren't in the parent class and change behavior that it inherited from the parent class. Let's look at both of those ideas.

In our `Car` and `Dragster` example above, we have the general car class with fields, properties, and behaviors that are common to all cars. If we make the `Car` class the parent of the `Dragster` class, the `Dragster` class inherits all those fields, properties, and behaviors. That makes perfect sense, because a `Dragster` is a `Car`; in fact, the inheritance relationship is commonly called an *is-a* relationship for this reason. But we also said above that a `Dragster` should be able to deploy a parachute, so we need to add a behavior that we didn't inherit from the `Car` parent class. How do we add a behavior to a class? The same way we've always done it – by adding a method to the class. So we can just add a `DeployParachute` method to the `Dragster` class to add that specialized behavior.

We capture the inheritance relationship between these two classes as shown in the UML in Figure 16.1; specifically, the child class (`Dragster`) is connected to the parent class (`Car`) with a line that has an open triangle at the parent class.



**Figure 16.1. UML Diagram for Car and Dragster Classes**

Notice that we've added a field to the `Dragster` class as well to keep track of whether or not the parachute is currently deployed. For child classes, we only list the new fields, properties, and methods; we don't need to list all the fields, properties, and methods that are inherited from the parent class.

This is a pretty small class hierarchy – we can end up with numerous levels of inheritance between parent and child classes in a larger program – but it does demonstrate an interesting characteristic of class hierarchies in general. Specifically, we can observe that classes higher in the class hierarchy are more general and classes lower in the class hierarchy are more specialized.

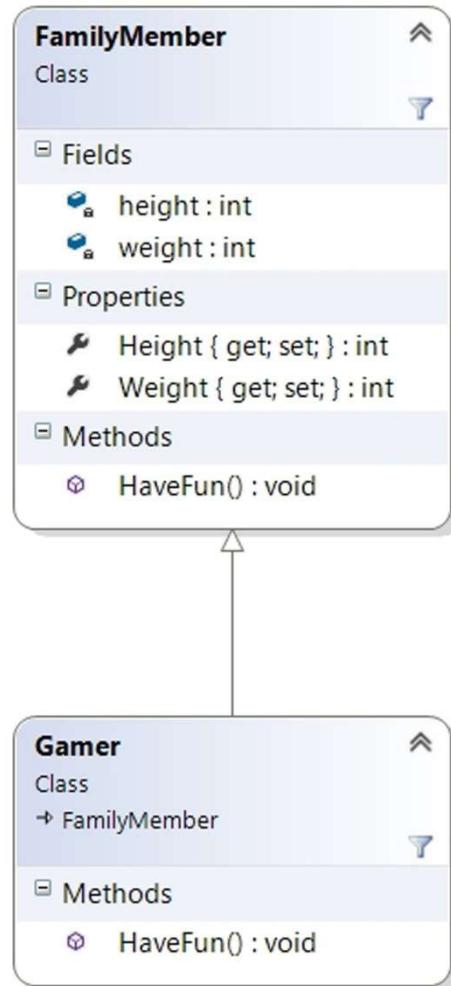
As a matter of fact, the root (top) of the C# class hierarchy is the `Object` class, the most general class of all in a C# program. Because all reference types in C# inherit from the `Object` class, we typically don't include that class in our class diagrams.

We get a couple of important benefits by making the `Car` class the parent class of the `Dragster` class. One benefit is that we only need to declare all the fields and define all the properties and methods for a generic car in one place (the `Car` class). If we made the `Dragster` class a “stand-alone” class instead of a child class, we'd have to copy all the fields, properties, and methods from the `Car` class into our `Dragster` class (and add the new stuff as well). We've said before that copying and pasting code is almost never a good idea, and it wouldn't be a good idea here either.

Even more importantly, if we use inheritance and we make changes to the `Car` class fields, properties, or methods, the `Dragster` class automatically gets those changes without any effort on our part. For example, if the `Car` class changes the way the `Accelerate` method works, calling the `Accelerate` method on a `Dragster` object would automatically use the modified inherited method. If instead we copied and pasted code from the `Car` class into the `Dragster` class, we'd have to remember that any time we changed the `Car` class we'd have to manually change the `Dragster` class as well. That may not seem so bad, but what if we also had `RallyCar`, `FormulaOneCar`, and `StockCar` classes as well? It would be crazy to try to keep them all synchronized using copying and pasting, especially since inheritance gives us a much more effective approach to use.

Well, we said we'd look at two different things we might do in a child class: add new behavior and change inherited behavior. The example above covers the first one, so let's move on to the second.

We change inherited behavior by *overriding* an inherited method. For this example, let's use the class hierarchy shown in Figure 16.2.



**Figure 16.2. Overriding Class Hierarchy**

The `FamilyMember` class has a couple of fields that would apply to all family members. It also has a `HaveFun` method defined as follows:

```

/// <summary>
/// Makes the family member have fun
/// </summary>
public virtual void HaveFun()
{
    Console.WriteLine("I'm writing code!");
}

```

This looks like the methods you've seen before, though we've added the `virtual` keyword to the method header. This keyword indicates that child classes are allowed to override the method.

Now let's say there's a specialized family member who is a gamer. Shockingly, this gamer doesn't have fun by writing code (we don't understand it either), they have fun by playing games. If we add the following code to the `Gamer` class

```

/// <summary>
/// Makes the gamer have fun
/// </summary>
public override void HaveFun()
{
    Console.WriteLine("I'm playing a game!");
}

```

then we've changed the inherited behavior to something else by overriding the inherited method. In fact, we include the `override` keyword in the method header of the child class to explicitly indicate that we're doing this.

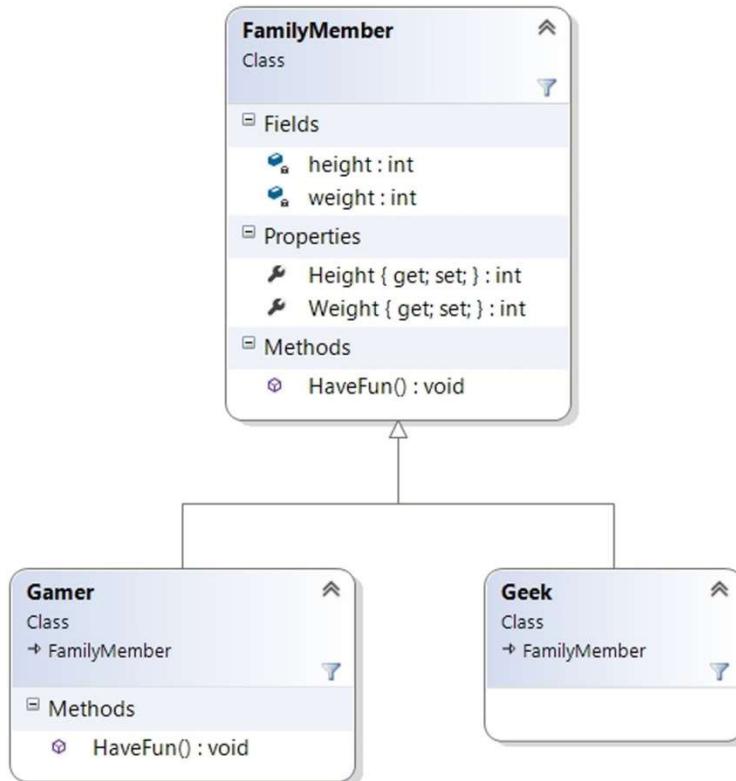
What does this do for us? If we call the `HaveFun` method on a `Gamer` object, we'll get the playing games message. We get the benefit of inheriting everything from the `FamilyMember` class while also getting the flexibility to change some of the inherited behavior if we need to.

One final comment before we move on. Although in real life people inherit characteristics from both their parents, in C# a child class can only have a single parent class. Although that might seem overly restrictive to you, letting child classes have multiple parents (not surprisingly called *multiple inheritance*) can lead to a number of serious problems, including something called the Diamond of Death.

Now that we have the basic inheritance concepts down, let's look at the syntax we use in C# to implement inheritance in our games.

## 16.2. Inheritance in C#

Let's run through all the syntax required to implement inheritance in C# for the class hierarchy shown in Figure 16.3.



**Figure 16.3. Example Class Hierarchy**

We know from personal experience that it's absolutely possible to be both a geek and a gamer, but let's ignore that possibility for this simple example.

We'll start by providing the implementation of the **FamilyMember** class. The code for that class is provided in Figure 16.4.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FamilyMemberExample
{
    /// <summary>
    /// A family member
    /// </summary>
    public class FamilyMember
    {
        #region Fields

        int height;
        int weight;

        #endregion

        #region Constructors
  
```

```

/// <summary>
/// Constructor
/// </summary>
/// <param name="height">the height of the family member</param>
/// <param name="weight">the weight of the family member</param>
public FamilyMember(int height, int weight)
{
    this.height = height;
    this.weight = weight;
}

#endregion

#region Properties

/// <summary>
/// Gets and sets the height
/// </summary>
public int Height
{
    get { return height; }
    set { height = value; }
}

/// <summary>
/// Gets and sets the weight
/// </summary>
public int Weight
{
    get { return weight; }
    set { weight = value; }
}

#endregion

#region Public methods

/// <summary>
/// Makes the family member have fun
/// </summary>
public virtual void HaveFun()
{
    Console.WriteLine("I'm writing code!");
}

#endregion
}
}

```

**Figure 16.4. FamilyMember.cs**

We haven't really done anything special yet to account for child classes for this class, though we have used the `virtual` keyword to indicate that child classes can override the `HaveFun` method. We've basically just developed the class to represent a general family member.

Let's work on our `Geek` class next. We make a class the child of a parent class in C# using a colon (`:`) in our class header. To make our `Geek` class a child class of the `FamilyMember` class, we use

```
public class Geek : FamilyMember
```

for our class header. This tells the compiler that `Geek` is a child class of the `FamilyMember` class.

We also need to develop the constructor for the `Geek` class. The complete code for the `Geek` class can be found in Figure 16.5.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FamilyMemberExample
{
    /// <summary>
    /// A geek
    /// </summary>
    public class Geek : FamilyMember
    {
        #region Constructors

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="height">the height of the family member</param>
        /// <param name="weight">the weight of the family member</param>
        public Geek(int height, int weight)
            : base(height, weight)
        {
        }

        #endregion
    }
}
```

**Figure 16.5. Geek.cs**

Notice that the constructor header contains something we've never seen before; specifically, it contains

```
: base(height, weight)
```

at the end of the header. This calls the constructor of the parent class for this class, which we want to do because the parent class constructor initializes the `FamilyMember` instance variables for us.

Okay, let's move on to the `Gamer` class; the code is provided in Figure 16.6.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```

namespace FamilyMemberExample
{
    /// <summary>
    /// A gamer
    /// </summary>
    public class Gamer : FamilyMember
    {
        #region Constructors

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="height">the height of the family member</param>
        /// <param name="weight">the weight of the family member</param>
        public Gamer(int height, int weight)
            : base(height, weight)
        {
        }

        #endregion

        #region Public methods

        /// <summary>
        /// Makes the gamer have fun
        /// </summary>
        public override void HaveFun()
        {
            Console.WriteLine("I'm playing a game!");
        }

        #endregion
    }
}

```

**Figure 16.6. Gamer.cs**

The constructor is very similar to the constructor we implemented for the `Geek` class, and the override of the `HaveFun` method is as we described in the previous section.

### 16.3. Polymorphism

We've already seen some very nice capabilities that inheritance gives us, but there's still another one we should talk about: *polymorphism*. Polymorphism means that a particular method call can behave differently – take multiple forms – based on the object on which it's called. Let's look at an example.

Let's build a list of 3 family members where one is a `FamilyMember`, one is a `Geek`, and one is a `Gamer`. Can we do this? Don't lists have to have elements that are all the same type or class? Well – yes, but because both `Geek` and `Gamer` are child classes of the `FamilyMember` class, we can declare our list as follows:

```
List<FamilyMember> familyMembers = new List<FamilyMember>();
```

This gives us a list of `FamilyMember` objects, and each of these objects can actually be either a `FamilyMember` object, a `Geek` object, or a `Gamer` object. In other words, we have a single variable (`familyMembers`) that refers to objects of different types, but we can do this because `Geek` and `Gamer` are child classes of `FamilyMember`. Let's add our family members to the list:

```
familyMembers.Add(new FamilyMember(69, 185));
familyMembers.Add(new Gamer(70, 200));
familyMembers.Add(new Geek(71, 165));
```

Okay, so now we have a list that's a mix of various object types, but you're still wondering why polymorphism is useful, right? Here's where it gets good. Let's go through the list and call the `HaveFun` method for each of the elements of the list:

```
foreach (FamilyMember familyMember in familyMembers)
{
    familyMember.HaveFun();
}
```

We don't have to know whether each element is a `FamilyMember`, a `Geek`, or a `Gamer` when we call the `HaveFun` method; because of polymorphism, the program automatically calls the appropriate `HaveFun` method for that item. The `FamilyMember` class uses the `HaveFun` method it defines, the `Geek` class inherits that method so it uses that method as well, and the `Gamer` class overrides that method to give it different behavior. The bottom line is that, through polymorphism, the program calls the appropriate method at run time based on the type of the object it's calling the method on.

Don't believe that it actually works that way? Check out the output in Figure 16.7.



**Figure 16.7. Polymorphism Program Output**

The first object in the list is a `FamilyMember` object so the call to the `HaveFun` method executes the method defined in that class and prints the coding message. The second object in the list is a `Gamer` object so the call to the `HaveFun` method executes the `HaveFun` method defined in the `Gamer` class. Finally, the third object in the list is a `Geek` object. The `Geek` class doesn't define a `HaveFun` method, but the `Geek` class inherits the `HaveFun` method from the `FamilyMember` class so it executes that method and prints the coding message.

You might be asking yourself, though, how the Common Language Runtime decides which method to use at run time. It does this using something called *method resolution*. Basically, it looks for a method with the given header in the actual class for the object. If the method is there, that's the one that will be used. If it's not, the CLR checks the parent class to see if the method is there. It uses the first method it finds that matches the method header, working its way all the way up to the `Object` class if necessary (more about that class soon). If the CLR couldn't find the method given your class hierarchy, you'll actually get a compilation error, so the method has to appear in the object you're using or in one of its parent classes.

Polymorphism is a very powerful capability that we get when we use inheritance. We've only scratched the surface here, but as you move on to develop more complicated solutions using inheritance, you're sure to use polymorphism again as well.

## 16.4. The Standard Bank Account Example

Let's explore a larger inheritance example. It's fairly common in textbooks to use a bank account class hierarchy to demonstrate the use of inheritance. It's common because it's a good example, so we'll use it here as well!

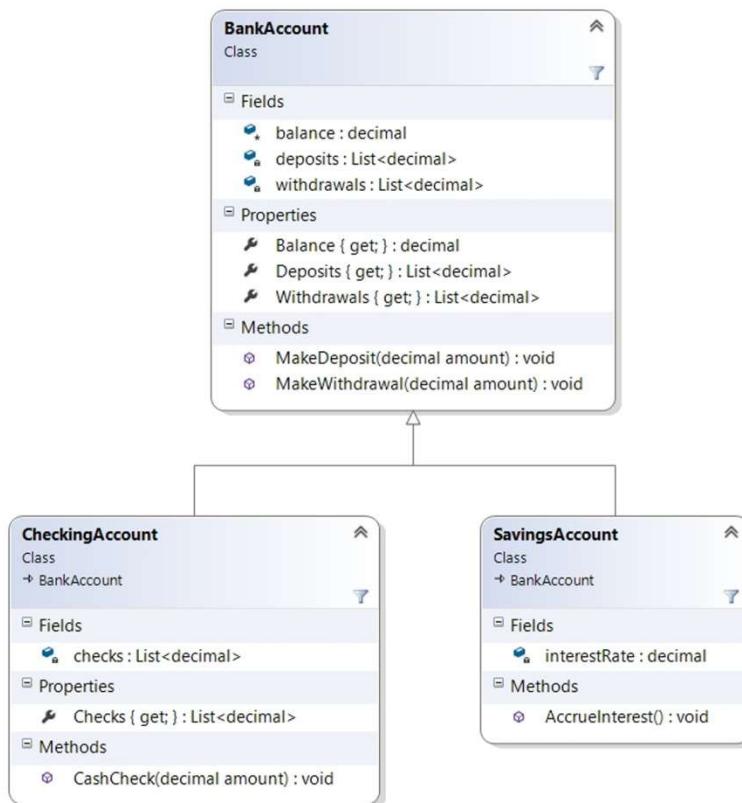
We'll start with a basic bank account. We want to be able to make deposits to and withdrawals from the account, so we need a `BankAccount` class that will accept deposits and provide withdrawals. We also probably want the bank account object to be able to give us the current balance in the account, so we need that as a property. Finally, to print a statement for the account at the end of each month, we'll have to maintain lists of the deposits and withdrawals for the account so we can print those lists on the statement as well.

It looks like the `BankAccount` class will need fields for the current balance, a list of deposits made to the account, and a list of withdrawals made from the account. We'll need a constructor (as always), and we'll need properties called `Balance`, `Deposits`, and `Withdrawals`. For the methods, we'll need methods called `MakeDeposit` and `MakeWithdrawal`.

Note that we've decided to use list objects (rather than arrays) for the lists of deposits and withdrawals since we don't know how long those lists will actually be. Using lists of `decimal` for the deposits and withdrawals isn't actually very realistic; we should really develop a `Transaction` class that tells what kind of transaction it is (deposit or withdrawal), the amount of the transaction, and the date and time of the transaction. We won't do that here so we can concentrate on the inheritance aspects of our solution, but if we were doing this "for real" we'd definitely store transactions, not just transaction amounts.

We haven't done anything that you haven't seen before yet, right? Here's where we get to the inheritance stuff. We've already developed a general bank account class that gives us useful fields, properties, and methods for all kinds of bank accounts, but we might need more. If we have a checking account, for example, we need to keep track of checks that have cleared the bank and deduct them from the account balance. And if we have a savings account, we need to keep track of the interest that has accrued for that account. Rather than making two brand new classes with most of the same fields, properties, and methods as our original bank account class, we'll make a checking account class and a savings account class as child classes of the bank account class. They'll have all the fields, properties, and methods of the bank account class – they inherit them from the parent class – but we also add new fields, properties, and methods to our checking account and savings account classes.

The resulting UML diagram for our class hierarchy is shown in Figure 16.8.



**Figure 16.8. Bank Account Class Hierarchy**

Just as for transactions, we'd really want to know more about checks than just the check amount; we'd at least want to know the check number, date, and the payee for the check as well! We won't develop a `Check` class for this problem – remember, we're concentrating on the inheritance stuff – but we would if we wanted a complete solution.

Now let's actually implement the above class hierarchy using C#. We'll start with the `BankAccount` class; see Figure 16.9.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace BankAccounts
{
    /// <summary>
    /// A bank account that accepts deposits and withdrawals.
    /// We can also access the current balance and lists of
    /// deposits and withdrawals for the account
    /// </summary>
    public class BankAccount
    {
        #region Fields
  
```

```
decimal balance;
List<decimal> deposits = new List<decimal>();
List<decimal> withdrawals = new List<decimal>();

#endregion

#region Constructors

/// <summary>
/// Constructor
/// </summary>
/// <param name="initialDeposit">the initial deposit opening the
/// account</param>
public BankAccount(decimal initialDeposit)
{
    // set initial balance and add to list of deposits
    balance = initialDeposit;
    deposits.Add(initialDeposit);
}

#endregion

#region Properties

/// <summary>
/// Gets the balance in the account
/// </summary>
public decimal Balance
{
    get { return balance; }
}

/// <summary>
/// Gets the list of deposits for the account
/// </summary>
public List<decimal> Deposits
{
    get { return deposits; }
}

/// <summary>
/// Gets the list of withdrawals for the account
/// </summary>
public List<decimal> Withdrawals
{
    get { return withdrawals; }
}

#endregion

#region Public methods

/// <summary>
/// Adds the deposit to the account. Prints an error
/// message if the deposit is negative
/// </summary>
```

```

/// <param name="amount">the amount to deposit</param>
public void MakeDeposit(decimal amount)
{
    // check for valid deposit
    if (amount > 0)
    {
        // increase balance and add deposit to deposits
        balance += amount;
        deposits.Add(amount);
    }
    else
    {
        // invalid deposit, print error message
        Console.WriteLine(
            "Deposits have to be larger than 0!");
    }
}

/// <summary>
/// Deducts the withdrawal from the account. Prints an error
/// message if the withdrawal is larger than the account
/// balance
/// </summary>
/// <param name="amount">the amount to withdraw</param>
public void MakeWithdrawal(decimal amount)
{
    // check for valid withdrawal
    if (amount <= balance &&
        amount > 0)
    {
        // deduct withdrawal and add withdrawal to withdrawals
        balance -= amount;
        withdrawals.Add(amount);
    }
    else
    {
        // invalid withdrawal, print error message
        Console.WriteLine(
            "Not enough money for withdrawal amount!");
    }
}

#endregion
}

```

**Figure 16.9. BankAccount.cs**

There's nothing new here, so let's start working on our `SavingsAccount` class. We're going to talk about how child classes inherit fields, properties, and methods in the following sections, so for now we'll just create the class and add the new field and the constructor.

```

using System;
using System.Collections.Generic;
using System.Linq;

```

```

using System.Text;

namespace BankAccounts
{
    /// <summary>
    /// A savings account that pays interest
    /// </summary>
    public class SavingsAccount : BankAccount
    {
        #region Fields

        decimal interestRate;

        #endregion

        #region Constructors

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="initialDeposit">the initial deposit opening the
        /// account</param>
        /// <param name="interestRate">the interest rate for the
        /// account</param>
        public SavingsAccount(decimal initialDeposit, decimal interestRate)
            : base(initialDeposit)
        {
            this.interestRate = interestRate;
        }

        #endregion
    }
}

```

**Figure 16.10. SavingsAccount.cs**

The `SavingsAccount` constructor has two parameters: the initial deposit for the account and the interest rate to be applied each time we accrue interest for the account.

## 16.5. Inheritance and Fields

A child class inherits all the fields of the parent class. In other words, a checking account object will have fields for the account balance, the list of deposits into the account, and the list of withdrawals from the account. These fields are inherited from the `BankAccount` class. The checking account object will also have a field for the list of cashed checks, but that field is provided directly by the `CheckingAccount` class. There are therefore two kinds of fields in a child class: fields inherited from the parent class and new fields defined by the child class.

There are a couple of subtleties here, though. For example, although the child class object has the fields, it may not be able to directly reference them from its methods. For example, we'd get a compiler error if we tried to include the following line in a `CheckingAccount` method that cashes a check:

```
balance -= amount;
```

Specifically, the compiler would tell us

```
'BankAccounts.BankAccount.balance' is inaccessible due to its protection level
```

Why do we get this error? Because `private` is the default access modifier for all the fields in our `BankAccount` class (and any class we define), including the `balance` field. We've consistently made our fields `private` to preserve information hiding, but this causes problems when we want to use inheritance.

So what should we do? Well, we could change the access modifier for the field to `public`, but this would really break our information hiding since any user of the `BankAccount` class could then directly access the field. Luckily, C# provides the `protected` access modifier to address this issue. Protected fields can be accessed directly by child classes of the parent class. We're going to have to go back and change our access modifier for the `balance` field to `protected` so our child classes can access that field:

```
protected decimal balance;
```

We won't change the other fields, though, because we're not going to have to access them directly.

One last thing about fields in child classes. What happens if we create a new field in our child class that has the same name as a field in the parent class? The new field hides the field in the parent class. This can lead to some errors that are pretty hard to find, so you should only do this if you're positive you really need to.

## 16.6. Inheritance, Properties, and Methods

As for fields, a child class inherits all the properties and methods of the parent class. We can also create new properties and methods in the child class. Finally, as discussed above, we can override a method that's inherited from the parent class. Creating new properties and methods in the child class is pretty self-explanatory, so let's look at properties and methods that are inherited from the parent class.

For example, let's look at what happens in the following code:

```
CheckingAccount account = new CheckingAccount(100.00);
Console.WriteLine(account.Balance);
```

Even though we didn't explicitly create a `Balance` property in the `CheckingAccount` class, that class inherits the property from the `BankAccount` class. This property works the same way in the `CheckingAccount` class as it did in the `BankAccount` class, simply returning the balance in the account. So properties and methods that are inherited from the parent class simply work the same way in the child classes that inherit them.

Now that we understand how fields, properties, and methods work with inheritance, let's finish our `CheckingAccount` and `SavingsAccount` classes (and make the few changes to the `BankAccount` class that we've discussed as well). The complete code for the `CheckingAccount` class is provided in Figure 16.11; the `SavingsAccount` class code can be found in Figure 16.12.

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

```
using System.Linq;
using System.Text;

namespace BankAccounts
{
    /// <summary>
    /// A checking account that lets us cash checks and access the list of
    /// checks that have been cashed
    /// </summary>
    public class CheckingAccount : BankAccount
    {
        #region Fields

        List<decimal> checks = new List<decimal>();

        #endregion

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="initialDeposit">the initial deposit opening the
        /// account</param>
        public CheckingAccount(decimal initialDeposit)
            : base(initialDeposit)
        {
        }

        #endregion

        #region Properties

        /// <summary>
        /// Gets the list of checks for the account
        /// </summary>
        public List<decimal> Checks
        {
            get { return checks; }
        }

        #endregion

        #region Public methods

        /// <summary>
        /// Cashes the check of the given amount. Prints an
        /// error message if the check amount is larger than
        /// the account balance
        /// </summary>
        /// <param name="amount">the amount of the check</param>
        public void CashCheck(decimal amount)
        {
            // check for valid check amount
            if (amount <= balance)
            {

```

```

        // deduct check and add check to checks
        balance -= amount;
        checks.Add(amount);
    }
    else
    {
        // invalid check, print error message
        Console.WriteLine(
            "Not enough money in account to cover check");
    }
}

#endregion
}
}

```

**Figure 16.11. CheckingAccount.cs**

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace BankAccounts
{
    /// <summary>
    /// A savings account that pays interest
    /// </summary>
    public class SavingsAccount : BankAccount
    {
        #region Fields

        decimal interestRate;

        #endregion

        #region Constructors

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="initialDeposit">the initial deposit opening the
        /// account</param>
        /// <param name="interestRate">the interest rate for the
        /// account</param>
        public SavingsAccount(decimal initialDeposit, decimal interestRate)
            : base(initialDeposit)
        {
            this.interestRate = interestRate;
        }

        #endregion

        #region Public methods

```

```

/// <summary>
/// Adds accrued interest to the account balance
/// </summary>
public void AccrueInterest()
{
    // calculate interest and add to balance
    balance += balance * interestRate;
}

#endregion
}
}

```

**Figure 16.12. SavingsAccount.cs**

## 16.7. The Object Class

We've actually been inheriting methods since we started writing our own classes! C# has a class that's defined as the root (the top) of the entire C# class hierarchy; it's called the `Object` class. Whenever we define a class that doesn't explicitly extend another class, we're actually implicitly extending the `Object` class.

Let's take a closer look at the `ToString` method that the `BankAccount` class inherits from the `Object` class. Not surprisingly, the method converts an object of the class to a `string`. We can write a short program that prints out the `string` for a `BankAccount` object:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BankAccounts
{
    class Program
    {
        /// <summary>
        /// Demonstrates ToString method
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            // create new bank account and print ToString results
            BankAccount account = new BankAccount(100.00m);
            Console.WriteLine("Object: " + account.ToString());
        }
    }
}

```

When we run the above code, we get the output

```
Object: BankAccounts.BankAccount
```

The default `ToString` method from the `Object` class prints the “fully qualified name of the type of the Object.” Because our `BankAccount` class is declared in the `BankAccounts` namespace in our example, `BankAccounts.BankAccount` is the output of the default method.

But if we really want to convert a bank account object to a `string`, wouldn't we want some more meaningful information than the type for the object? Let's say that we'd like to print the balance when we print a particular object. How do we do that? By overriding the `ToString` method. Let's add the following method to the `BankAccount` class:

```
/// <summary>
/// Converts bank account to string
/// </summary>
/// <returns>the string</returns>
public override string ToString()
{
    return "Balance: " + balance;
}
```

Now when we run our driver code again, we get

```
Balance: 100.00
```

You'll probably find yourself overriding the `ToString` method fairly regularly in practice. It's certainly not required (we haven't done it up to this point), but it can be helpful for debugging and other display purposes.

## 16.8. Polymorphism Revisited

Let's solidify your understanding of polymorphism with one more example. Say we wanted a list of 6 bank accounts where the even elements are checking accounts and the odd elements are savings accounts. This is certainly a reasonable organization if we have 3 people, each of whom has a checking account and a savings account (we're a small bank!). We start by declaring our list as follows:

```
List<BankAccount> accounts = new List<BankAccount>();
```

This gives us a list of `BankAccount` objects, and each of these objects can actually be either a bank account object, a checking account object, or a savings account object. In other words, we have a single variable (`accounts`) that refers to objects of different types, but we can do this because `CheckingAccount` and `SavingsAccount` are child classes of `BankAccount`. Let's add our accounts to the list:

```
accounts.Add(new CheckingAccount(100.00m));
accounts.Add(new SavingsAccount(50.00m, 0.02m));
accounts.Add(new CheckingAccount(300.00m));
accounts.Add(new SavingsAccount(500.00m, 0.02m));
accounts.Add(new CheckingAccount(1000.00m));
accounts.Add(new SavingsAccount(50000.00m, 0.02m));
```

Now the bank decides to deposit \$20.00 into every account as a sign of appreciation to its customers. That means we need to go through the list and make a deposit of 20.00 into each account. Because of polymorphism, we can simply use:

```
// deposit $20 into each account
foreach (BankAccount account in accounts)
{
    account.MakeDeposit(20.00m);
}
```

We don't have to know whether each item is a checking account or a savings account when we call the `MakeDeposit` method; because of polymorphism, the program automatically calls the appropriate `MakeDeposit` method for that item. For both checking account and savings account objects, the `MakeDeposit` method is inherited from the `BankAccount` class, but the method is called on checking account or savings account objects. In addition, one or both of those classes could have overridden the `MakeDeposit` method and the code would still work properly. The bottom line is that, through polymorphism, the program takes care of those details for us so we don't have to.

Let's add the following code (which also uses polymorphism) to our program to print out each object:

```
// output each account
foreach (BankAccount account in accounts)
{
    Console.WriteLine(account);
}
```

C# automatically calls the `ToString` method when we call the `Console WriteLine` method with an object, so the above code uses the `ToString` method we provided in the `BankAccount` class. Of course, the `CheckingAccount` and `SavingsAccount` classes inherit the `ToString` method from the `BankAccount` class, so the method is actually called on the checking account and savings account objects in the list. The complete code for the polymorphism program we've developed in this section is provided in Figure 16.13 and the output from the program is shown in Figure 16.14.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BankAccounts
{
    /// <summary>
    /// Demonstrates polymorphism
    /// </summary>
    class Program
    {
        /// <summary>
        /// Demonstrates polymorphism
        /// </summary>
        /// <param name="args">command-line arguments</param>
        static void Main(string[] args)
        {
            // create list and add accounts
            List<BankAccount> accounts = new List<BankAccount>();
            accounts.Add(new CheckingAccount(100.00m));
            accounts.Add(new SavingsAccount(50.00m, 0.02m));
            accounts.Add(new CheckingAccount(300.00m));
        }
    }
}
```

```
accounts.Add(new SavingsAccount(500.00m, 0.02m));
accounts.Add(new CheckingAccount(1000.00m));
accounts.Add(new SavingsAccount(50000.00m, 0.02m));

// deposit $20 into each account
foreach (BankAccount account in accounts)
{
    account.MakeDeposit(20.00m);
}

// output each account
foreach (BankAccount account in accounts)
{
    Console.WriteLine(account);
}

Console.WriteLine();
}
```

**Figure 16.13.** Program.cs

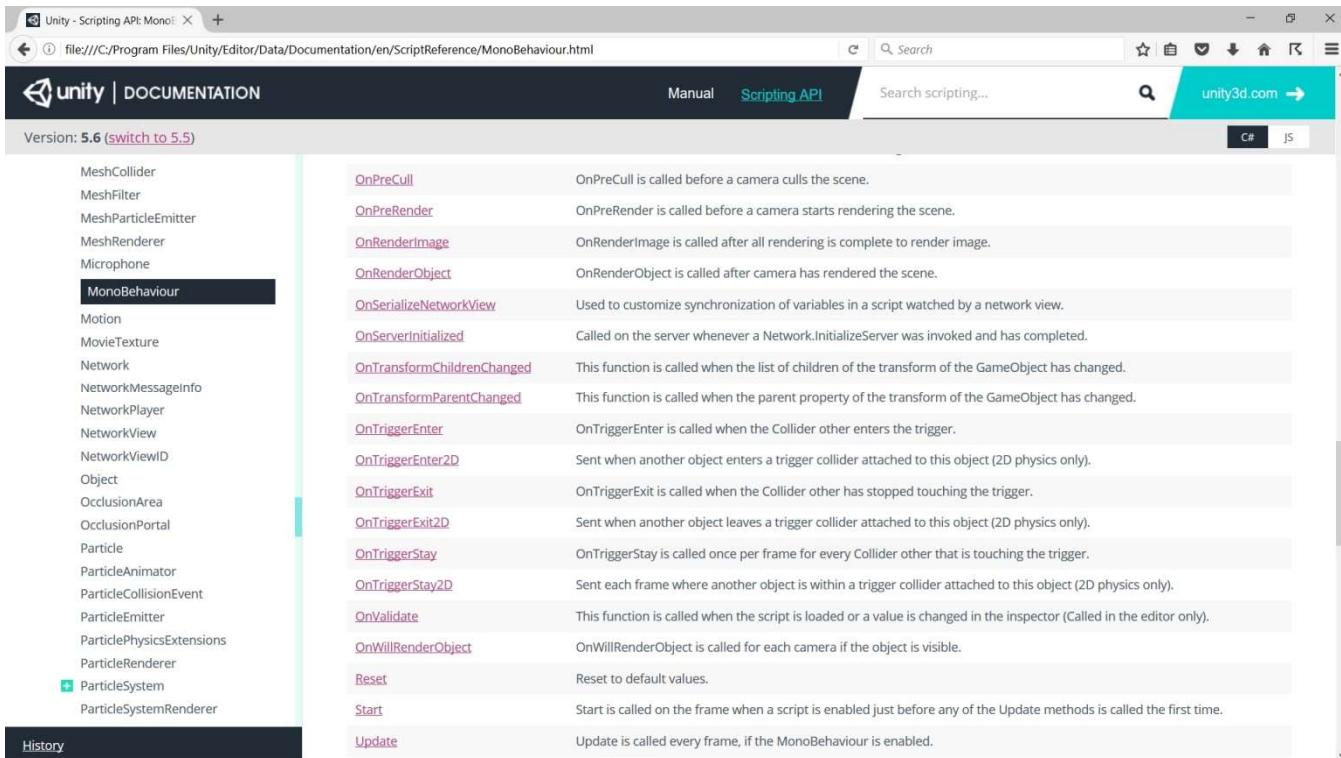


**Figure 16.14. Bank Account Polymorphism Program Output**

## 16.9. The MonoBehaviour Class

We've discussed in previous chapters that the C# scripts (classes) we add to our Unity games are, by default, child classes of the `MonoBehaviour` class. We've briefly talked about some basic inheritance concepts along the way, but now that we've covered inheritance in detail you should have a better understanding of how the inheritance works in our scripts.

All our scripts include the `Start` and `Update` methods in the template for the script. Figure 16.15. shows an excerpt from the Messages section of the MonoBehaviour documentation from the Unity Scripting Reference. As you can see at the bottom of the figure, the `MonoBehaviour` class has `Start` and `Update` methods; those are the methods we inherit in our new scripts.



**Figure 16.15. MonoBehaviour Documentation**

In previous chapters, we've also used the inherited `OnCollisionEnter2D`, `OnMouseDown`, `OnTriggerEnter2D`, and `OnTriggerStay2D` methods to implement the game functionality we needed. We've also accessed the inherited `gameObject` and `transform` fields as we've needed them. We've discovered that it's helpful to look through the MonoBehaviour documentation when we need to do something specific in our scripts, because lots of times there's already a method we've inherited from the `MonoBehaviour` class that we can use<sup>31</sup>.

## 16.10. Putting It All Together

Let's build a complete, though small, game using the inheritance concepts we've learned in this chapter. Here's the problem description:

Design and implement a game where the player moves the mouse over different teddy bears to destroy them. Green teddy bears will simply disappear (10 points), purple teddy bears will explode (25 points), and yellow teddy bears will burn (50 points). The game should spawn a random teddy bear every second.

### *Understand the Problem*

This a pretty straightforward problem to understand. Although the problem description doesn't say we need to display the current score, that's definitely something we'll want to do as a standard component of games with a score.

---

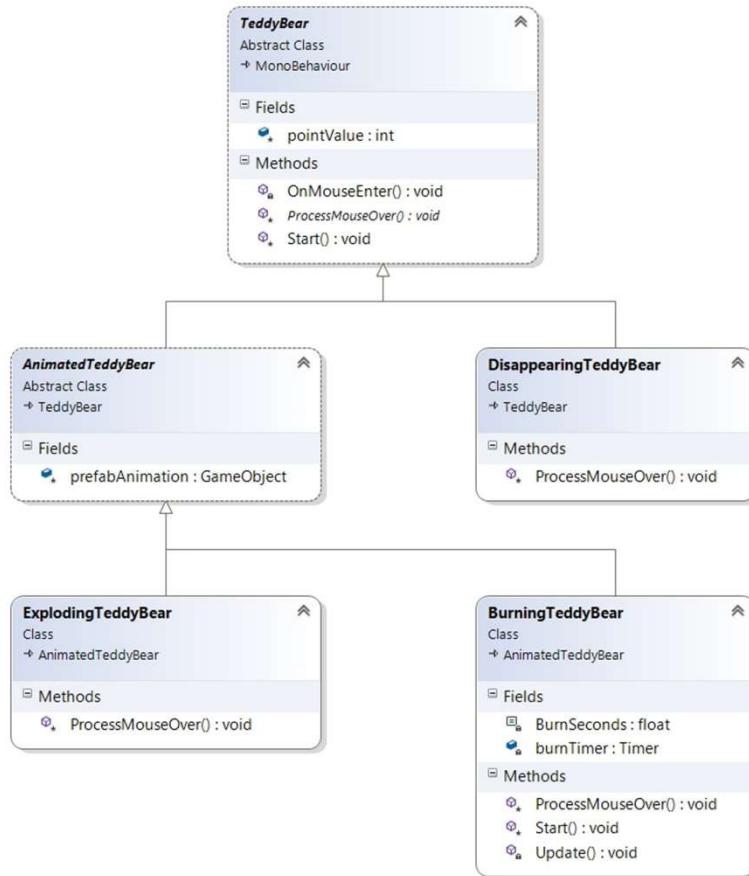
<sup>31</sup> Unity doesn't actually use inheritance to provide these methods to us in our child classes, but conceptually it works that way. If we implement one of those methods in the child class, that method is called when appropriate.

## Design a Solution

Let's think about blowing up and burning teddy bears first (we mean in the game, of course, not in real life). In previous problems, when we needed an explosion we used an `Explosion` prefab with an attached `Explosion` script that played the explosion animation. For our current problem, we're going to need both an explosion animation for the purple teddy bears and a fire animation for the yellow teddy bears. We'll build prefabs for both of our required animations, but we won't actually need a script for the fire animation; you'll see why when we get there.

What about the teddy bears? Well, we know we'll need general teddy bear behavior – like moving and bouncing off the edges of the window – for all the teddy bears, but each teddy bear has specialized behavior as well. Wow, this looks like a great opportunity for us to apply our inheritance understanding! And there was much rejoicing ...<sup>32</sup>

It also turns out that the exploding and burning teddy bears share behavior because each of them will have an animation that they may be either starting or playing. Those observations lead to the class hierarchy shown in Figure 16.16.



**Figure 16.16. Teddy Bear Class Diagram**

<sup>32</sup> The appropriate response here is a very insincere “Yaay”

We'll start looking at the code soon, but there's something you should notice about the `TeddyBear` class – it's identified as an *abstract class*. What's an abstract class? It's a class that serves as a parent class for one or more child classes but we can't actually instantiate objects of the class.

There's actually a very good reason for making `TeddyBear` an abstract class. We want to make sure all of the child classes include implementation of their required behavior (in the `ProcessMouseOver` method) when the mouse goes over them, but we don't know what the child classes have to do when the mouse goes over them. Using an abstract class lets us handle that in an elegant way that we'll discuss as we go through the `TeddyBear` code. We'll also discuss why `AnimatedTeddyBear` is an abstract class when we work through the code.

Because all our teddy bear types have a point value, we've included a `pointValue` field in the `TeddyBear` class for all the child classes to inherit. The `start` method will apply the impulse force required to get the teddy bear moving (behavior we need for all the teddy bears) and the `OnMouseEnter` method will call the child-specific `ProcessMouseOver` method when the mouse enters the collider for a teddy bear.

The `DisappearingTeddyBear` class implements the `ProcessMouseOver` method to make the teddy bear disappear.

For the `AnimatedTeddyBear` class, we add a `prefabAnimation` field for the animation (for explosions or fire) that will be played.

The `ExplodingTeddyBear` and `BurningTeddyBear` classes both need to implement the `ProcessMouseOver` method. The `BurningTeddyBear` class also overrides the `Start` method so it can start the burn timer (teddy bears don't burn forever – sigh) and also includes the `Update` method so it can tell when the teddy bear should be destroyed.

### *Write Test Cases*

We need to make sure our test plan covers all three of the *concrete* (in other words, not abstract) teddy bear classes: `DisappearingTeddyBear`, `ExplodingTeddyBear`, and `BurningTeddyBear`. In addition, we need to make sure the overall game works properly, spawning teddy bears as appropriate and keeping and displaying the score properly. The test cases listed below are designed to meet all those testing needs.

#### **Test Case 1**

##### **Checking Disappearing Teddy Bears**

Step 1. Input: Hard-coded spawning of disappearing teddy bears

Expected Result: Moving green teddy bears that bounce off the walls and each other properly

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear disappears, score increases by 10

This test case makes sure the `DisappearingTeddyBear` class is implemented properly.

## Test Case 2

### **Checking Exploding Teddy Bears**

Step 1. Input: Hard-coded spawning of exploding teddy bears

Expected Result: Moving purple teddy bears that bounce off the walls and each other properly

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear explodes, score increases by 25

Step 3. Input: Mouse over teddy bear near collision with other teddy bear

Expected Result: Teddy bear explodes, score increases by 25, other teddy bear doesn't collide with explosion

This test case makes sure the `ExplodingTeddyBear` class is implemented properly.

## Test Case 3

### **Checking Burning Teddy Bears**

Step 1. Input: Hard-coded spawning of burning teddy bears

Expected Result: Moving yellow teddy bears that bounce off the walls and each other properly

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear starts burning, score increases by 50

Step 3. Input: Mouse over teddy bear near collision with other teddy bear

Expected Result: Teddy bear starts burning, score increases by 50, teddy bears bounce off each other properly

This test case makes sure the `BurningTeddyBear` class is implemented properly.

## Test Case 4

### **Checking Random Spawning**

Step 1. Input: Hard-coded spawning of random teddy bears

Expected Result: Moving green, purple, and yellow teddy bears that bounce off the walls and each other properly, with new teddy bear every second

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear reacts properly, score increases properly

This test case makes sure the game is randomly spawning the teddy bears properly.

### *Write the Code*

As usual, let's work on our code a little at a time, moving through the steps of getting each test case to pass before moving on to the next chunk of code. If we start on Test Case 1 (which makes sense), we need to implement the `TeddyBear` and `DisappearingTeddyBear` classes. Since `TeddyBear` is the parent class we'll implement that class first.

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

/// <summary>
/// An abstract class for a teddy bear
/// </summary>
```

```
public abstract class TeddyBear : MonoBehaviour
{
```

As the UML diagram from our design indicates, we make the `TeddyBear` class an abstract class.

```
#region Fields

[SerializeField]
protected int pointValue;

#endregion
```

We mark the `pointValue` field with `[SerializeField]` so we can change it in the Inspector. Because our `DisappearingTeddyBear`, `ExplodingTeddyBear`, and `BurningTeddyBear` classes are all child classes (directly or indirectly) of the `TeddyBear` class, we'll be able to populate the field in the Inspector for all those scripts. The field is protected so the child classes can access the field without exposing the field to all the classes in the game.

```
/// <summary>
/// Use this for initialization
/// </summary>
virtual protected void Start()
{
    // apply impulse force to get teddy bear moving
    const float MinImpulseForce = 3f;
    const float MaxImpulseForce = 5f;
    float angle = Random.Range(0, 2 * Mathf.PI);
    Vector2 direction = new Vector2(
        Mathf.Cos(angle), Mathf.Sin(angle));
    float magnitude = Random.Range(MinImpulseForce, MaxImpulseForce);
    GetComponent<Rigidbody2D>().AddForce(
        direction * magnitude,
        ForceMode2D.Impulse);
}
```

The `Start` method applies the force to get the teddy bear moving; all the child classes inherit this method, so they all start moving. We marked the `Start` method as `virtual` so that child classes (specifically, the `BurningTeddyBear` class) can override the method as necessary. Also, by default the `Start` method is `private`. We changed it to `protected` so the `Start` method in the `BurningTeddyBear` class can call it to get the burning teddy bear moving.

How did we know at this point to mark the `Start` method `virtual` and `protected` as we were writing the `TeddyBear` class? That's a great question, because when we're implementing a parent class we don't necessarily know what child classes will be implemented and what they'll need access to.

Okay, confession time. We started with everything as `private` and when we discovered we needed access to a method (to override it, call it, or both) as we implemented our child classes, we came back and marked that method as appropriate. This is a really good approach to use because we didn't initially know which method(s) needed to be `protected` (and `virtual`). We're showing and discussing the (mostly) final `TeddyBear` code here rather than walking through all the iterations we went through as we developed it.

```

/// <summary>
/// Called when the mouse enters the collider
/// </summary>
void OnMouseEnter()
{
    ProcessMouseOver();
}

```

When the mouse enters the collider for any of the teddy bears, this method calls the `ProcessMouseOver` method. We'll discuss that method next.

```

#region Protected methods

/// <summary>
/// Processing for when the mouse is over the teddy bear
/// </summary>
protected abstract void ProcessMouseOver();

#endregion
}

```

Here's something new. We're defining an abstract method by providing the method header (including the `abstract` keyword) followed by a ; instead of a method body. What does this do for us? It forces any child class to override the `ProcessMouseOver` method with an actual implementation that includes a method body; that implementation is called a concrete method. Since most methods are actually concrete, we usually just call them methods.

There's actually an exception to the override discussion above. A child class can choose not to override the abstract method in the parent class, but in that case the child class also has to be abstract. Any class in the class hierarchy that can actually be instantiated will need to inherit or contain concrete implementations of all abstract methods in the hierarchy.

Why does forcing a child class to implement the `ProcessMouseOver` method help us? Remember, the `OnMouseEnter` method calls the `ProcessMouseOver` method when the mouse enters the collider for the teddy bear. Because we don't know what the child classes need to do when that happens, we have them provide their processing in their overridden `ProcessMouseOver` method. That way we can include detection of the mouse entering the teddy bear collider in the parent class, with each child class providing the custom processing for that child class when that happens. This is so totally awesome that you should take a deep breath or two before continuing.

Let's look at the `DisappearingTeddyBear` class next.

```

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

/// <summary>
/// A teddy bear that disappears when the mouse passes over it
/// </summary>
public class DisappearingTeddyBear : TeddyBear
{

```

As indicated in our design, `DisappearingTeddyBear` is a child class of `TeddyBear`.

```
#region Protected methods

/// <summary>
/// Destroys the teddy bear
/// </summary>
protected override void ProcessMouseOver()
{
    Destroy(gameObject);
}

#endregion
}
```

This method simply destroys the game object the script is attached to, which makes the game object disappear when the mouse enters the collider for the game object.

Before we can run Test Case 1, we need to create a prefab for a disappearing teddy bear, including Rigidbody 2D, Box Collider 2D, and `DisappearingTeddyBear` (the script) components. We also need to set the Point Value field in the script to 10 in the Inspector. Check out the Unity project for this problem if you want more details about that prefab.

We also need the game to start spawning disappearing teddy bears. We wrote a `TeddyBearSpawner` script and attached it to the main camera to handle this for us (as we've done for previous games).

### *Test the Code*

#### **Test Case 1**

##### **Checking Disappearing Teddy Bears**

Step 1. Input: Hard-coded spawning of disappearing teddy bears

Expected Result: Moving green teddy bears that bounce off the walls and each other properly

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear disappears, score increases by 10

Well heck. It's nice to see that the teddy bear disappears in Step 2 the way it's supposed to, but there's no score display and nothing in the `ProcessMouseOver` method above to increase the score, so the test case fails. Let's fix that now.

### *Write the Code, Part 2*

In our fish game in Chapter 14, we had our `Fish` script handle tracking and displaying the score. As we mentioned then, this made sense because the fish is the player's avatar in that game and it makes sense to have the player keep track of their own score. The player doesn't have an avatar in this game, though, so we should use a different approach here.

We've also seen in our previous solutions that it sometimes makes sense to have a high-level "game manager script" that handles game-level kinds of things (like in our Ted the Collector game). Since the score display is a game-level function, we'll write a new `TeddyBearDestruction` script to handle this.

Like we did for our fish game, we need to add a Text component to our scene; here are the instructions (again) for doing that. Right click in the Hierarchy window and select UI > Text. As you can see, you actually end up with a number of new components, including a Canvas that the text is drawn on in the game. Change the name of the Text component to ScoreText.

Select ScoreText in the Hierarchy window, select the Anchor Preset for the top left corner, and change the Pos X and Pos Y values in the Rect Transform component to position the text (we used 150 and -80 for these values). In the Text (Script) component, change the Font Style to Bold, the Font Size to 24, and the Color to white.

Here's our `TeddyBearDestruction` script:

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

/// <summary>
/// Game manager
/// </summary>
public class TeddyBearDestruction : MonoBehaviour
{
    // score support
    [SerializeField]
    Text scoreText;
    int score = 0;

    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
    {
        // set initial score text
        scoreText.text = "Score: " + score;
    }

    /// <summary>
    /// Adds the given points to the score
    /// </summary>
    /// <param name="points">points to add</param>
    public void AddPoints(int points)
    {
        score += points;
        scoreText.text = "Score: " + score;
    }
}
```

Next, we attach this script to the main camera. After doing so, drag the ScoreText component from the Hierarchy window onto the Score Text field of the script in the Inspector.

Now we need a way for the `DisappearingTeddyBear` script to call the `TeddyBearDestruction` `AddPoints` method. One good way to do that is to get a reference to the `TeddyBearDestruction` script that's attached to the main camera. We should realize, though, that all of our teddy bear classes will need

a reference to this script. That means that it makes sense to put the field for this reference, and setting that field to a value, in the `TeddyBear` class rather than in each of the child classes.

This really only requires that we make two changes. First, we add the required field to the `TeddyBear` class:

```
// score support
protected TeddyBearDestruction teddyBearDestruction;
```

Notice that we make the field `protected` so the child classes can access the field to call the `AddPoints` method.

Second, we give the field its value in the `TeddyBear Start` method:

```
// score support
teddyBearDestruction = Camera.main.GetComponent<TeddyBearDestruction>();
```

The last thing we need to do is add code to the `DisappearingTeddyBear ProcessMouseOver` method to add points to the score:

```
teddyBearDestruction.AddPoints(pointValue);
```

### *Test the Code, Part 2*

When we run Test Case 1 again, the test case works as expected, so we can move on.

### *Write the Code, Part 3*

To get Test Case 2 to pass, we need to implement the `AnimatedTeddyBear` and `ExplodingTeddyBear` classes. Since `AnimatedTeddyBear` is the parent class we'll implement that class first.

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

/// <summary>
/// A teddy bear with an animation
/// </summary>
public abstract class AnimatedTeddyBear : TeddyBear
{
```

This is the first time we've seen a child class that's also abstract; that works fine, though. Remember that we said that classes that will actually be instantiated need to contain concrete implementations of all the abstract methods of all its parent classes in the class hierarchy, but it's certainly okay to have multiple abstract classes in that hierarchy.

```
#region Fields

[Serializable]
protected GameObject prefabAnimation;

#endregion
```

```
}
```

We mark the `prefabAnimation` field with `[SerializeField]` so we can populate it in the Inspector and we make it `protected` so child classes can access it as necessary.

Even though this is a very simple class, it inherits everything from the `TeddyBear` class and adds a field that we know will be useful for the teddy bear classes that need to play an animation.

Let's take a look at the `ExplodingTeddyBear` class next.

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

/// <summary>
/// An exploding teddy bear
/// </summary>
public class ExplodingTeddyBear : AnimatedTeddyBear
{
    #region Protected methods

    /// <summary>
    /// Explodes the teddy bear
    /// </summary>
    protected override void ProcessMouseOver()
    {
        teddyBearDestruction.AddPoints(pointValue);
        Instantiate(prefabAnimation, transform.position, Quaternion.identity);
        Destroy(gameObject);
    }

    #endregion
}
```

The method above overrides the abstract method from the `TeddyBear` class. Even though the `TeddyBear` class isn't the parent class (`AnimatedTeddyBear` is), `TeddyBear` is the grandparent of the `ExplodingTeddyBear` class. Remember, concrete classes have to implement all abstract methods in the classes above them in the class hierarchy.

The method adds the points for the exploding teddy bear, instantiates the `prefabAnimation` (which will be the explosion prefab), and destroys the teddy bear game object.

Before we can run Test Case 2, we need to create a prefab for an exploding teddy bear, including Rigidbody 2D, Box Collider 2D, and `ExplodingTeddyBear` (the script) components. We also need to create a prefab for an explosion and populate the Point Value field in the script with 25 and the Prefab Animation field in the script with the explosion prefab. Finally, we change the `TeddyBearSpawner` script to only spawn exploding teddy bears.

*Test the Code, Part 3***Test Case 3****Checking Exploding Teddy Bears**

Step 1. Input: Hard-coded spawning of exploding teddy bears

Expected Result: Moving purple teddy bears that bounce off the walls and each other properly

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear explodes, score increases by 25

Step 3. Input: Mouse over teddy bear near collision with other teddy bear

Expected Result: Teddy bear explodes, score increases by 25, other teddy bear doesn't collide with explosion

The test case works as expected, so we can move on.

*Write the Code, Part 4*

To get Test Case 3 to pass, we need to implement the `BurningTeddyBear` class. Before we look at the details of that class, we have some work to do in the Unity editor.

First, we need to create a prefab for a burning teddy bear, including Rigidbody 2D, Box Collider 2D, and `BurningTeddyBear` (the script) components. We start by dragging the burningteddybear sprite from the sprites folder in the Project window into the Hierarchy window and adding the components listed above. Change the name of the game object to `BurningTeddyBear`.

Next, we create a prefab for the fire we need; we do this by following the same steps we did to create the `Explosion` prefab, but you shouldn't include the first few frames of the fire sprite strip because we don't need those "startup" frames in our looping fire animation. We don't actually need to attach a script to the fire prefab. We needed the `Explosion` script to destroy the `Explosion` game object once the explosion animation finished, but our fire animation will just keep playing until the Fire object is destroyed.

After creating the Fire prefab, drag it into the Hierarchy window and drop it onto the `BurningTeddyBear` game object in the Hierarchy window. This makes that object a child game object (not to be confused with a child class) of the `BurningTeddyBear` game object; we used the same approach in Chapter 14 for the circle problem. We want the fire to be a child game object so that it follows the teddy bear around as it burns.

At this point, we need to make a couple of adjustments to the Fire child game object. First, we should shift it up so the fire appears at the top of the teddy bear's head. To do this, select the Fire child game object in the Hierarchy window and change the Y value in the Position field of the Transform component to 0.28.

Also, you may have noticed that the fire appears in front of the teddy bear's head. It actually looks better if the fire appears behind the teddy bear's head, so change the Order in Layer field of the Sprite Renderer component to -1 (recall that we used Order in Layer in Chapter 9 as well).

Finally, we don't want the fire animation to be visible when the burning teddy bear is spawned, we only want to make it visible once the teddy bear starts burning. Uncheck the check box just to the left of the Sprite Renderer title at the top of the Sprite Renderer component. That disables the component so it doesn't actually get drawn.

Apply the changes we made to the Fire prefab and save the BurningTeddyBear game object as a prefab.

Here's the `BurningTeddyBear` code:

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

/// <summary>
/// A burning teddy bear
/// </summary>
public class BurningTeddyBear : AnimatedTeddyBear
{
    #region Fields

    // burn support
    Timer burnTimer;
    const float BurnSeconds = 2f;

    #endregion
}
```

We're going to have the teddy bear burn for 2 seconds before it's destroyed; we'll use the fields above to support that.

```
#region Public methods

/// <summary>
/// Use this for initialization
/// </summary>
override protected void Start()
{
    // create burn timer
    burnTimer = gameObject.AddComponent<Timer>();
    burnTimer.Duration = BurnSeconds;

    // start teddy bear moving
    base.Start();
}
```

Recall that we marked the `Start` method in the `TeddyBear` class to be virtual so we could override it here. That overriding means that this `start` method will get called instead of the one in the `TeddyBear` class for game objects that have the `BurningTeddyBear` script attached to them.

We needed to override the method so we could create the burn timer. We check whether or not the burn timer is finished in the `Update` method (coming soon), so we need to create it before then to avoid a `NullReferenceException` when we access its `Finished` property.

We still need to start the teddy bear moving, though, so we also need to run the `TeddyBear Start` method. That's what the last line of code above does. We use the `base` keyword when we need to call a method in the parent class from the child class.

```
/// <summary>
/// Update is called once per frame
/// </summary>
```

```

void Update()
{
    // check for burn complete
    if (burnTimer.Finished)
    {
        Destroy(gameObject);
    }
}

#endregion

```

The `Update` method destroys the game object the script is attached to when the burn timer finishes. That also destroys all child game objects, so both the `BurningTeddyBear` and the `Fire` game objects are destroyed when that happens.

```

#region Protected methods

/// <summary>
/// Burns the teddy bear
/// </summary>
protected override void ProcessMouseOver()
{
    teddyBearDestruction.AddPoints(pointValue);

    // make fire animation visible
    SpriteRenderer fireRenderer =
        prefabAnimation.GetComponent<SpriteRenderer>();
    fireRenderer.enabled = true;

    // start burn timer
    burnTimer.Run();
}

#endregion
}

```

The `ProcessMouseOver` method above adds the points for the burning teddy bear to the score. It then retrieves a reference to the `SpriteRenderer` component for the fire prefab (which the `prefabAnimation` field is holding) and enables the sprite renderer so the fire animation is displayed. Finally, the method starts the burn timer so the teddy bear will be destroyed once the burn timer is finished.

Finally, we populate the `Point Value` field in the script with 50 and the `Prefab Animation` field in the script with the fire prefab (the child game object in the Hierarchy window, not the prefab from the Project window), apply those changes to the `BurningTeddyBear` prefab, and change the `TeddyBearSpawner` script to only spawn burning teddy bears.

*Test the Code, Part 4*

### **Test Case 3**

#### **Checking Burning Teddy Bears**

Step 1. Input: Hard-coded spawning of burning teddy bears

Expected Result: Moving yellow teddy bears that bounce off the walls and each other properly

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear starts burning, score increases by 50

Step 3. Input: Mouse over teddy bear near collision with other teddy bear

Expected Result: Teddy bear starts burning, score increases by 50, teddy bears bounce off each other properly

This test case also passes, but we actually discovered a problem while we were running the test case. If we start a teddy bear burning, then pass the mouse over it while it's burning, we earn an additional 50 points (and we can do that multiple times). This isn't valid behavior, but the test case didn't detect it, so our first step is to revise the test case appropriately.

*Write Test Cases, Part 2*

### **Test Case 3**

#### **Checking Burning Teddy Bears**

Step 1. Input: Hard-coded spawning of burning teddy bears

Expected Result: Moving yellow teddy bears that bounce off the walls and each other properly

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear starts burning, score increases by 50

Step 3. Input: Mouse over teddy bear near collision with other teddy bear

Expected Result: Teddy bear starts burning, score increases by 50, teddy bears bounce off each other properly

Step 4. Input: Mouse over teddy bear that's already burning

Expected Result: Teddy bear continues burning, score doesn't change

*Write the Code, Part 5*

The cleanest way to solve this problem is to check to make sure the burn timer isn't already running before we do the processing in the `ProcessMouseOver` method. Here's the revised method:

```
/// <summary>
/// Burns the teddy bear
/// </summary>
protected override void ProcessMouseOver()
{
    if (!burnTimer.Running)
    {
        teddyBearDestruction.AddPoints(pointValue);

        // make fire animation visible
        SpriteRenderer fireRenderer =
            prefabAnimation.GetComponent<SpriteRenderer>();
        fireRenderer.enabled = true;

        // start burn timer
        burnTimer.Run();
    }
}
```

*Test the Code, Part 5*

Our revised Test Case 3 passes, so we can write the final piece of code for our problem solution.

*Write the Code, Part 6*

To make our solution pass Test Case 4, the only thing we need to add is random spawning of teddy bears to the game. The `SpawnBear` method from the `TeddyBearSpawner` script is shown below; this method is run each time the spawn timer is finished.

```
/// <summary>
/// Spawns a new teddy bear at a random location
/// </summary>
void SpawnBear()
{
    // generate random location
    Vector3 location = new Vector3(Random.Range(minSpawnX, maxSpawnX),
        Random.Range(minSpawnY, maxSpawnY),
        -Camera.main.transform.position.z);
    Vector3 worldLocation = Camera.main.ScreenToWorldPoint(location);

    // spawn random teddy bear type at location
    GameObject teddyBear;
    int typeNumber = Random.Range(0, 3);
    if (typeNumber < 1)
    {
        teddyBear = Instantiate(prefabDisappearingTeddyBear) as GameObject;
    }
    else if (typeNumber < 2)
    {
        teddyBear = Instantiate(prefabExplodingTeddyBear) as GameObject;
    }
    else
    {
        teddyBear = Instantiate(prefabBurningTeddyBear) as GameObject;
    }
    teddyBear.transform.position = worldLocation;
}
```

This code generates a random location, then randomly generates a 0, 1, or 2 – the `Random` `Range` method we’re using here takes the inclusive lower bound and exclusive upper bound for the range of numbers – and instantiates a `DisappearingTeddyBear`, `ExplodingTeddyBear`, or `BurningTeddyBear` game object based on the generated number.

*Test the Code, Part 6***Test Case 4****Checking Random Spawning**

Step 1. Input: Hard-coded spawning of random teddy bears

Expected Result: Moving green, purple, and yellow teddy bears that bounce off the walls and each other properly, with new teddy bear every second

Step 2. Input: Mouse over teddy bear

Expected Result: Teddy bear reacts properly, score increases properly

That finishes off our solution to this problem. This is obviously a very simple game, but it gave us a chance to really hone our skills using inheritance.

## 16.11. Common Mistakes

### *Forgetting to Call Parent Class Constructor*

In a child class constructor, you need to either call the constructor in the parent class or explicitly initialize all the fields in the parent class and the child class yourself. If you're getting strange behavior in the child class, you may have assumed the parent class constructor ran without your calling it.

### *Using Infinite Recursion in an Overridden Method*

This can be a hard one to find. Say we've overridden a method called `MakeDeposit` in a child class of the `BankAccount` class, but we also want to call the `MakeDeposit` method in the `BankAccount` class from our child class method. We need to be sure we use `base.MakeDeposit` for the call rather than simply using `MakeDeposit`. The latter approach will just keep calling the same method in the child class (this is called recursion) until the program runs out of memory.

### *Trying to Directly Access a Private Field*

This occurs when a child class tries to directly access a field that's `private` in the parent class (remember, this is the default, so if there is no access modifier, it's `private`). The compiler will give us an error message for this. The typical solution is to change the access modifier for the field in the parent class to `protected`, but that's not always the right answer, so you should think about it and make sure you're doing the right thing.

# Chapter 17. Delegates and Event Handling

We said a number of times in previous chapters that we should use delegates and event handling as part of some of our problem solutions. We said, though, that we'd put off learning about those constructs until later in the book. Well ... it's later!

Delegates in C# are very useful in a number of ways. For example, they let us easily specify specific behavior for instances of a more general class. In addition, they let us build a robust structure for handling events that occur in our games. This chapter looks at how we can use C# delegates for those two purposes. We'll also explore how to use different versions of [UnityEvent](#), a set of built-in Unity classes that lets us implement some Unity-specific event handling.

## 17.1. What Are Delegates?

The first thing we need before actually using delegates is an understanding of what they are. According to the Delegates (C# Programming Guide) topic in the Visual Studio (VS) help:

“A `delegate` is a type that defines a method signature. When you instantiate a delegate, you can associate its instance with any method with a compatible signature. You can invoke (or call) the method through the delegate instance.”

Hmmm, does that clarify everything for us? Perhaps not quite! Let's look at delegates in a slightly different way, then come back to this definition in a little while. We'll start with an idea that you should already have a firm grasp on: value and reference types.

As you know, variables that are declared as a reference type don't hold the actual object for the reference type; instead, they hold a reference to that object's actual location in memory. So a reference type variable holds the memory address for the object.

In some languages (like C and C++), a reference to a memory address is called a pointer. In fact, those languages also support something called “function pointers.” As you might suspect, a function pointer is a pointer to a function rather than to an object. Why does that help? Because it means we can pass function pointers as arguments to methods (among other things), essentially passing behavior specifications along so other methods can use them. But C and C++ aren't C#, so why do we care? Because, as the VS help says:

“Delegates are like C++ function pointers but are type safe.”

So we get the benefit of function pointers in C# as well (type safety helps make them even better, which we'll discuss soon). One more quote from the VS help:

“Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates.”

Those are precisely the two uses we mentioned in the introduction, so let's look at each of them.

## 17.2. Customizing Behavior With Delegates

Let's build a small game that lets us use delegates to specify the behavior for various balls in the game. We're using delegates because we don't want to bother specifying a bunch of `Ball` child classes with different behavior, although that is of course another approach we could use here.

When the mouse intersects the ball, the ball will do whatever the delegate says to do. You should be sure to download the code from the web site so you can follow along.

The first thing we'll do is define the delegate; you can find the following definition in the `BallBehavior.cs` file in the scripts folder of the Unity project:

```
/// <summary>
/// Delegate for ball behavior
/// </summary>
/// <param name="ball">the ball</param>
public delegate void BallBehavior(Ball ball);
```

As the first definition from VS help said, we're defining the method signature for the delegate. When we define a method to use for the delegate, we'll have to make sure that the return type for the method is `void` and that the method has a single `Ball` parameter. The access modifier and method names DON'T have to match. In fact, the method name (`BallBehavior`) in the delegate specification is actually the name of the type for the delegate (just as a class name for a class we define is the name of the type for that class).

We only have a single field defined in the `Ball` class:

```
#region Fields

// method specifying ball behavior
BallBehavior behavior;

#endregion
```

Notice that the type of the variable is the delegate type we specified when we declared the delegate.

We need to expose a property so a consumer of the `Ball` class (for this example, a ball spawner) can specify the ball behavior for an instance of the class:

```
#region Properties

/// <summary>
/// Sets the ball behavior
/// </summary>
/// <value>ball behavior</value>
public BallBehavior Behavior
{
    set { behavior = value; }
}
```

You should recall from previous chapters that when we want to change the sprite that's rendered for a particular game object, we set the sprite for the `SpriteRenderer` component for that game object; we can do a similar thing for color. Because we want balls with different behaviors to be different colors, we also need to expose a `Color` property so the ball spawner can set the color for the ball to be the correct color.

```
/// <summary>
/// Sets the ball color
/// </summary>
/// <value>ball color</value>
public Color Color
{
    set
    {
        SpriteRenderer spriteRenderer =
            GetComponent<SpriteRenderer>();
        spriteRenderer.color = value;
    }
}
```

In our `SpriteRenderer` examples in previous chapters, we saved the component in a field because we expected to access that component multiple times over the life of the object. In our current example, though, we only expect the color to be changed once (right after the Ball game object is created), so we use a local variable in the set accessor instead.

As in our example from the previous chapter where we had different teddy bears with different behaviors, we'll use the `OnMouseEnter` method to trigger the appropriate behavior when the mouse intersects with the game object.

```
/// <summary>
/// Called when the mouse enters the collider
/// </summary>
void OnMouseEnter()
{
    behavior(this);
}
```

This should remind you of our example in the previous chapter, where we had an abstract `ProcessMouseOver` method that we called from the `OnMouseEnter` method. The key difference is that in the previous chapter, different child classes implemented the `ProcessMouseOver` method to implement their specific (different) behavior, leading to a class hierarchy of multiple classes. In our example here, we have a single `Ball` class, and the different behaviors of the different Ball game objects are determined by the `BallBehavior` delegate that was used to set the `behavior` field for each Ball game object.

The `Ball` class has no idea what the `behavior` method does, because it was provided at run time when the `Behavior` property was accessed to set that field, but we do know that the method being called requires a single `Ball` argument. We provide `this` for that argument so that whatever the method does it will do to this `Ball`.

That's all well and good, but it doesn't actually demonstrate how we can effectively use delegates to give balls different behavior. To see that, we need to create a script to actually spawn the balls.

Before we do that, though, we need to create a Ball prefab the spawner can spawn. Add a sprite for a white ball to the project and drag the sprite into the Hierarchy window. Because the `OnMouseEnter` method gets called based on when the mouse enters the collider for a game object, click the Add Component button in the Inspector and select Physics 2D > Circle Collider 2D. Add the `Ball` script to the Ball game object in the Hierarchy window, create a prefabs folder in the Project window, and drag the Ball game object from the Hierarchy window onto the prefabs folder to create the prefab. Finally, delete the Ball game object from the Hierarchy window.

As usual, we'll need to use a timer for our spawner, so go to your Operating System and copy the `Timer.cs` file from one of the previous Unity projects into the scripts folder for this example. Okay, we're finally ready to start working on our `BallSpawner` script. Lots of the code in this script looks like the `TeddyBearSpawner` script from the previous chapter, so we'll only look at the major differences here.

Instead of picking a random teddy bear type to spawn, we'll pick a random color (and the associated behavior) when it's time to spawn a new ball. We'll start by just spawning red balls, which will move left, then add the other colors (and behaviors) after we get that working. We know, though, that one of the things we need to do when we spawn a new ball is set the `Behavior` property with a `BallBehavior` for the ball. Here's a method that moves the ball to the left:

```
/// <summary>
/// Moves the given ball to the left
/// </summary>
/// <param name="ball">ball to move</param>
void MoveLeft(Ball ball)
{
    Vector3 position = ball.transform.position;
    position.x -= BallMoveAmount;
    ball.transform.position = position;
}
```

At this point in the book, you should be able to easily understand how the code in the method body works (we declared a `BallMoveAmount` constant in our script). The important point about this method is that it matches the requirements to be used as a `BallBehavior` delegate; specifically, it returns `void` and has a single `Ball` parameter. Remember, the access modifier and method names DON'T have to match our `BallBehavior` definition.

How do we use the `MoveLeft` method to create a red ball that moves to the left? Here's a `SpawnBall` method that does that:

```
/// <summary>
/// Spawns a new ball at a random location
/// </summary>
void SpawnBall()
{
```

```

// generate random location
Vector3 location = new Vector3(Random.Range(minSpawnX, maxSpawnX),
    Random.Range(minSpawnY, maxSpawnY),
    -Camera.main.transform.position.z);
Vector3 worldLocation = Camera.main.ScreenToWorldPoint(location);

// spawn red ball at location
GameObject ball = Instantiate(prefabBall) as GameObject;
Ball ballScript = ball.GetComponent<Ball>();
ballScript.Color = Color.red;
ballScript.Behavior = MoveLeft;
ball.transform.position = worldLocation;
}

```

The first block of code generates a random location for the ball; we've seen that code before, but the second block of code is more interesting.

The first two lines of code in that block instantiate a new instance of the Ball prefab and get the `Ball` script attached to that prefab so we can access the properties in the script. The third line of code changes the color of the ball using the built-in Unity `Color` enumeration and the fifth line of code moves the new Ball game object to the random location we generated in the first block of code.

The fourth line of code is the one that uses the delegate concepts we're exploring in this section. Recall that setting the `Behavior` property requires that we include a `BallBehavior` delegate on the right of the `=` because the type of the property is `BallBehavior`. Because our `MoveLeft` method matches the requirements to be used as a `BallBehavior` delegate, we can set the `Behavior` property to the `MoveLeft` method. And that's all we have to do to make this ball move to the left when the mouse intersects with the collider for the ball!

Attach the `BallSpawner` script to the main camera in the scene, drag the Ball prefab from the Project window onto the Prefab Ball value in the script in the Inspector, and run the game. You should see red balls spawned periodically, and if you move the mouse over a ball you should see it move to the left.

All we have to do to finish this example is write `MoveRight`, `MoveUp`, and `MoveDown` methods to implement the other 3 ball behaviors and change the second block of code in the `SpawnBall` method to randomly pick between the 4 different ball colors (and behaviors). We'll assume you can easily write the 3 additional behavior methods; here's the revised second block of code for the `SpawnBall` method:

```

// spawn random ball type at location
GameObject ball = Instantiate(prefabBall) as GameObject;
Ball ballScript = ball.GetComponent<Ball>();
int typeNumber = Random.Range(0, 4);
if (typeNumber < 1)
{
    ballScript.Color = Color.red;
    ballScript.Behavior = MoveLeft;
}
else if (typeNumber < 2)
{
    ballScript.Color = Color.green;
    ballScript.Behavior = MoveRight;
}

```

```

else if (typeNumber < 3)
{
    ballScript.Color = Color.blue;
    ballScript.Behavior = MoveUp;
}
else
{
    ballScript.Color = Color.yellow;
    ballScript.Behavior = MoveDown;
}
ball.transform.position = worldLocation;

```

Now when you run the game, you should get all 4 different color balls with red balls moving left, green balls moving right, blue balls moving up, and yellow balls moving down.

It takes a while for some new programmers to really understand how delegates work, but if you review this section as needed you'll find that delegates (and events, coming up next!) are really powerful and useful.

### 17.3. Events and Event Handling

The other really useful thing we can use delegates for is to implement an event system in our game. In other words, objects can invoke specific events when particular conditions occur in the game. Other objects that are listening for those events will “hear” them when they’re invoked and do anything they need to in response to that event.

To see why this is a help, let’s refactor our fish game from Chapter 14. In that chapter, we had the Fish game object keep track of and display the player’s score. That made sense in the context of that example (where our focus was on learning how to do text output in the game), but it’s more common to actually have a Heads Up Display (HUD) that displays information to the player. This becomes even more appropriate as we display more information than score (like health, timers, and so on) to the player, so let’s learn how to do that now. Of course, we’ll use an event and an event handler as part of our solution.

Starting from the Unity project from Chapter 14, create an empty game object by right clicking in the Hierarchy window and selecting Create Empty. Rename the new game object HUD, drag the Canvas and Event System onto the HUD game object, and make sure the X and Y values of the Transform component are both set to 0.

If you run the game now, you’ll see that it still works the way it used to. The problem, though, is that the `Fish` script has a reference to the ScoreText Text component inside the HUD. This is a problem because the `Fish` script really shouldn’t “know about” components within other game objects in the game.

One approach we could use to have the HUD handle displaying the score would be to write a `HUD` script that exposes a static method that the `Fish` script calls to tell the `HUD` to add points to the score. There’s a real problem with this approach, though. For this approach to work, the `Fish` script has to know about the `HUD` script and the methods it exposes. A better object-oriented design would have the `Fish` script totally unaware of the existence of the `HUD` script. It is just a fish, after all, so it shouldn’t have to understand how it fits into the larger game implementation!

Our design and implementation will have the `Fish` script invoke an event when the score changes (without having to care about who might be listening) and have the HUD listen for that event and change the text output when it "hears" that the event occurred. With this approach, we have a solid object-oriented design that works well (and uses delegates).

Let's start by looking at the `PointsAddedEvent.cs` file. You should know that there are a number of ways to structure events in C#; we're showing you the way we do it for all our events in our company games. Here's the code:

```

/// <summary>
/// Delegate for handling the event
/// </summary>
/// <param name="points">the points to add</param>
public delegate void PointsAddedEventHandler(int points);

/// <summary>
/// An event that indicates that points have been added
/// </summary>
public class PointsAddedEvent
{
    // the event handlers registered to listen for the event
    event PointsAddedEventHandler eventHandlers;

    /// <summary>
    /// Adds the given event handler as a listener
    /// </summary>
    /// <param name="handler">the event handler</param>
    public void AddListener(PointsAddedEventHandler handler)
    {
        eventHandlers += handler;
    }

    /// <summary>
    /// Invoke the event for all event handlers
    /// </summary>
    /// <param name="points">the points to add</param>
    public void Invoke(int points)
    {
        if (eventHandlers != null)
        {
            eventHandlers(points);
        }
    }
}

```

Up until this point, we've always only had one class in each file we create. In contrast, the `PointsAddedEvent.cs` file contains both the delegate for handling the event and the class for the event itself. We think it's better to keep both of those in a single file, so that's how we do it.

The delegate specification is similar to the delegate we discussed in the previous section, so it doesn't need any further explanation. The `PointsAddedEvent` class, however, merits a closer look.

The first thing we see in the class is a private `eventHandlers` field defined as the delegate type. It probably seems strange, though, that the variable declaration also includes the `event` keyword. What does that do?

Marking the variable as an `event` indicates that it's "a special kind of multicast delegate".

That means that the `eventHandlers` field can actually hold multiple event handlers (kind of like a list of the delegates, though simpler to interact with). This is useful when you might have multiple objects listening for the same event to be invoked; using an `event` field ensures that all of the listeners will hear the event.

The `AddListener` method lets an object that wants to listen for this event add a method to the set of event handlers (delegates) that will be notified when the event is fired. Using `+=` simply adds the handler parameter to the set of delegates that will be called when the event is fired.

The `Invoke` method goes through the set of event handlers that have been added as listeners and calls the delegate that was provided when that event handler was added as a listener. In other words, this part works exactly like the `MoveLeft`, `MoveRight`, `MoveUp`, and `MoveDown` delegates getting called in the `Ball` `OnMouseEnter` method. The only real difference is that each delegate isn't held in a unique variable like our `behavior` field in the `Ball` class; instead, our delegates here are bundled together in the `eventHandlers` field in the `PointsAddedEvent` class.

So the general approach will be that an object that invokes the event (an instance of the `Fish` class, in this case) will declare a field for a `PointsAddedEvent` object. Objects that want to listen for that event (an instance of the `HUD` class, in this case) will call the `AddListener` method to provide a method (delegate) that gets called when the event is invoked. When the object actually invokes the event, it calls each of the delegates that have been added as listeners so they can do whatever they need to do to process the event.

Okay, let's look at how this is all implemented in the example code, starting with the `Fish` class. First, we see a field for the event (note that we need to construct a new instance of the event class to use it):

```
// events fired by the class
PointsAddedEvent pointsAddedEvent = new PointsAddedEvent();
```

We also expose a method that lets other objects add listeners for this event:

```
/// <summary>
/// Adds the given event handler as a listener
/// </summary>
/// <param name="handler">the event handler</param>
public void AddListener(PointsAddedEventHandler handler)
{
    pointsAddedEvent.AddListener(handler);
}
```

This method makes it easy for other classes to add their event handlers as listeners.

Next, we need to create a `HUD` script that listens for the `PointsAddedEvent` so it can update the score when appropriate. Given that statement, though, we immediately realize that the `HUD` script would need

to know about the `Fish` script so it could call the `AddListener` method in the `Fish` script. This is just as bad as the `Fish` script having to know about the `HUD` script in the possible implementation approach we rejected!

This is unfortunately one of those areas where we need to slightly break the purity of our object-oriented approach to make our game work in a reasonable way. We'll solve this problem by implementing a static `EventManager` class that exposes an `AddListener` method for each of the events that can be invoked by scripts in the game. When a script calls one of the `EventManager` `AddListener` methods, the event manager will call the `AddListener` method on the script that actually invokes that event. Essentially, the `EventManager` class implements a wrapper around all the `AddListener` methods exposed by the scripts in the game. This is a win because each script only needs to know about the single `EventHandler` class no matter how many events they need to listen for.

You could certainly argue that if we're not going to implement a "pure" object-oriented solution, we should have just ignored all this event handling stuff and implemented an "ugly but simple" static `AddPoints` method in the `HUD` script! That's not the right way to think about this, though. Imagine a more complicated game where the `HUD` script needs to listen for events invoked by a number of different scripts in the game. In our chosen solution, the `HUD` script still only needs to know about the `EventManager` class, it doesn't need to know about all the other scripts that invoke the events it needs to listen for. We've had to sacrifice a little purity in the name of practicality, but our solution scales up to higher levels of script interaction complexity very cleanly, because all the mappings between listener and event invoking scripts are contained in a single class: the `EventManager` class.

Okay, here's the `EventManager` class, which only needs to expose a single `AddListener` method for our simple example:

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

/// <summary>
/// Manages connections between event listeners and event invokers
/// </summary>
public static class EventManager
{
    /// <summary>
    /// Adds the given event handler as a listener
    /// Game objects tagged as Fish invoke PointsAddedEvent
    /// </summary>
    /// <param name="handler">the event handler</param>
    public static void AddListener(PointsAddedEventHandler handler)
    {
        // add listener to all fish
        GameObject[] fish = GameObject.FindGameObjectsWithTag("Fish");
        foreach (GameObject currentFish in fish)
        {
            Fish script = currentFish.GetComponent<Fish>();
            script.AddListener(handler);
        }
    }
}
```

First, the code finds all the game objects in the scene that have been tagged with the Fish tag; those are the game objects that will have a `Fish` script attached to them. The code then retrieves the `Fish` script from each of those game objects and calls the `AddListener` method to add the provided event handler as a listener.

In this example, there's obviously only a single fish, but in more complicated examples there may be multiple game objects that invoke a particular event. For example, if we had teddy bears invoke the event to add points when they've been eaten, we'd need to add the listener to each of those teddy bears. Even though it's a bit of overkill for this example, finding all the game objects with the given tag is a general approach that should always work.

Of course, we need to add the Fish tag to all the game objects that have a `Fish` script attached to them for that approach to work. Add a Fish tag to the Fish prefab in the Project folder, which automatically tags all (in this case, one) instances of that prefab in the scene.

Next, we implement the `HUD` script:

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

/// <summary>
/// The HUD for the game
/// </summary>
public class HUD : MonoBehaviour
{
    #region Fields

    // score support
    [SerializeField]
    Text scoreText;
    int score = 0;

    #endregion

    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
    {
        // add listener for PointsAddedEvent
        EventManager.AddListener(HandlePointsAddedEvent);

        // initialize score text
        scoreText.text = "Score: " + score;
    }

    #region Private methods

    /// <summary>
    /// Handles the points added event by updating the displayed score
    /// </summary>
    /// <param name="points">points to add</param>
```

```

void HandlePointsAddedEvent(int points)
{
    score += points;
    scoreText.text = "Score: " + score;
}

#endregion
}

```

The script saves a copy of the `Text` object used to display the score in a field so it doesn't have to retrieve it every time points are added to the score. It also keeps track of the current score, making sure the score starts at 0.

In the `Start` method, the code tells the `EventManager` to add the `HandlePointsAddedEvent` method as a listener for the `PointsAddedEvent`. You should note that the `HandlePointsAddedEvent` method matches the requirements to be used as a `PointsAddedEventHandler` delegate as defined in the `PointsAddedEvent.cs` file we discussed above. The code also initializes the `text` value for the score text.

Finally, what does the `HandlePointsAddedEvent` method do when it "hears" the event? It simply adds the provided points to the current score, then sets the `text` value of the `Text` object to display the new score.

Be sure to attach the `HUD` script to the `HUD` game object or none of this will work! You also need to set the Score Text value in the `HUD` script by dragging the Score Text `Text` object from the Hierarchy window onto that value in the Inspector.

Finally, we finish our changes to the `Fish` script. Specifically, we remove the using directive for the `UnityEngine.UI` namespace (since we're no longer holding a `Text` object as a field), we remove the `score` and `scoreText` fields, we remove the initialization of the score text from the `Start` method, and in the `OnCollisionEnter2D` method we replace

```

// update score
score += bearPoints;
scoreText.text = "Score: " + score;

```

with

```

// update score
pointsAddedEvent.Invoke(bearPoints);

```

That last change makes the `Fish` script invoke the `PointsAddedEvent`; because the `HandlePointsAddedEvent` method in the `HUD` script was added as a listener for that event (through the `EventManager`), the score text is updated in the `HUD` when the event is invoked.

Run the game again and you'll see that everything works properly. Although the game works the same as it did in Chapter 14, it's a much better object-oriented design, and it also gave us a chance to learn a powerful new C# programming capability. Sweet.

## 17.4. Refactoring the Teddy Bear Destruction Game

Because delegates and event handling can be confusing for beginning programmers, we'll do one more example here. The complete game we develop in the final chapter will use multiple event handlers in our code.

For this example, we'll refactor the teddy bear destruction game from Chapter 16. In our previous solution, the `TeddyBearDestruction` script exposes an `AddPoints` method, which requires that all the teddy bear classes know about this method. We'll replace that structure with an event and an event handler instead.

As usual, we start by implementing the event, but our event is identical to the one we used in the previous section so there's no need to discuss that further here. The `EventManager` is more complicated for this example, though, so let's look at that now.

Before we look at the details of our implementation, let's think about how the events will work in this game. As we said in the previous section, a HUD is a pretty standard mechanism for displaying information to the player, so we'll use the `HUD` game object (and the `HUD` script) from the previous section as well. The additional complexity in our `EventManager` class comes from the invokers of the `PointsAddedEvent`.

In our previous example, the `Fish` script was the only script that invoked the event and the fish was already in the scene when the `HUD` script called the `EventManager` `AddListener` method to add a listener for the event. The problem is more complicated here because the invokers of the event (all three kinds of teddy bear) are actually spawned as the game progresses. We need the `HUD` script to listen for the events that are invoked by the newly-spawned teddy bears as well as the ones that are already in the scene (if there are any) when the `HUD` script calls the `EventManager` `AddListener` method. So there are actually two things the `EventManager` class needs to do: when the `HUD` script calls the `AddListener` method it needs to add the provided delegate from the `HUD` as a listener to each of the teddy bears that are already in the scene, and when a new teddy bear is spawned it needs to add the `HUD` delegate as a listener for that teddy bear as well.

Let's see how we can do both those things:

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

/// <summary>
/// Manages connections between event listeners and event invokers
/// </summary>
public static class EventManager
{
    #region Fields

    // save lists of invokers and listeners
    static List<TeddyBear> invokers = new List<TeddyBear>();
    static List<PointsAddedEventHandler> listeners =
        new List<PointsAddedEventHandler>();

    #endregion
}
```

The `EventManager` class in the previous section didn't have any fields, but to support the functionality we need this time we need to store lists of both the event invokers and the delegates (event handlers) that listen for the event.

```
#region Public methods

/// <summary>
/// Adds the given script as an invoker
/// </summary>
/// <param name="invoker">the invoker</param>
public static void AddInvoker(TeddyBear invoker)
{
    // add invoker to list and add all listeners to invoker
    invokers.Add(invoker);
    foreach (PointsAddedEventHandler listener in listeners)
    {
        invoker.AddListener(listener);
    }
}
```

When a new teddy bear is spawned, it will call the `EventManager` `AddInvoker` method shown above. That method adds the provided `TeddyBear` script to the list of invokers, then adds all the listeners in the `listeners` list as listeners for the `PointsAddedEvent` that the provided script could invoke. We haven't added an `AddListener` method to the `TeddyBear` script yet, so we'll get compilation errors at this point, but we'll fix that soon.

```
/// <summary>
/// Adds the given event handler as a listener
/// </summary>
/// <param name="handler">the event handler</param>
public static void AddListener(PointsAddedEventHandler handler)
{
    // add listener to list and to all invokers
    listeners.Add(handler);
    foreach (TeddyBear teddyBear in invokers)
    {
        teddyBear.AddListener(handler);
    }
}

#endregion
```

The `AddListener` method works much like it did in the previous section, but this time we don't need to tag any of the game objects that have scripts that will invoke the event because the `EventManager` doesn't have to find them all when a listener is added. Instead, each of those game objects will add themselves as an invoker when they're spawned. We also add the provided event handler to the `listeners` list so the `AddInvoker` method can add the event handler as a listener to a newly-spawned teddy bear.

Next, we add the required code to our `TeddyBear` script. This is very similar to the code we added to our `Fish` script in the previous section, with a new field:

```
// events fired by the class
protected PointsAddedEvent pointsAddedEvent = new PointsAddedEvent();
```

and a new method:

```
/// <summary>
/// Adds the given event handler as a listener
/// </summary>
/// <param name="handler">the event handler</param>
public void AddListener(PointsAddedEventHandler handler)
{
    pointsAddedEvent.AddListener(handler);
}
```

Notice that we marked the `pointsAddedEvent` field as `protected` so that child classes can access that field to call its `Invoke` method. Although we could add the field and the method to the `BurningTeddyBear`, `DisappearingTeddyBear`, and `ExplodingTeddyBear` classes, we know from our work with inheritance that there's a better way. Because the `TeddyBear` class is an ancestor (in this case, parent or grandparent) of all three of these classes, we add the field and method to the `TeddyBear` class instead. That way, all three of these classes will inherit them and we only need to include that code in one place.

We also remove the `teddyBearDestruction` field and the reference to that field in the `Start` method in the `TeddyBear` script because we won't need those any more now that we're using an event system.

The `Start` method in the `TeddyBear` script is the appropriate place for the script to call the `EventManager` `AddInvoker` method because the `TeddyBear` `Start` method executes whenever any of the 3 types of teddy bear is spawned in the game. All we need to do is add the following code:

```
// add as a PointsAddedEvent invoker
EventManager.AddInvoker(this);
```

Remember, the `this` keyword gives us a "self reference", so that's how we pass this particular `TeddyBear` script as an argument into the method.

We're getting close, but we still need to change the `BurningTeddyBear`, `DisappearingTeddyBear`, and `ExplodingTeddyBear` scripts to invoke the event instead of trying to call the `TeddyBearDestruction` `AddPoints` method in their implementations of the `ProcessMouseOver` method. Replace that method call with the following code in each of those scripts:

```
pointsAddedEvent.Invoke(pointValue);
```

Finally, our code compiles again!

It would also actually be reasonable to remove the line of code that invokes the event from the three scripts and move it into the `TeddyBear` `OnMouseOver` method instead; we could have also done this in the previous chapter instead of having each of the scripts call the `TeddyBearDestruction` `AddPoints` method. We chose to implement the code this way in case we decide later to add a child class that doesn't add points when the mouse is over the teddy bear, but either approach is fine.

We can now remove the entire `TeddyBearDestruction` script from our game. The only functionality that script contained was displaying the score, which the HUD now handles instead. Remove the Teddy Bear Destruction component from the Main Camera and delete the script from the scripts folder in the Project window.

Run the game to see that it works as it did before, though this time it uses what we've learned about events and event handlers.

## 17.5. UnityEvent and UnityAction

Up to this point, we've been defining our own events and the delegates that are called when those events are invoked. That's a general C# approach that doesn't depend on the Unity engine, so it's valid in any C# code you write. It turns out, though, that Unity has some built-in event support that we can use when we're developing Unity games.

The `UnityEngine.Events` namespace contains a set of classes called `UnityEvent` that we can use instead of defining our own events. The simplest version of `UnityEvent` in that namespace is for an event that assumes any listeners that have been added don't have any parameters; all those listeners are called when the event is invoked. You should know that the Unity documentation calls the listeners "callback" or "callback methods"; that's Unity-specific terminology that's interchangeable with the C# "delegate".

The other versions of `UnityEvent` in the `UnityEngine.Events` namespace are generics that assume the callback methods have 1, 2, 3, or 4 parameters. We've seen and used generics before; remember, when we use the `List` generic we specify the data type the list will hold between a `<` and a `>`. We'll see how to use one of these `UnityEvent` classes in the next section.

What happens if you need to call methods with more than 4 parameters when your event is invoked? The `UnityEngine.Events` namespace also has an abstract `UnityEventBase` class you can extend for however many parameters you need. All the versions of the `UnityEvent` class provided in the namespace use that approach.

The concepts we've discussed so far in this chapter are important for understanding how callbacks (delegates) and events work even though we haven't used `UnityEvent` yet. With that said, when we're writing Unity games we might as well use `UnityEvent`, so let's do that now.

## 17.6. Refactoring the Teddy Bear Destruction Game Again

In this section, we'll start with our Teddy Bear Destruction game from Section 17.4, converting it to use `UnityEvent` instead of our custom `PointsAddedEvent`.

Start by replacing all the code in the `PointsAddedEvent` script with the following:

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

```

/// <summary>
/// An event that indicates that points have been added
/// </summary>
public class PointsAddedEvent : UnityEvent<int>
{
}

```

This gives us a new class called `PointsAddedEvent` that's identical to the one parameter version of `UnityEvent` where the single parameter for the event handlers listening for the event has to be an `int`. We need to create a new class rather than using `UnityEngine<int>` directly because all the versions of `UnityEngine` that have 1 or more parameters are defined as abstract classes. What's an abstract class? Recall that it's a class that serves as a parent class for one or more child classes but we can't actually instantiate objects of the abstract class. The `PointsAddedEvent` class gives us a concrete version of that abstract class; we can't create instances of abstract classes, so we need a concrete class to call the constructor for the event in the `TeddyBear` script.

This change also gives us a number of compilation errors to guide us in our refactoring effort!

Let's start in the `TeddyBear` script. Start by adding a using directive for the `UnityEngine.Events` namespace; we'll need that soon. Our compilation error is in the header for the `AddListener` method because we don't have a `PointsAddedEventHandler` delegate any more.

What do we use for the delegate type for the parameter? The `UnityEngine.Events` namespace also provides 0 to 4 parameter versions of a `UnityAction` class that serves as the delegate for our event handlers. Changing our `AddListener` method header to

```
public void AddListener(UnityAction<int> handler)
```

gets rid of this compilation error for us.

You might be wondering why we didn't just include the following in our `PointsAddedEvent.cs` file for the delegate (similar to what we did for the original event):

```

/// <summary>
/// Delegate for handling the event
/// </summary>
public class PointsAddedEventHandler : UnityAction<int>
{
}

```

We can't do that because the `UnityAction` classes are marked as `sealed`, which means we can't define child classes from them.

Our remaining compilation errors are all in the `EventManager` class. To get rid of them, replace all occurrences of `PointsAddedEventHandler` with `UnityAction<int>`.

Now that everything compiles, go ahead and run the game. As you can see, everything works like it did before. The difference is that we're using the built-in Unity classes for our delegate and event rather than the more general C# delegate and event we defined in Section 17.4. Both approaches are valid, but when developing Unity games we'll use built-in Unity classes as much as possible.

## 17.7. Menu Buttons

Although we covered some aspects of a UI in Chapter 14 (specifically, text input and output), you might have wondered why we deferred menu system components until this chapter. We did that because the menu system components (like menu buttons) use one or more [UnityEvents](#) to implement the required behavior. We couldn't talk about how that works until we learned about [UnityEvent](#)!

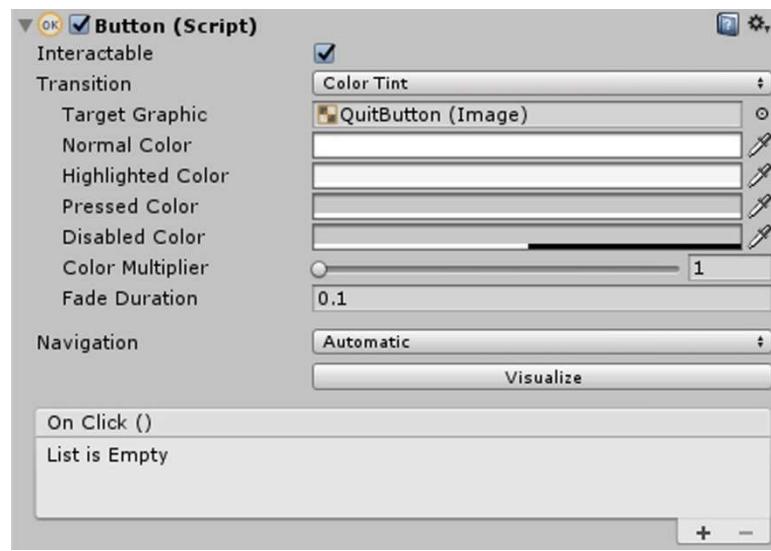
In this section, we'll look at a single menu button and how it works, then we'll add a very simple menu system to our fish game in the next section.

Start by creating a new 2D Unity project, creating a scenes folder, and saving the current scene as scene0 in that folder. Create a sprites folder and copy an image for a quit button into that folder.

Right click in the Hierarchy window and select UI > Image. Drag your sprite from the sprites folder in the Project window onto the Source Image value of the Image (script) component in the Inspector. You may need to change the X and Y location of the image so you can see it in the Game view. If your sprite doesn't look like the original in the Game view, click the Set Native Size button in the Image (script) component. Rename the Image in the Hierarchy window to QuitButton.

At this point, our button is really just like any other static sprite in the game, it doesn't do anything to let the player interact with it. To let the player click the button, we need to add an *interaction component* to our image. Unity provides a number of different interaction components, but you probably won't be surprised to learn that the one we want to add here is a Button component.

Click the Add Component button in the Inspector and select UI > Button. The Button component should look like the figure below in the Inspector.



**Figure 17.1. Button Component**

If you run the game and move the mouse over the button, you'll see that it darkens very slightly when the mouse is over the button. That's because the Normal Color value is pure white (RGBA 255/255/255/255) and the Highlighted Color is not quite white (RGBA 245/245/245/255). Feel free to adjust the Highlighted Color value to make it more obvious when the button is highlighted if you'd like.

You'll notice that clicking the button doesn't actually have any effect in our "game". That's because the Button component has an `OnClick UnityEvent` to define what to do when the button is clicked. See, here's why we needed to learn about `UnityEvent` first! If you look at the Button component in the Inspector, you'll see an `On Click ()` value at the bottom that says `List is Empty`. This is the list of listeners that have been added to listen for the event, but since we haven't added any listeners yet, nobody "hears" the event when it's invoked.

At this point, we need a method that we can add as a listener for when the quit button is clicked. Create a new scripts folder in the Project window, create a new C# script named `QuitButtonListener`, and change the script to:

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

/// <summary>
/// Listens for the quit button OnClick event
/// </summary>
public class QuitButtonListener : MonoBehaviour
{
    /// <summary>
    /// Handles the on click event from the quit button
    /// </summary>
    public void HandleOnClickEvent()
    {
        Application.Quit();
    }
}
```

We know we can use the `HandleOnClickEvent` method to listen for the `UnityEvent` invoked by the quit button because the method matches the callback signature for a no-parameter `UnityEvent`. When the `HandleOnClickEvent` method is called, the Unity game will close. The documentation for the `Application` `Quit` method tells us that "Quit is ignored in the editor", so we'll have to actually build our game to check it out. We'll do that once we're done with our work in the editor.

Attach the new script to the Main Camera in the Hierarchy window.

Now we can add our `HandleOnClickEvent` method as a listener for the `UnityEvent` invoked by the quit button. Select the Quit Button in the Hierarchy window. In the Inspector, click the `+` at the bottom right of the `On Click ()` value in the Button (Script) component. Click the small circle to the right of the `None (Object)` entry, select the Main Camera in the Select Object popup that appears, then close the popup. Click the `No Function` entry and select `QuitButtonListener > HandleOnClickEvent ()`. There, we've added our `HandleOnClickEvent` method as a listener for the `UnityEvent` invoked by the quit button.

As we said above, we need to actually build our game to make sure our quit button quits the game. Select `File > Build Settings ...` from the main menu bar. The `Scenes In Build` window at the top is empty, which means only the currently open scene will be included in the built game; although that's definitely what we want here, let's explicitly include our current scene in the list. Click the `Add Open Scenes` button below the bottom right of the `Scenes In Build` window to add `scene0` to the build. Click the `Build And Run` button at the bottom right of the popup. In the resulting file popup, create a new folder called `Build` and double-click the new folder (this is personal preference, you can put your built game

anywhere you want). Set the File name to MenuButtons and click the Save button. Wait patiently while your game builds.

Once the Unity Player window opens, click the Play! button near the bottom right. Notice that the quit button highlights when you mouse over it, and when you click it the game closes. Success!

In case you're wondering, you can also run your game by using your OS to navigate to your MenuButtons.exe file and double-clicking that file. If you want to distribute your game to someone else, you need to give them both the MenuButtons.exe file and the MenuButtons\_Data folder. They'll need both of those to run your game.

## 17.8. Adding a Menu System to the Fish Game

Now that we know how menu buttons work, let's add a menu system to our fish game from Section 17.3. This will be a pretty simple menu system – we'll only have play and quit buttons on a main menu – but it will give us a chance to use what we've learned in this chapter, particularly the previous couple of sections.

We'll also finally have a game with multiple scenes! Copy and paste the Unity project from Section 17.3 to create a new project. Open the project and rename scene0 in the scenes folder to game instead. Next, we'll add a second scene for our main menu.

Right click the scenes folder in the Project window and select Create > Scene. Rename the new scene to mainmenu. Double click the mainmenu scene in the Project window to open that scene. Add sprites for a play button and a quit button to the sprites folder in the Project window. Create Images for a Play Button and a Quit Button and add Button components to both of them like we did in the previous section. Place the buttons in the scene in a reasonable way by changing the X and Y locations of the images.

Now we'll implement the script to handle the On Click event for the buttons on the main menu. Create a new script called `MainMenu` and double click the new script to open it in Visual Studio. Change the script to:

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

/// <summary>
/// Listens for the OnClick events for the main menu buttons
/// </summary>
public class MainMenu : MonoBehaviour
{
    /// <summary>
    /// Handles the on click event from the play button
    /// </summary>
    public void HandlePlayButtonOnClickEvent()
    {
        SceneManager.LoadScene("game");
    }
}
```

```
/// <summary>
/// Handles the on click event from the quit button
/// </summary>
public void HandleQuitButtonOnClickEvent()
{
    Application.Quit();
}
```

Attach the `MainMenu` script to the Main Camera.

The `HandlePlayButtonOnClickEvent` method loads the game scene, moving us to our gameplay. The `SceneManager` class is in the `UnityEngine.SceneManagement` namespace, so we need to add a `using` directive for the namespace. The `HandleQuitButtonOnClickEvent` is identical to the `HandleOnClickEvent` method from the previous section (with a slightly different name, of course).

Select the Play Button in the Hierarchy window and add the `HandlePlayButtonOnClickEvent` method as a listener for its `On Click ()` event. Select the Quit Button in the Hierarchy window and add the `HandleQuitButtonOnClickEvent` method as a listener for its `On Click ()` event.

Now we need to set our build to include both scenes. Select `File > Build Settings ...` from the main menu bar. Drag the `mainmenu` and game scenes from the `scenes` folder in the Project window onto the `Scenes In Build` window to add those scenes to the build. The order matters, because the scene at the top of the list will be the scene the game starts in when you run it in the Unity Player. Click the `Build And Run` button at the bottom right of the popup. In the resulting file popup, create a new folder called `Build` and double-click the new folder. Set the File name to `FishGame` and click the `Save` button.

Once the Unity Player window opens, click the `Play!` button near the bottom right. Both the play button and the quit button should work as expected. Awesome.

# Chapter 18. Exceptions, Equality, and Hash Codes

At this point, we've covered almost all the basic C# and Unity ideas needed for an introduction to those topics. In this chapter, we'll cover exceptions as well as equality and hash codes for reference types.

## 18.1. Exceptions

Most of you have probably already discovered that there are times when a program you wrote will “blow up” – if you try to call a method on an object you haven’t yet created, for example. The program doesn’t really blow up, of course; instead, C# throws an *exception*, which then terminates the program. To keep our program from terminating because of the exception, we can handle (or catch) that exception by using an *exception handler*. We can also explicitly throw exceptions in our code as necessary. We’ll cover all those ideas in this section.

Although our Unity games don't typically terminate when exceptions are thrown, we can see those exceptions getting thrown in the Console window as our game runs. Because this is an indication of a problem in our game, those exceptions need to be fixed or handled as well.

The key idea behind all this is that our programs throw exceptions when something unusual happens that has significant negative effects on the program’s ability to continue executing correctly. We’ll see specific examples below, but remember that exceptions aren’t about “business as usual,” they’re for taking care of unusual circumstances.

Let’s look at a simple example. Say we have the following code to divide two numbers provided by the user:

```
// get numerator and denominator
Console.WriteLine("Enter numerator: ");
double numerator = double.Parse(Console.ReadLine());
Console.WriteLine("Enter denominator: ");
double denominator = double.Parse(Console.ReadLine());

// calculate and display result
double result = numerator / denominator;
Console.WriteLine("Result of division is: " + result);
```

As long as the user enters numbers at the above prompts, the code works fine (although we get strange results if the denominator is 0 or both the numerator and denominator are 0).

What happens if the user doesn’t enter a number, though? Let’s say we run the code above and enter the string Bob for the numerator. The code (the `Parse` method) will throw an exception – specifically, a `FormatException` – and terminate the program. How do we make sure the user can’t blow up our program by doing this? By including an exception handler. Exception handlers include a *try block* and one or more *catch blocks*, though we’ll also use a *finally block* when we start doing file IO in the next chapter.

Because the user can crash our program by causing the `FormatException` to be thrown, we should probably do something about that!

Here's the general syntax for exception handlers:

---

### SYNTAX: Exception Handlers

```
try
{
    executable statements
}
catch (ExceptionClass exceptionObject)
{
    executable statements
}
catch (ExceptionClass exceptionObject)
{
    executable statements
}
. . .
finally
{
    executable statements
}
```

*ExceptionClass*, the class of the exception we're catching

*exceptionObject*, the object of the exception class that's created when the exception is thrown

*executable statements*, one or more executable statements

---

We first put the executable code that could throw the exception inside a try block; in other words, between the { and the } after the `try`. We then include a catch block for each exception we need to handle. We include the class of the exception we're catching (`FormatException` is one such class) and the exception object that's created when the exception is thrown (you can name that object anything you want). Then, between the curly braces of the catch block, you include whatever code you want executed if that exception is thrown. For our example, we end up with

```
try
{
    // get numerator and denominator
    Console.Write("Enter numerator: ");
    double numerator = double.Parse(Console.ReadLine());
    Console.Write("Enter denominator: ");
    double denominator = double.Parse(Console.ReadLine());

    // calculate and display result
    double result = numerator / denominator;
    Console.WriteLine("Result of division is: " + result);
}
catch (FormatException fe)
{
    Console.WriteLine("Invalid input!");
}
```

If either of the calls to the `Parse` method throws a `FormatException`, the program immediately goes to the catch block to execute the code in that block. If all of the code in the try block executes without throwing an exception, the program simply skips all the catch blocks after the try block code is done and proceeds to the code following the exception handler.

That solves our immediate problem (catching the thrown exception), but if this exception does get thrown, we probably want to give the user an opportunity to retry their input. If we realize that the `FormatException` is only thrown if there's some error in the user input, our solution becomes clearer – we should let the user keep trying until they provide valid input. We'll do that soon.

If you have code within a try block that could raise multiple exceptions, you would still use a single try block; you simply include multiple catch blocks, one for each exception you want to catch. You should know, however, that only one catch block will be executed for any given exception; we'll look at the implications of that rule more closely a little later.

In some cases, we also include a finally block in our exception handler. The code in the finally block will get executed whether or not the code in the try block throws an exception. If the code in the try block doesn't throw an exception, the program proceeds to the finally block after the try block code is done. If the code in the try block throws an exception and none of the catch blocks handle that exception, the program immediately proceeds to the finally block code. If the code in the try block throws an exception and one of the catch blocks handles that exception, the code in that catch block is executed, then the program proceeds to the finally block after the catch block code is done.

When is a finally block useful? When we have some code that we know we want to execute whether or not an exception is thrown in the try block. We'll see a great example of this in the next chapter when we're doing file IO. We'll discover that we should always close files that we're reading from or writing to, whether or not we experience a read or write exception. We'll therefore make sure we close files in the finally block of our exception handlers.

### *Exception Handling for Input Validation*

Recall that one of the problems we solved with a while loop was validating that a user entered a valid (in range) GPA value using the following code:

```
// prompt for and get GPA
Console.WriteLine("Enter a GPA (0.0-4.0): ");
double gpa = double.Parse(Console.ReadLine());

// loop for a valid GPA
while (gpa < 0.0 || gpa > 4.0)
{
    // print error message and get new GPA
    Console.WriteLine("Invalid entry! GPA must be between 0.0 and 4.0.");
    Console.WriteLine();
    Console.WriteLine("Enter a GPA (0.0-4.0): ");
    gpa = double.Parse(Console.ReadLine());
}
```

We already know that the user could crash the above code by entering a string rather than a number, and we also know we can solve that problem using an exception handler. This is a somewhat harder problem to solve than the previous one, though.

One of the things we need to decide is what to include in the try block. We know we need to include both calls to the `Parse` method, and though we could do this with two separate exception handlers, it makes more sense to include the entire chunk of code above in the try block. Our first cut at our solution could therefore be something like:

```
try
{
    // prompt for and get GPA
    Console.WriteLine("Enter a GPA (0.0-4.0): ");
    double gpa = double.Parse(Console.ReadLine());

    // loop for a valid GPA
    while (gpa < 0.0 || gpa > 4.0)
    {
        // print error message and get new GPA
        Console.WriteLine("Invalid entry! GPA must be between " +
            "0.0 and 4.0.");
        Console.WriteLine();
        Console.WriteLine("Enter a GPA (0.0-4.0): ");
        gpa = double.Parse(Console.ReadLine());
    }
}
catch (FormatException fe)
{
    Console.WriteLine("Invalid entry! GPA must be a number.");
    Console.WriteLine();
}
```

This solves our immediate problem – the user can't crash the code by entering a string – but if they enter a string the catch block executes and they don't get a chance to try another input. We need the exception handler to be contained inside a loop to make this happen.

We definitely need a while loop, but what should we use for our loop condition? Conceptually, we need to loop while the user hasn't provided a valid input, so let's use a Boolean flag to tell whether the user has provided valid input:

```
bool valid = false;
while (!valid)
{
    try
    {
        // prompt for and get GPA
        Console.WriteLine("Enter a GPA (0.0-4.0): ");
        double gpa = double.Parse(Console.ReadLine());

        // loop for a valid GPA
        while (gpa < 0.0 || gpa > 4.0)
        {
            // print error message and get new GPA
            Console.WriteLine("Invalid entry! GPA must be between " +
                "0.0 and 4.0.");
            Console.WriteLine();
            Console.WriteLine("Enter a GPA (0.0-4.0): ");
            gpa = double.Parse(Console.ReadLine());
        }
    }
}
```

```
        }
    valid = true;
}
catch (FormatException fe)
{
    Console.WriteLine("Invalid entry! GPA must be a number.");
    Console.WriteLine();
}
}
```

We start by setting the flag to `false` to force us into the loop body the first time. The only way we get to the line of code that changes the flag to `true` (just above the catch block) is when the user enters a number that's between 0.0 and 4.0.

There are a couple of things we can clean up in the above code. For example, the code that prompts for and reads in the GPA is included twice. We originally did that to give us a self-contained while loop to do the range checking, but now that all the code is contained in another while loop we no longer need to have that duplication.

It's also a little confusing understanding when the flag gets set to `true`. We can certainly figure that out, but we can make that clearer as well. The following code provides our final solution:

```
bool valid = false;
while (!valid)
{
    try
    {
        // prompt for and get GPA
        Console.WriteLine("Enter a GPA (0.0-4.0): ");
        double gpa = double.Parse(Console.ReadLine());

        // check for valid range
        if (gpa < 0.0 || gpa > 4.0)
        {
            Console.WriteLine("Invalid entry! GPA must be between " +
                "0.0 and 4.0.");
            Console.WriteLine();
        }
        else
        {
            valid = true;
        }
    }
    catch (FormatException fe)
    {
        Console.WriteLine("Invalid entry! GPA must be a number.");
        Console.WriteLine();
    }
}
```

That's one way we can use exception handlers to provide very robust input validation.

### Multiple Catch Blocks

We said above that we can have multiple catch blocks for our exceptions, so let's look at an example of that here. It turns out that the `double Parse` method can actually throw three different exceptions: `ArgumentNullException` (if the string we're parsing is `null`), `FormatException` (which we've already discussed), and `OverflowException` (if the number is too small or too large to fit into a `double`).

In our example here, it's impossible for a null string to be parsed, because if the user just presses the Enter key we read in an empty string (a string with no characters in it) rather than a null string. Let's say we want to print a different message for each of the other two exceptions above; here's our new code:

```
bool valid = false;
while (!valid)
{
    try
    {
        // prompt for and get GPA
        Console.WriteLine("Enter a GPA (0.0-4.0): ");
        double gpa = double.Parse(Console.ReadLine());

        // check for valid range
        if (gpa < 0.0 || gpa > 4.0)
        {
            Console.WriteLine("Invalid entry! GPA must be between " +
                "0.0 and 4.0.");
            Console.WriteLine();
        }
        else
        {
            valid = true;
        }
    }
    catch (FormatException fe)
    {
        Console.WriteLine("Invalid entry! GPA must be a number.");
        Console.WriteLine();
    }
    catch (OverflowException oe)
    {
        Console.WriteLine("Invalid entry! GPA must be between " +
            "0.0 and 4.0.");
        Console.WriteLine();
    }
}
```

Catch blocks work much like the if and else if blocks in an if statement, where the block for the first Boolean expression that evaluates to `true` is executed then the code leaves the if statement. For catch blocks, the first block that matches the exception that was thrown is the block that's executed, then the code proceeds to the finally block (if there is one) or out of the exception handler.

You probably noticed that the error message in the `OverflowException` catch block is the same as the error message if the user enters a valid number but it's out of range. We know that if the user enters a number that's too small or too large to fit in a `double` that the number is also not between 0.0 and 4.0.

That means we can just give the user an error message that makes sense to them without requiring that they understand the valid ranges for C# data types.

### Throwing Exceptions

Up to this point, we've talked about how we can go about handling exceptions thrown by our programs. We now discuss how we can explicitly throw exceptions ourselves (which we can later catch elsewhere in our code if we choose to).

Let's say we wanted to have the user enter a date in the format mm/dd/yyyy. There's actually a lot of work we'd need to go through to check that format, but for this example, we'll just make sure the string the user enters contains two / characters. While there are really clean ways to do this (using something called *regular expressions*, which are beyond the scope of this book), we'll use the approach shown below:

```
// prompt for and read in date
Console.WriteLine("Enter date as mm/dd/yyyy: ");
string date = Console.ReadLine();

// count slashes
int slashCount = 0;
string tempString = date;
while (tempString.IndexOf('/') != -1)
{
    slashCount++;
    tempString = tempString.Substring(tempString.IndexOf('/') + 1);
}
```

When this code finishes executing, the `slashCount` variable holds the number of slashes contained in the input string. If that number is 2, the string contains the correct number of slashes, otherwise the input format is incorrect.

Here's where we get to learn how to throw an exception. Check out the following code:

```
// throw exception for wrong number of slashes
if (slashCount != 2)
{
    throw new ApplicationException("Invalid date format entered");
}
```

The condition simply checks the value of `slashCount`; the body of the if statement is the new stuff. To throw an exception, we start with the `throw` keyword. We then need to create a new exception object using a constructor for the exception class we want to use. The `ApplicationException` class is designed to be used “when a non-fatal application error occurs”, so that's the exception we throw here.

The `ApplicationException` constructor is overloaded, and we're using the overload that lets us provide an error message as an argument. If we were to run the code above and force the exception to be thrown (we won't provide any slashes in our input), we'd get the output shown in Figure 18.1. Notice that the error message provided with the exception is the error message we passed as an argument to the constructor.



**Figure 18.1. Thrown Exception Output**

Of course, an even better solution would be to enclose the above code in an input validation loop like we did above for GPA so the user can try again if they get the date format incorrect. We kept our example here simple to focus on throwing the exception; see if you can figure out how to add the input validation loop as well.

### *User-Defined Exceptions*

C# provides a class hierarchy with a number of useful pre-defined exceptions in the hierarchy. The `Exception` class is the base (root) class for all exceptions in that hierarchy.

If none of the pre-defined exceptions are exactly what you want, you can define a new exception class yourself. The only constraint is that you should make the `Exception` class the parent class for your new class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CustomException
{
    /// <summary>
    /// A custom exception class
    /// </summary>
    public class MyException : Exception
    {
    }
}
```

Because your new exception class inherits the fields and methods from the `Exception` class, you can throw and handle your new exception just as we've been doing all along.

## 18.2. Equality

Up to this point in the book, when we wanted to compare two variables we used `==` for value types or for `strings` (which are, as we know, a reference type). We haven't, however, compared any of the objects for the classes we've written to see if they're equal. So what is equality anyway? We know what it means for two integers to be equal, but what does it mean for two objects of a particular class to be equal?

For reference types, the default `Equals` method (we can also use `==` for the same behavior) that's inherited by any class we define simply checks for *reference equality*. If two variables refer to the same object in memory, the `Equals` method returns `true`, otherwise it returns `false`. In many cases what we really want is *value equality*, not reference equality. For value equality, what we really care about is the fact that the important characteristics of two objects are the same even if they're two distinct objects in memory.

Let's make this more concrete. Let's assume we've developed a `Mackerel` class that has fields and properties for length, weight, and the damage the mackerel inflicts when you whack something with it (we know, we know). What would we mean if we said that two `Mackerel` objects were equal? We'd probably mean that their length, weight, and damage are the same. We already learned about overriding methods when we talked about inheritance, and that's exactly what we need to do here. Specifically, here's how we would override the `Equals` method:

```
/// <summary>
/// Standard equals comparison
/// </summary>
/// <param name="obj">the object to compare to</param>
/// <returns>true if the objects are equal, false otherwise</returns>
public override bool Equals(object obj)
{
    // comparison to a null object returns false
    if (obj == null)
    {
        return false;
    }

    // if the parameter can't be cast to Mackerel return false
    Mackerel objMackerel = obj as Mackerel;
    if (objMackerel == null)
    {
        return false;
    }

    // compare properties
    return (Length == objMackerel.Length) &&
           (Weight == objMackerel.Weight) &&
           (Damage == objMackerel.Damage);
}
```

The last statement in the method does the actual comparison we're trying to capture. The two `if` statements are also necessary, however. The standard return value for calling `Equals` with a `null` parameter is `false`, because the object we're using to call the `Equals` method can't possibly be `null`; if it were, we'd get a `NullReferenceException` for trying to call a method on a `null` object. The second

if statement makes sure we're actually comparing to a `Mackerel`; if we aren't, the objects can't be equal so we return `false`.

There is another important concept in the above code as well. We've included the line of code that says

```
Mackerel objMackerel = obj as Mackerel;
```

This line tries to cast `obj` as a `Mackerel` object. If `obj` isn't actually a `Mackerel` object, that type cast returns `null`; that's why we check for `null` in the next line to determine whether we're comparing to a `Mackerel` object. If `obj` is a `Mackerel` object, the `objMackerel` variable gets a reference to `obj` as a `Mackerel` object.

Why do we have to go through all this? Because the standard `Equals` method requires that we pass in an `object` argument, not a `Mackerel` argument. That means that somewhere in the `Equals` method we need to start treating that parameter as a `Mackerel` so we can access its properties to do the comparison. The type casting code above is what does that for us.

Microsoft's documentation recommends that if we override the `Equals` method that we also provide a type-specific `Equals` method that (in this case) has a `Mackerel` parameter rather than an `object` parameter; this apparently enhances performance. That method is provided below.

```
/// <summary>
/// Type-specific equals comparison
/// </summary>
/// <param name="mackerel">the mackerel to compare to</param>
/// <returns>true if the mackerels are equal, false otherwise</returns>
public bool Equals(Mackerel mackerel)
{
    // comparison to a null mackerel returns false
    if (mackerel == null)
    {
        return false;
    }

    // compare properties
    return (Length == mackerel.Length) &&
        (Weight == mackerel.Weight) &&
        (Damage == mackerel.Damage);
}
```

Notice that in this case we don't include `override` in the method signature. That's because this is an overload, not an override. Remember, overloaded methods have the same method name but different parameters.

At this point, we could declare victory and move on, but you should have noticed that our two `Equals` methods have a lot of duplicated code. Now that we've written our type-specific `Equals` method we can just use that method to make a huge simplification to our first `Equals` method. Here's the new code:

```
/// <summary>
/// Standard equals comparison
/// </summary>
/// <param name="obj">the object to compare to</param>
```

```
/// <returns>true if the objects are equal, false otherwise</returns>
public override bool Equals(object obj)
{
    return Equals(obj as Mackerel);
}
```

If `obj` is `null` or isn't actually a `Mackerel` object, the `obj as Mackerel` type cast returns `null` (which we then pass as an argument to the type-specific `Equals` method), so the if statement at the beginning of the type-specific `Equals` method handles the two special cases for the equality check. This refactoring removes our duplicated code so that each chunk of code only appears in one place in our game, something we keep saying is an important characteristic of good software.

We can now use our `Equals` methods to compare any object to a `Mackerel` object.

### 18.3. Hash Codes

It turns out that if we only provide the `Equals` methods for our `Mackerel` class, the compiler gives us the following warning:

```
Mackerel overrides Object.Equals(object o) but does not override
Object.GetHashCode()
```

What's that all about? It turns out that there's a very useful thing called a *hash code*. A hash code assigns a number to an entity (an object, in our case). In a perfect world, every unique object gets a unique hash code, but that doesn't always happen in practice. Why do we care? Because hash codes can be used to efficiently store objects in a variety of ways. If you continue your studies in programming, you'll definitely be learning about a variety of data structures, including hash tables. Not surprisingly, hash tables use the hash codes we're examining here.

So why does the compiler give us the above warning? Because if the `Equals` method returns `true`, the `GetHashCode` method (which is inherited from the `Object` class) better return the same number for both objects! The default implementation of `GetHashCode` can't provide this guarantee when we override the `Equals` method, so we need to override the `GetHashCode` method as well.

To write our `GetHashCode` method, we basically need to use the data inside the object to calculate the hash code for the object. For our `Mackerel` class, we have `length`, `damage`, and `weight` instance variables we can use. One reasonable implementation of the method is as follows:

```
/// <summary>
/// Calculates a hash code for the mackerel
/// </summary>
/// <returns>the hash code</returns>
public override int GetHashCode()
{
    const int HashMultiplier = 31;
    int hash = length;
    hash = hash * HashMultiplier + weight;
    hash = hash * HashMultiplier + damage;
    return hash;
}
```

This is a pretty good hash function that will be fast and has a good chance of generating unique hash values for `Mackerel` objects that have different instance variable values. Using a prime number for the `HashMultiplier` helps ensure (but doesn't guarantee) unique hash values.

One final comment. There are many, many possible hash functions, and coming up with good hash functions that are quick and have a good chance of generating unique hash values for each object has been an area of research for some time.

## Chapter 19. File IO

As part of our motivation for using arrays, we talked about storing 12,000 GPAs for the students at our university. We showed how we can store those GPAs in an array, and we showed how to read into elements of an array, but we glossed over a couple of important considerations. First of all, who in their right mind is going to sit at the keyboard and type in 12,000 GPAs? Even more importantly, what happens to those GPAs after the program terminates? They disappear! So every time we want to do something with those GPAs, we have to enter all 12,000 again! There just has to be a way to store these GPAs for later use, and there is – *files*.

There are lots of types of files; one common type of file is called a *text file*. A text file simply consists of a set of characters. Essentially, any character we can enter from the keyboard can be read from a file instead, and any character we could print to the screen can also be output to a file. Thus, files give us a great way to store large amounts of data for input to computer programs, output from computer programs, or both.

We can also save our objects into *binary files*. This is a very useful capability if we want those objects to *persist* across program executions. Just like variables that are value types, objects disappear when the program terminates. If we write them to a file, though, we can retrieve them later.

Files are incredibly useful in the game domain, of course. We regularly use files to store configuration information for the games we write: how many points we want per game, how the game tutorial progresses from one activity to the next, the stats for each NPC, and so on. Those configuration files are usually read-only, since they represent core information about how the game works. We want to have files that we can also write to, though: saved game files, saved profile information for specific players, etc. You should be able to see why being able to do file IO will help us build better games.

In this chapter, we'll start by looking at how we do file IO in any C# application. We'll then move on to the best ways to interact with files in Unity.

### 19.1. Streams

To understand how file input and output work in C#, we need to understand *streams*. In general, streams are what you actually do input from and output to in C#. You've actually already been using streams in your console applications because the `Console` class actually gave us access to an input stream (when we used the `Read` or `ReadLine` methods), an output stream (when we used the `Write` or `WriteLine` methods), and an error stream (which we didn't use at all). In other words, we've already been using streams, we just didn't realize it!

Think of a stream as a sequence of data that flows into your program from some input device (the keyboard or an input file, say) or out of your program to some output device (e.g., the screen or an output file). In other words, you have a stream of data that you read from or write to, with some input or output device on the other end of the stream.

Our goal for file IO is to actually use streams to and from files rather than using the standard input stream (from the keyboard) and the standard output stream (to the console window). To do that, we'll use the `File` and `FileStream` classes from the `System.IO` namespace. The `FileStream` object is the

stream we actually interact with, while the `File` class gives us useful utility methods for creating, opening and closing file streams.

## 19.2. Text Files

Text files are really just files of Unicode characters. Text files can be created using any number of programs, including Microsoft Wordpad, Notepad++, and programs that you write. When we interact with text files, we can use the `StreamReader` and `StreamWriter` classes to interact with the underlying file stream. This is particularly helpful since those classes provide handy `ReadLine` and `WriteLine` methods. We certainly could interact with a `FileStream` object directly instead, but then we have to handle dealing with each byte that we write ourselves; let's do it the easier way!

Let's start by creating a file that we can send text output to. One way we can easily create a `StreamWriter` object is to use the `File` class as shown below.

---

### SYNTAX: Creating StreamWriter Objects

```
StreamWriter streamWriterObjectName = File.CreateText("diskFileName");
```

`streamWriterObjectName`, the name of the `StreamWriter` object  
`diskFileName`, the actual name of the file on the disk (can include path)

---

Just as you need to know the name of the book you're going to write, you need to know the name of the file you want to write to disk before you can create the file object. C# assumes that the file you're trying to create should be in the same folder as the program you're running, but you can also use the fully-qualified path name for the output file (e.g., `@"C:\chamillard\book\Chapter19\speech.txt"` or `"C:\\chamillard\\book\\Chapter19\\speech.txt"`) if you'd like.

For our example, we can use the following:

```
// create stream writer object
StreamWriter output = File.CreateText("speech.txt");
```

It turns out, however, that if you read the documentation for the `File CreateText` method, you'll discover that the method can throw one of six different exceptions! What should we do if that happens? One alternative would be to just let the program crash, but that's almost never a good idea. Instead, we'll put the call to the `CreateText` method inside a `try` block and include a `catch` block for any exceptions. We'll definitely do that, but we'll defer discussing the details until we're all done with our file output.

Now that we have a `StreamWriter` object, we can write to the file using the `WriteLine` method. Let's write a few lines from a speech:

```
// write a few speech lines
output.WriteLine("Four score and seven years ago");
output.WriteLine("our fathers brought forth on this continent");
output.WriteLine("a new nation, conceived in liberty");
```

```
output.WriteLine("and dedicated to the proposition that all " +
    "men are created equal");
```

As you can see, sending output to the file is just as easy as sending output to the console window using the `Console` class.

Once we're done sending output to the file, the last thing we need to do is close the file. Because we do that in the finally block of our exception handling code, now is a good time to look at the complete chunk of code:

```
StreamWriter output = null;
try
{
    // create stream writer object
    output = File.CreateText("speech.txt");

    // write a few speech lines
    output.WriteLine("Four score and seven years ago");
    output.WriteLine("our fathers brought forth on this continent");
    output.WriteLine("a new nation, conceived in liberty");
    output.WriteLine("and dedicated to the proposition that all " +
        "men are created equal");
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    // always close output file
    if (output != null)
    {
        output.Close();
    }
}
```

Notice that we always close the stream writer (which also closes the underlying output file the stream writer is using) in the finally block. Why do we do it there? Because we know that the code in the finally block always executes whether or not any exceptions were thrown in the try block and handled (or not) in a catch block. By doing it this way, we make sure that the file associated with the stream writer is always closed when we leave this chunk of code.

You should note that we needed to move the declaration of our `StreamWriter` variable outside the try block to make it visible in the finally block, and we needed to initialize it to `null` so the compiler didn't think we were trying to use an uninitialized variable.

So how can we tell that the code above actually writes to the file properly? Perhaps the easiest way would be to just open the file using Notepad++ and making sure it contained the speech lines we wrote. Another way, though, would be to open the file in our code, read it back in, and confirm that it's correct by echoing each line to the console window. Because you need to know how to do file input as well, we'll take that approach.

Before we do that, though, you should know that the `File.CreateText` method creates a new file; if a file with the given name already exists, the contents of that file are overwritten. That may be exactly what we want, but we might want to do something different instead. For example, we may want to append text to an existing file. If you need to do something like that, it's better to open the file using the `File.Open` method instead. The `Open` method has two parameters: a path to the file (which can just be the file name) and a  `FileMode` value.  `FileMode` is an enumeration that has six different values you can use to specify how the operating system should open the file.

Because the `Open` method returns a `FileStream` rather than a `StreamWriter`, you'd need to use the following code to replace what we have above:

```
output = new StreamWriter(File.Open("speech.txt", FileMode.Create));
```

Using the `Open` method is obviously a little more complicated than using the `CreateText` method, but if you need finer-grained control of how the file is opened it's definitely the way to go.

Now let's move on to reading from a text file. The ideas are very similar to those we used when we wrote to a file; we're just using a file stream for input rather than output. Our first step is to open the text file for reading as shown below.

---

#### SYNTAX: Creating StreamReader Objects

```
StreamReader streamReaderObjectName = File.OpenText("diskFileName");
```

`streamReaderObjectName`, the name of the  `StreamReader` object  
`diskFileName`, the actual name of the file on the disk (can include path)

---

For the code we're currently writing, we use:

```
// create stream reader object
StreamReader input = File.OpenText("speech.txt");
```

The `OpenText` method can also throw a number of exceptions, so we'll put the call to the `OpenText` method inside a `try` block and include a `catch` block for any exceptions.

Now that we have a  `StreamReader` object, we can read from the file using the `ReadLine` method. Reading from a file is a little trickier than writing to a file, though, because we need a way to know when we reach the end of the file. The good news is that the `ReadLine` method returns `null` when it reaches the end of the file stream. That means we can use a `while` loop to read each line from the file until `ReadLine` returns `null`. Here's the complete block of file input code:

```
// read from the text file
StreamReader input = null;
try
{
    // create stream reader object
    input = File.OpenText("speech.txt");
```

```

// read and echo lines until end of file
string line = input.ReadLine();
while (line != null)
{
    Console.WriteLine(line);
    line = input.ReadLine();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    // always close input file
    if (input != null)
    {
        input.Close();
    }
}

```

We of course want to close the file once we're done reading from it; we do that in the finally block just as we did for file output to make sure the file gets closed whether or not an exception is thrown during our file input processing.

It's important to note that to properly read from an input file you need to know the format and the contents of the file. For example, consider a file that consists of a line containing a person's name followed by a line containing their birthday (and so on for multiple people). Any program that reads from that file needs to read in the name followed by the birthday; trying to read the birthday first, followed by the name, would lead to some potentially interesting but undoubtedly incorrect behavior! You need to make sure you know how the information is arranged in the input file to read from it properly.

Now, you may be thinking that by reading an entire line at a time from the file that we've made it difficult to have multiple values on a line. For example, what if we wanted to store X and Y coordinates for a set of points in the file? It would certainly be more intuitive to have both coordinates for each point on a single line; that way, each line in the file would represent information about a single point.

We can actually do this fairly easily using the `String Split` method. Before we can do that, though, we need to know the format of each line in the file. Let's assume that each line is formatted with the x and y values separated by a comma. Files that are saved as CSV (comma-separated value) files have precisely this format, so this is a pretty common and intuitive file format. Let's look at a snippet of code to get us started, assuming `input` is a `StreamReader` that's already been opened:

```

string line = input.ReadLine();
string[] tokens = line.Split(',');

```

The second line of code is the new stuff, of course. The `Split` method breaks the `line` string into an array of strings; each element in the array is one of the strings that's separated by commas in the `line` string. For example, if `line` has the value `100,200` then `tokens[0]` is `100` and `tokens[1]` is `200`.

Although we now have the x and y coordinates extracted from the line we read from the file, we probably want them as integers instead of the strings that are stored in the tokens array. That conversion is easy to do using the `int.Parse` method we learned about in Chapter 5:

```
int x = int.Parse(tokens[0]);
int y = int.Parse(tokens[1]);
```

That's all you need to know to write to or read from text files. Let's look at binary files next.

### 19.3. Files of Objects

Text files are very useful for storing character and value type information, but except for some of the first programs we wrote, all our programs have contained one or more objects. It seems like we should also be able to read objects from and write objects to files so that their information (the state of the object) can also persist across program executions. There is a way, of course, and that's what we'll discuss in this section.

There's another reason we don't want to store game information in a straight text file. Imagine that you use such a file to save player profile information; it would be very easy for the player to open up their profile and modify their information to get an advantage in the game. Similarly, players could open up a text file containing game configuration information to reduce the NPC stats to make the game easier. Although doing either of those things would almost definitely adversely affect the player's experience with the game, there are unfortunately many players who will hack a game to gain an advantage. Storing this information more securely than in straight text is therefore generally a good idea for the games we build.

When we store and retrieve objects from files, we're actually interacting with binary files instead of text files. Many of the ideas are similar, but we'll use some new classes and features to do the binary file IO.

The first new concept we need concerns our ability to *serialize* an object. Because binary files will store a sequence of bytes to represent each object that's saved in the file, we need a way to convert the objects we want to save into a stream of bytes. The good news is that we don't have to worry about the details of how the conversion to bytes actually works, all we need to do is indicate that we want objects of a particular class to be serializable. Let's actually work through a complete example to show how all this works.

Let's create a `Deck` of `Card` objects like we did in Chapter 12. We'll actually save the `Deck` object to a binary file, then read it back in and print its contents to the console window to show that everything worked properly.

The first thing we need to do is indicate that `Deck` objects are serializable. This is actually really easy – check it out:

```
[Serializable]
public class Deck
```

We're using something called an attribute to mark the `Deck` class as `Serializable` (`SerializeField`, which we've been using since Chapter 4, is also an attribute). Because we know that a `Deck` holds `Card` objects, we also need to mark the `Card` class as `Serializable`:

```
[Serializable]
public class Card
```

You might be wondering why we didn't need to also mark the `Rank` and `Suit` enumerations as `Serializable`. That's because C# enumerations have integers as their underlying data type, and integers are automatically serializable in C#.

Let's look at the code required to create a `Deck` object and write it to a file named `deck.dat`:

```
// create deck and write to file
Deck deck = new Deck();
FileStream output = null;
try
{
    output = File.Create("deck.dat");
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(output, deck);
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    // always close output file
    if (output != null)
    {
        output.Close();
    }
}
```

This time, we're using a `FileStream` object so we can write the serialized `Deck` object to the file stream. We're using the `File Create` method instead of the `CreateText` method because we need a `FileStream` object, not a `StreamWriter` object. The biggest difference from when we wrote to a text file are the next two lines:

```
BinaryFormatter formatter = new BinaryFormatter();
formatter.Serialize(output, deck);
```

The `BinaryFormatter` class lets us serialize instances of classes that are marked as serializable to a file stream. So the first line of code above creates a new `BinaryFormatter` object for us and the second line of code above serializes our `Deck` object to the file stream attached to the `deck.dat` file.

Note that the file name we're using to store the object information has a `.dat` extension rather than a `.txt` extension. We do that to show that this isn't a text file; instead, it's a file of bytes. While you could open up the file with a text editor, most of what you'd see would be meaningless to you.

The rest of the code is just as we used when outputting to a text file; we catch any exceptions that are thrown and make sure we always close the output file. Let's read the deck back in from the binary file and print it to the console window:

```
// read deck from file and print to console window
FileStream input = null;
```

```

try
{
    input = File.OpenRead("deck.dat");
    BinaryFormatter formatter = new BinaryFormatter();
    Deck fileDeck = (Deck)formatter.Deserialize(input);
    fileDeck.Print();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    // always close input file
    if (input != null)
    {
        input.Close();
    }
}

```

We're using a `FileStream` object so we can read the serialized `Deck` object from the file stream. We're using the `File` `OpenRead` method instead of the `OpenText` method because we need a `FileStream` object, not a `StreamReader` object. The biggest difference from when we read from a text file are the next two lines:

```

BinaryFormatter formatter = new BinaryFormatter();
Deck fileDeck = (Deck)formatter.Deserialize(input);

```

The `BinaryFormatter` class lets us deserialize instances of classes from a file stream; deserializing simply means converting from a stream of bytes into the actual object. So the first line of code above creates a new `BinaryFormatter` object for us and the second line of code above deserializes our `Deck` object from the file stream attached to the `deck.dat` file. Because the `Deserialize` method actually returns an `Object`, we need to cast the returned value into a `Deck` object.

Once we have the `Deck` object, we tell it to print itself so we can verify that it was saved to the file and retrieved from the file properly. The rest of the code is just as we used when inputting from a text file; we catch any exceptions that are thrown and make sure we always close the input file.

That's all you need to know to write to or read from binary files, so now we can move on to Unity-specific file IO.

## 19.4. PlayerPrefs Class

The `UnityEngine` namespace includes a `PlayerPrefs` class that "Stores and accesses player preferences between game sessions". Example information that you might want to store and access includes saved games and customization of a particular player's profile.

Let's work through an example; we'll start with the fish game we developed in Section 17.3, though we've modified that game to use `UnityEvent` and to display a countdown timer. In our example here, we'll enhance the game to use a high score table of the top 10 scores players achieve in 30 seconds. That means we've changed the game a bit to include a countdown timer and to end the game after 30 seconds,

at which point we'll display the high score table. The table will be sorted, of course, and will persist between executions of the game using the `PlayerPrefs` class.

Although our focus in this section is on the high score table storage and retrieval, there are a number of interesting changes related to game play, so we'll discuss those first.

We added a `TimerFinishedEvent` to the project and added functionality to the `Timer` class so that it invokes this event when the timer finishes (although we left the `Running` property as well). Using an event when the timer finishes matches the way people use timers in the real world much more accurately. Up to this point, a game object with a timer looks at the timer (accessing its `Running` property) on every frame to see if it's done yet; wouldn't that be a horrible way to use a timer in the real world? In the real world, we set a timer, start it running, then forget about it until it tells us its finished, usually with a sound, a vibration, or both. That's how our new `Timer` works, invoking the `TimerFinishedEvent` when the timer finishes so whoever is listening can handle the event appropriately. In our project here, we've refactored the `TeddyBearSpawner` to handle the `TimerFinishedEvent` instead of checking the timer's `Running` property every `Update`, and we've added a `FishGame` script that also listens for that event (from a game timer, not a spawn timer) and handles it as well.

Although we already have an `EventManager` class to connect event invokers and listeners for those events, we don't actually need to use that class for the `TimerFinishedEvent` (in fact, it would be harder to do so). To understand why we don't use the `EventManager` class here, remember our motivation for adding the `EventManager` class in the first place – we did it so objects in the game don't have to know about other objects in the game. In both the `TeddyBearSpawner` and `FishGame` (which runs the game timer and retrieves and displays the high score table when the timer finishes by listening for and handling the `TimerFinishedEvent` for the game timer) scripts, though, the timers are fields in those scripts, so the scripts already know about the timers they contain. It's therefore easier and appropriate in an object-oriented design to just have those scripts add their listeners to their `Timer` fields directly.

Because we've added a game timer to the game, it makes sense to show that timer to the player so they know how much time they have left. Displaying the timer is a job most appropriately assigned to the `HUD` game object, but how does the `HUD` script attached to that game object know the current value of the timer and when it changes (from 30 to 29, for example)? Hopefully by this point you already know the answer! We added a `TimerChangedEvent` to the project and have the `Timer` invoke that event when the value of the timer changes. The `HUD` script listens for that event and changes the timer display when the event is invoked just as it does for the score display and the `PointsAddedEvent`.

In this case, it makes sense to use the `EventManager` because the game timer is added as a component in the `FishGame` script, which is attached to the Main Camera, and the `HUD` game object shouldn't have to know about the Main Camera. This actually leads us to an interesting problem in the `EventManager` class.

Recall that in the `EventManager` class, we already have an `AddListener` method with the method header

```
public static void AddListener(UnityAction<int> handler)
```

to let scripts add listeners for the `PointsAddedEvent`. We would need exactly the same method header for a script to add a listener for the `TimerChangedEvent` (which passes the new timer value as an `int` when it's invoked), and we're not allowed to have two methods with identical method headers in the same class. We changed the method names to `AddPointsAddedEventListener` and `AddTimerChangedEventListener` to solve this problem. Here's the revised `EventManager` class:

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

/// <summary>
/// Manages connections between event listeners and event invokers
/// </summary>
public static class EventManager
{
    /// <summary>
    /// Adds the given event handler as a listener
    /// Game objects tagged as Fish invoke PointsAddedEvent
    /// </summary>
    /// <param name="handler">the event handler</param>
    public static void AddPointsAddedEventListener(
        UnityAction<int> handler)
    {
        // add listener to all fish
        GameObject[] fish = GameObject.FindGameObjectsWithTag("Fish");
        foreach (GameObject currentFish in fish)
        {
            Fish script = currentFish.GetComponent<Fish>();
            script.AddListener(handler);
        }
    }

    /// <summary>
    /// Adds the given event handler as a listener
    /// Game objects tagged as MainCamera invoke TimerChangedEvent
    /// </summary>
    /// <param name="handler">the event handler</param>
    public static void AddTimerChangedEventListener(
        UnityAction<int> handler)
    {
        // add listener to main camera
        GameObject mainCamera = GameObject.FindGameObjectWithTag(
            "MainCamera");
        FishGame script = mainCamera.GetComponent<FishGame>();
        script.AddTimerChangedEventListener(handler);
    }
}
```

And here's our new `FishGame` class (without the high score table processing code):

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

```

using UnityEngine.Events;

/// <summary>
/// Game manager
/// </summary>
public class FishGame : MonoBehaviour
{
    // game timer support
    public const int GameSeconds = 30;
    Timer gameTimer;

    /// <summary>
    /// Awake is called before Start
    /// </summary>
    void Awake()
    {
        // create and start game timer
        gameTimer = gameObject.AddComponent<Timer>();
        gameTimer.AddTimerFinishedEventListener(
            HandleGameTimerFinishedEvent);
        gameTimer.Duration = GameSeconds;
        gameTimer.Run();
    }

    /// <summary>
    /// Adds the given event handler as a listener
    /// </summary>
    /// <param name="handler">the event handler</param>
    public void AddTimerChangedEventListener(UnityAction<int> handler)
    {
        gameTimer.AddTimerChangedEventListener(handler);
    }

    /// <summary>
    /// Handles the game timer finished event
    /// </summary>
    void HandleGameTimerFinishedEvent()
    {
        // display high score table
    }
}

```

The `gameTimer` field invokes two events: the `TimerFinishedEvent` and the `TimerChangedEvent`. The `FishGame` script listens for the `TimerFinishedEvent` so it knows the game is over and it should display the high score table. The `FishGame` script doesn't actually care about the `TimerChangedEvent`, but the `HUD` script does so it can update the countdown timer it displays. The `AddTimerChangedEventListener` method above provides a way for the `HUD` script to actually add (through the `EventManager`) a listener for that event, which is invoked by the `gameTimer` internal to the `FishGame` script.

To make this work properly, we need to make sure that the `gameTimer` field isn't `null` when the `EventManager` class calls the `AddTimerChangedEventListener` method. How do we do that?

Our implementation of the `HUD` script calls a new `AddTimerChangedEventListener` method in the `EventManager` class from within its `Start` method; the `Start` method is where we've done all our

initialization work up to this point in the book. The `Start` method is called just before any of the `Update` method(s) are called for the script, but we don't know in what order the `Start` methods will be called for all our game objects when a scene starts. If we added the timer to our `FishGame` script in the `Start` method and the `Start` method for the `HUD` script attached to the `HUD` game object gets called before the `Start` method for the `FishGame` script attached to the Main Camera, the `gameTimer` field will be `null` when the `HUD` script tries to add a listener for the `TimerChangedEvent`.

If we need to do initialization before the `Start` methods start getting called in a scene, we can use the `Awake` method. The documentation for the `MonoBehaviour Awake` method states that "Awake is always called before any Start functions. This allows you to order initialization of scripts." This is exactly what we need here, so we initialize and start the `gameTimer` in the `FishGame Awake` method so the `HUD` script can safely add its listener to that timer in the `HUD Start` method.

Let's start our work to handle displaying the high scores table. We start by creating another canvas we'll use to display the high score table when the timer has expired. We're using an additional canvas because we want it to overlay the gameplay in the scene when it's displayed, and we want to control when the canvas is visible so that it's only displayed when the game ends.

On the top menu bar, select `GameObject > UI > Canvas`. Rename the new Canvas in the Hierarchy window to `HighScoreCanvas`. Right click `HighScoreCanvas` and select `UI > Image`. Rename the image to `Background` and change its Width and Height to 400. We want the background to be a partially-transparent shade of gray, so click `Color` in the `Image (Script)` component of the Inspector and change the color to gray and the A (alpha) value to 127.

Next, we should add a label at the top of the canvas. Right click `HighScoreCanvas` in the Hierarchy window and select `UI > Text`. Rename the new Text object `Label` in the Hierarchy window. In the Scene view, drag the `Label` to near the top of the canvas and drag the left and right edges to the left and right edges of the canvas. Next, click the center Alignment button in the Paragraph section of the `Text (Script)` component to center the label on the canvas. Change the `Text` value from `New Text` to `High Scores`. Finally, in the Character section of the `Text (Script)` component, change the `Font Style` to `Bold` and the `Font Size` to 24.

Let's add one more text component to the canvas. We'll simply have the player press the Escape key to close the game after they've seen the high score table, so we'll add instructions to do that at the bottom of the canvas. Go ahead and do that now. At this point, the Game view should look like the figure below.



**Figure 19.1. Initial High Scores Canvas**

Okay, let's make it so the game actually exits when the player presses the Escape key. Select Edit > Project Settings > Input from the main menu bar. Add an axis by adding 1 to the number of Axes listed in the Size value and click the bottom axis (which is the new one). Change the name of the axis to Exit, set the positive button to escape, and make sure the Type is set to Key or Mouse Button.

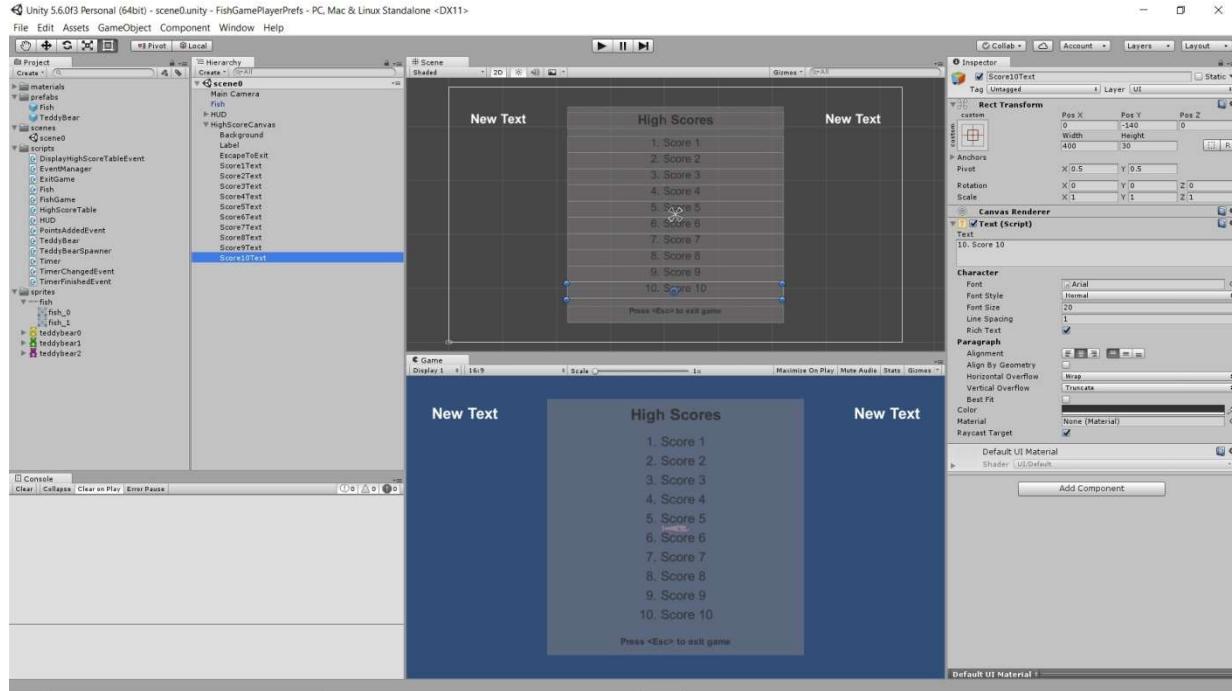
Next, we need to add a script that exits the game when the player presses the Escape key. Right click the scripts folder in the Project window and select Create > C# Script. Rename the script to ExitGame and double-click it to open it in Visual Studio. Here's the completed script:

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

/// <summary>
/// Exits the game
/// </summary>
public class ExitGame : MonoBehaviour
{
    /// <summary>
    /// Update is called once per frame
    /// </summary>
    void Update()
    {
        // exit game as appropriate
        if (Input.GetAxis("Exit") > 0)
        {
            Application.Quit();
        }
    }
}
```

Attach the script to the HighScoreCanvas and build the game by selecting File > Build Settings ... from the main menu bar (remember, we can only close the game down by running it from a built executable). Click the Add Open Scenes button at the bottom right of the Scenes In Build box, then click the Build And Run button at the bottom of the popup. Click Play! in the player when it starts up and press the Escape key to close the game down.

Now let's add the Text elements we'll need to display the top 10 scores. We do this by right-clicking the HighScoreCanvas game object in the Hierarchy window, selecting UI > Text, renaming the new element appropriately (we named ours Score1Text through Score10Text), placing the elements in a reasonable way on the canvas, and changing the text for each element so we could tell them apart! The figure below shows how everything looks after adding those Text elements.



**Figure 19.2. Final High Scores Canvas**

Of course, we don't actually want to start the game with the high score canvas displayed, so we need to disable it when the game starts. The best place to do this is in the `FishGame` script that's attached to the main camera, so we add the following code to a new `Start` method in that script:

```
// disable high score canvas
GameObject highScoreCanvas = GameObject.Find("HighScoreCanvas");
highScoreCanvas.SetActive(false);
```

We originally added this code to the end of the existing `Awake` method in the `FishGame` script, but it turned out that we were deactivating the `HighScoreCanvas` before it could add an event listener it needed to use (we'll get to that later in this section).

When you run the game now, you'll see that it plays normally without displaying the high score canvas. You should also note that pressing the Escape key doesn't exit the game because the script that processes that input is attached to the `HighScoreCanvas` game object, so it's not active when the game starts either.

Recall that our `FishGame` script contains a `HandleGameTimerFinishedEvent` method that gets called when the game timer finishes. We included a comment in that method saying we'd display the high score table at that point, but how should we do that? We can assume at this point that we'll be writing a `HighScoreTable` script that we attach to the `HighScoreCanvas` game object to handle retrieving the saved high score data, adding the current score in the list of high scores as appropriate, saving the

(potentially) revised high score data, setting all the Text elements on the canvas appropriately, and displaying the canvas.

There are several reasonable approaches we could use when the game timer expires. In one such approach, the `FishGame` script could retrieve the `HighScoreTable` component of the `HighScoreCanvas` game object, then call a (yet to be written) `Display` method in that script. This is a reasonable solution, since the `FishGame` script is like the game manager for the game, so it's reasonable for it to know about the game objects in the game. This approach requires even more detailed knowledge about those objects, though, because the `FishGame` script also needs to know that the `HighScoreCanvas` game object contains a `HighScoreTable` component with a `Display` method. Although that single piece of detailed information is okay in this particular game, let's use a more general approach that doesn't require that level of detailed knowledge.

Specifically, we'll add a `DisplayHighScoreTableEvent` to our solution; the `HighScoreTable` script will listen for that event and the `FishGame` script will invoke that event when the game timer finishes. We'll use the `EventManager` to build that connection, of course. This is a better solution because it follows our standard event pattern and because we don't need to have the `FishGame` script know the low-level details of the `HighScoreCanvas` game object.

Here's the code for the new event:

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

/// <summary>
/// An event that indicates that the high score table
/// should be displayed
/// </summary>
public class DisplayHighScoreTableEvent : UnityEvent<int>
{
}
```

Notice that we're using the version of the `UnityEvent` that has a single `int` parameter. That's because we need to include the score for the current game when we invoke the event so the `HighScoreTable` script can include the current score in the list of high scores as appropriate.

Next, we add a `DisplayHighScoreTableEvent` field to the `FishGame` script and add the following method so the `EventManager` can add a listener for that event:

```
/// <summary>
/// Adds the given event handler as a listener
/// </summary>
/// <param name="handler">the event handler</param>
public void AddDisplayHighScoreTableListener(UnityAction<int> handler)
{
    displayHighScoreTableEvent.AddListener(handler);
}
```

We also add the following to the `HandleGameTimerFinishedEvent` method in the `FishGame` script:

```
// display high score table
HUD hudScript = hud.GetComponent<HUD>();
displayHighScoreTableEvent.Invoke(hudScript.Score);
```

To support getting the current game score to update when we invoke the event, we add a `hud` field to the `FishGame` script, mark it with `[SerializeField]`, and populate it in the Inspector. We also add a `score` property to the `HUD` script so we can get the current game score from the `HUD` game object. In this case, we are using detailed information about the `HUD` game object in the `FishGame` script. Although we avoided that approach for the `HighScoreCanvas`, if we're going to store game-level information (like the score) in the `HUD`, we need the `FishGame` script to be able to access that information directly.

Next, we add the following method to the `EventManager` class so the `HighScoreTable` script can add a listener for the `DisplayHighScoreTableEvent`:

```
/// <summary>
/// Adds the given event handler as a listener
/// Game objects tagged as MainCamera invoke DisplayHighScoreTableEvent
/// </summary>
/// <param name="handler">the event handler</param>
public static void AddDisplayHighScoreTableEventListener(
    UnityAction<int> handler)
{
    // add listener to main camera
    GameObject mainCamera = GameObject.FindGameObjectWithTag("MainCamera");
    FishGame script = mainCamera.GetComponent<FishGame>();
    script.AddDisplayHighScoreTableEventListener(handler);
}
```

Now we can get started on the `HighScoreTable` script, which we attach to the `HighScoreCanvas` game object. We start by adding our listener for the `DisplayHighScoreTableEvent` (in the `Awake` method so it's added before the `FishGame` script deactivates the `HighScoreCanvas`) and by enabling the `HighScoreCanvas` when that event is invoked:

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

/// <summary>
/// A high score table
/// </summary>
public class HighScoreTable : MonoBehaviour
{
    /// <summary>
    /// Awake is called before Start
    /// </summary>
    void Awake()
    {
        EventManager.AddDisplayHighScoreTableEventListener(
            HandleDisplayHighScoreTableEvent);
    }
}
```

```

/// <summary>
/// Handles the display high score table event
/// </summary>
/// <param name="score">current game score</param>
void HandleDisplayHighScoreTableEvent(int score)
{
    // display high score table
    gameObject.SetActive(true);
}
}

```

If you run the game now, you'll see that the HighScoreCanvas appears when the game timer expires. It would be nicer to pause the game once that happens, so we'll take care of that before we finish off the game.

For now, though, let's (finally!) actually work on storing, retrieving, and displaying the high score table using the `PlayerPrefs` class. Before we start implementing the details, let's talk about how the `PlayerPrefs` class works in general.

The `PlayerPrefs` class stores pairs of keys and values. We can think of the key as the "name" of a piece of information we're storing and the value as the actual information. For example, if we wanted to store the player's name (Doofus42, say), we could use "Player Name" as the key and "Doofus42" as the value. In that case, we'd use

```
PlayerPrefs.SetString("Player Name", "Doofus42");
```

to save that key/value pair. Similarly, if we want to retrieve the player name from storage, we'd use

```
string playerName = PlayerPrefs.GetString("Player Name");
```

The key we use is always a `string`, and the only data types we can save for the value are `float`, `int`, or `string`. We can use the `PlayerPrefs.HasKey` method to find out if a particular key has a value associated with it, which we might want to do before setting a key/value pair to make sure we don't destroy a previous key/value pair with the same key. If we call `GetFloat`, `GetInt`, or `GetString` with a key that doesn't exist in the preferences (there's no key/value pair for that key), `GetFloat` returns `0.0f`, `GetInt` returns `0`, and `GetString` returns an empty string.

Using `PlayerPrefs` to store a complex object (like a high score table) is somewhat awkward because we can only retrieve a single value for a specific key. Alternatively, we could use a standard C# binary file (like we learned about in Section 19.3) to store and access this information, but that approach has a significant drawback. Specifically, different Operating Systems have different rules about where applications can write data to device storage. Rather than trying to handle all those different rules in our code, we can simply use `PlayerPrefs` instead. It's definitely easier for us to write code that has a single value per key rather than trying to handle all the different devices we can deploy a Unity game to.

So how do we store the set of 10 high scores in a single `float`, `int`, or `string`? By using a comma-separated value (CSV) string, of course! Let's look at a couple of helper methods we'll use to work with those CSV strings, then we'll get back to our `HighScoreTable HandleDisplayHighScoreTableEvent` method.

Our first helper method converts a string to a list of high scores:

```
/// <summary>
/// Extracts a list of high scores from the given csv string
/// </summary>
/// <returns>list of high scores</returns>
/// <param name="csvString">csv string of high scores</param>
List<int> GetHighScoresFromCsvString(string csvString)
{
    List<int> scores = new List<int>();

    // only need to process a non-empty string
    if (!string.IsNullOrEmpty(csvString))
    {
        // extract scores and add to list
        string[] scoreArray = csvString.Split(',');
        foreach (string score in scoreArray)
        {
            scores.Add(int.Parse(score));
        }
    }

    return scores;
}
```

The method starts by creating a new, empty list of scores. We only need to process non-empty strings (empty strings don't contain any scores), so we use an if statement to check for that. The first line of code in the if body uses the `String Split` method to extract the scores (as strings) from the CSV string. We pass two arguments into the `Split` method: the string we want to split and the separator (or delimiter) – in this case, a comma – that separates the values in the string. The method returns an array of the values as strings. The `foreach` loop walks the array of values and adds each one to the `scores` list, parsing each value string into an `int` along the way. The last line of code in the method returns the list of scores.

Our second helper method converts a list of high scores to a csv string (we had to add a `using` directive for the `System.Text` namespace to get this to compile):

```
/// <summary>
/// Converts a list of high scores to a csv string
/// </summary>
/// <returns>csv string</returns>
/// <param name="scores">list of high scores</param>
string GetCsvStringFromHighScores(List<int> scores)
{
    StringBuilder csvString = new StringBuilder();

    // append the scores and commas to the csv string
    for (int i = 0; i < scores.Count - 1; i++)
    {
        csvString.Append(scores[i]);
        csvString.Append(',');
    }
}
```

```

    // add last score (with no following comma)
    csvString.Append(scores[scores.Count - 1]);

    return csvString.ToString();
}

```

This method uses a `StringBuilder` to build up our string; remember, using that approach lets us avoid creating lots of new string objects as we go along because strings are immutable. The for loop adds each score, followed by a comma, to the string (except for the last score). We then add the last score to the string and return the string, using `ToString` to convert it to a `string` (instead of a `StringBuilder`).

Our final helper method doesn't actually deal with CSV strings, but it does add the current game score to the list of high scores as appropriate:

```

/// <summary>
/// Adds the given score to the list. If the score table is already
/// full, inserts the score in the appropriate location and drops the
/// lowest score from the list or does nothing if the given score is
/// lower than all the scores in the list
/// </summary>
/// <param name="score">the score to add</param>
/// <returns>true if the score was added, false otherwise</returns>
public bool AddScore(int score, List<int> scores)
{
    // make sure we should add the score
    if (scores.Count < MaxNumScores || 
        score > scores[MaxNumScores - 1])
    {
        // make sure we don't grow the list past full size
        if (scores.Count == MaxNumScores)
        {
            scores.RemoveAt(MaxNumScores - 1);
        }

        // find location at which to add the score
        int addLocation = 0;
        while (addLocation < scores.Count &&
               scores[addLocation] > score)
        {
            addLocation++;
        }

        // insert the score in the list at the appropriate location
        scores.Insert(addLocation, score);

        return true;
    }
    else
    {
        return false;
    }
}

```

We make the `AddScore` method return `true` if the given score is actually added to the high scores and `false` otherwise. We do that so the consumer of this method only saves the high scores back to player prefs if the list of high scores has changed. At this point in the book, you should be able to understand how the method above works. If not, write down a list of 10 (we've added a `MaxNumScores` constant, set to 10, to our `HighScoreTable` script) high scores on a piece of paper and add a new high score that falls in the middle of the list using the code above as your algorithm for doing that.

Let's finish implementing the `HighScoreTable` `HandleDisplayHighScoreTableEvent` method, which of course uses the above helper methods.

```
/// <summary>
/// Handles the display high score table event
/// </summary>
/// <param name="score">current game score</param>
void HandleDisplayHighScoreTableEvent(int score)
{
    // retrieve high scores from storage
    List<int> scores = GetHighScoresFromCsvString(
        PlayerPrefs.GetString("High Scores"));
}
```

The line of code above retrieves the value in player prefs associated with the "High Scores" key (that value will be an empty string the first time the game runs because we've never saved a value for the High Scores key) and passes that value into the helper method that converts it to a list of scores.

```
// add current score and save as appropriate
bool scoreAdded = AddScore(score, scores);
if (scoreAdded)
{
    PlayerPrefs.SetString("High Scores",
        GetCsvStringFromHighScores(scores));
    PlayerPrefs.Save();
}
```

The block of code above adds the current game score to the list of scores using the `AddScore` method discussed above. If the score was actually added to the high scores (remember, that's why the `AddScore` method returns `bool` instead of `void`), the block of code also assigns a new value (including the new score in the appropriate place in the csv string) to the "High Scores" key and saves the new high scores back to player prefs. Because writing to a file takes a relatively long time compared to other program operations, we only want to pay that performance price when we actually need to.

There's actually another reason to save back to player prefs at this point. The documentation for the `PlayerPrefs` `Save` method tells us "By default Unity writes preferences to disk during `OnApplicationQuit()`." Although we're going to quit the application soon (when the player presses the Escape key after looking at the high scores table), it's better to make sure we save the information now just in case the game crashes or something else goes wrong.

```
// disable all score Text elements
Text[] textElements = gameObject.GetComponentsInChildren<Text>();
foreach (Text textElement in textElements)
{
    if (textElement.name.Contains("Score"))
    {
```

```
        textElement.enabled = false;  
    }  
}
```

The block of code above disables all the Text elements for the scores on the HighScoreCanvas (we had to add a using directive for the `UnityEngine.UI` namespace to get this to compile). We couldn't just disable all the Text elements on the canvas because the High Scores label and the Press `<Esc>` to exit game message are also Text elements on the canvas.

```
// enable and populate Text elements for high scores
for (int i = 0; i < scores.Count; i++)
{
    string textElementName = "Score" + (i + 1) + "Text";
    int textElementIndex = GetTextElementIndex(
        textElementName, textElements);
    textElements[textElementIndex].enabled = true;
    textElements[textElementIndex].text = (i + 1) + ". " + scores[i];
}
```

The block of code above re-enables those Text elements that actually contain a high score (there could be fewer than `MaxNumScores` high scores saved at this point). First, we build the name of the Text element we want to process by adding 1 to `i` because our for loop indexes are 0-based but our Text element names are 1-based. We then use an additional helper method we wrote (see the code accompanying the chapter for the details) to find the index of the Text element with the name we're looking for. Finally, we enable that text element so it will be displayed and set the text appropriately.

```
// display high score table  
gameObject.SetActive(true);  
}
```

The final line of code sets the HighScoreCanvas game object to be active so it's displayed.

When you run the game for the first time, you should get something like the figure below (though of course your score will hopefully be higher!).



**Figure 19.3.** High Score Display After First Play

Okay, we're almost done solving this problem! The last thing we want to do is pause the gameplay when the high scores are displayed.

If you do a web search on Unity "Pause Game", you'll find a number of reasonable suggestions. The best suggestion is to set `Time.timeScale` to 0. This does pause the game, but unfortunately it also seems to make the game unresponsive to user input, so the game won't close when the player presses the Escape key.

We said "seems to" above because this doesn't actually make the game unresponsive to user input! If we replace `Input.GetAxis("Exit") > 0` with `Input.GetKeyDown(KeyCode.Escape)` in our `ExitGame` script, the game exits as before when the player presses the Escape key. What's going on?

This took some research to figure out, but it turns out that the `Input.GetAxis` method applies some input smoothing to the input, so when we set the `timeScale` to 0 that axis input never changes from 0. Although we could use the `GetKeyDown` method approach shown above, we're big fans of using the named input axes because we think it makes our code more readable. Luckily, there's a way to do that! We can simply use the `Input.GetAxisRaw` method instead, which "Returns the value of the virtual axis identified by `axisName` with no smoothing filtering applied". That's just what we need here, so setting `Time.timeScale` to 0 in our `HighScoreTable.HandleDisplayHighScoreTableEvent` method and changing `.GetAxis` to `.GetAxisRaw` in our `ExitGame.Update` method completes our solution to the problem in this section.

So where is the high score information in the player prefs actually saved? That depends on the OS you're building for, so you should read the information about the storage location in the `PlayerPrefs` documentation. The storage location for most Operating Systems is partially based on the Company Name and Product Name for the game, so you should be sure to set those by selecting Edit > Project Settings > Player from the main menu bar. The Company Name and Product Name values are at the top of the Inspector.

## 19.5. Game Configuration Storage

The previous section showed how we can use `PlayerPrefs` to store information that potentially changes as the player plays our game multiple times (though we could of course also save multiple changes in a single play session, as the player collects upgrades, for example).

We may also want to retrieve read-only information about the configuration of the game itself. Including game configuration information in a separate file (instead of as constants as we've been doing up to this point) is a good approach for a number of reasons. It lets non-programmers help tune the game during playtesting, because changes they make to the configuration file immediately propagate into the game without any programmer intervention. In addition, it also helps support patching the game after it's released, since game developers can simply include the new configuration file in the patch instead of having to release a new executable for the entire game.

For our example, we'll again start with the fish game we developed in Section 17.3. We include the revised `Timer`, `TimerFinishedEvent`, and `EventManager` classes from the previous section, but only to control teddy bear spawning and to add points to the player score; we don't include a game timer or a high score table in this example. We'll enhance the game by using a configuration file that contains the following information: the fish move units per second, the minimum teddy bear impulse force, the

maximum teddy bear impulse force, the minimum teddy bear spawn delay, the maximum teddy bear spawn delay, and the point value for each bear.

Next, we need to decide what format we'll use for the configuration data file and where we'll store it. We could save the file as a binary file as discussed in section 19.3, which would make it harder for players to modify. It's more common during development to save these kinds of files as CSV or XML files so they're easy for designers (rather than programmers) to change to tune the game. Using a CSV or XML file makes it easy for players to modify, but that may not be so bad; players can play with the different values in the file to change the gameplay<sup>33</sup>. For this example, we'll use an XML file that we store in the same location as the executable for our game.

Let's start with a `ConfigurationData` class that we'll use to store the information listed above. This class is really just a data container for the configuration data we read from the file. We're going to use something called a *data contract* to specify what's contained in the XML file that contains the configuration data that corresponds to this class. Basically, the data contract contains a set of *data member* elements that give the name of XML elements, the data type those XML elements should contain, and their location in the XML file. Here's the code for our `ConfigurationData` class:

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization;
using UnityEngine;

/// <summary>
/// A class providing access to configuration data
/// </summary>
[DataContract]
public class ConfigurationData
{
```

We mark the class with a `DataContract` attribute like we marked our `Deck` with the `Serializable` attribute in Section 19.3.

```
#region Fields

[DataMember(Order = 0)]
int fishMoveUnitsPerSecond = 5;

[DataMember(Order = 1)]
float minTeddyImpulseForce = 3;

[DataMember(Order = 2)]
float maxTeddyImpulseForce = 5;

[DataMember(Order = 3)]
int minBearSpawnDelay = 1;

[DataMember(Order = 4)]
int maxBearSpawnDelay = 2;
```

---

<sup>33</sup> We recently released our Battle Paddles game on [www.burningteddy.com](http://www.burningteddy.com). Because we didn't get a chance to tune the game before we closed Peak Game Studios, we included an XML file that players can use to tune the game as they see fit.

```
[DataMember(Order = 5)]  
int bearPoints = 10;  
  
#endregion
```

Each of the fields in the class is marked with a `DataMember` attribute that tells where the field will be in the XML file (that's the `Order` part), the name of the XML element that contains the field, and the data type that should be included for the XML element. In the code above, we also initialized each of those fields with a default value.

```
#region Properties  
  
    /// <summary>  
    /// Gets the fish move units per second  
    /// </summary>  
    public int FishMoveUnitsPerSecond  
    {  
        get { return fishMoveUnitsPerSecond; }  
    }  
  
    /// <summary>  
    /// Gets the min teddy impulse force  
    /// </summary>  
    public float MinTeddyImpulseForce  
    {  
        get { return minTeddyImpulseForce; }  
    }  
  
    /// <summary>  
    /// Gets the max teddy impulse force  
    /// </summary>  
    public float MaxTeddyImpulseForce  
    {  
        get { return maxTeddyImpulseForce; }  
    }  
  
    /// <summary>  
    /// Gets the min bear spawn delay  
    /// </summary>  
    public int MinBearSpawnDelay  
    {  
        get { return minBearSpawnDelay; }  
    }  
  
    /// <summary>  
    /// Gets the max bear spawn delay  
    /// </summary>  
    public int MaxBearSpawnDelay  
    {  
        get { return maxBearSpawnDelay; }  
    }  
  
    /// <summary>  
    /// Gets the number of points a bear is worth  
    /// </summary>  
    public int BearPoints
```

```

{
    get { return bearPoints; }
}

#endregion
}

```

Our class exposes properties for each of the pieces of configuration data so consumers of the class can get those values.

Now let's look at the XML file (called ConfigurationData.xml) that contains the configuration data:

```

<ConfigurationData xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://schemas.datacontract.org/2004/07/">
    <fishMoveUnitsPerSecond>5</fishMoveUnitsPerSecond>
    <minTeddyImpulseForce>3</minTeddyImpulseForce>
    <maxTeddyImpulseForce>5</maxTeddyImpulseForce>
    <minBearSpawnDelay>1</minBearSpawnDelay>
    <maxBearSpawnDelay>2</maxBearSpawnDelay>
    <bearPoints>10</bearPoints>
</ConfigurationData>

```

As you can see, the XML file contains the appropriate data elements in the correct order, with valid values for each data element.<sup>34</sup>

Next, we need a static class that other scripts in the game can use to access the configuration data. We called that class `ConfigurationUtils`; here's the code:

```

using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization;
using System.Xml;
using UnityEngine;

```

We're again using a variety of namespaces that let us process the XML file.

```

/// <summary>
/// Provides utility access to configuration data
/// </summary>
public static class ConfigurationUtils
{
    #region Fields

    static ConfigurationData configurationData;

    #endregion
}

```

We have a single static field we use to hold the configuration data we load from the XML file.

---

<sup>34</sup> We actually generated the XML file by adding a temporary method to the `ConfigurationData` class telling it to save its default values so we made sure the XML header was correct.

```
#region Properties

    /// <summary>
    /// Gets the fish move units per second
    /// </summary>
    public static int FishMoveUnitsPerSecond
    {
        get { return configurationData.FishMoveUnitsPerSecond; }
    }

    /// <summary>
    /// Gets the min teddy impulse force
    /// </summary>
    public static float MinTeddyImpulseForce
    {
        get { return configurationData.MinTeddyImpulseForce; }
    }

    /// <summary>
    /// Gets the max teddy impulse force
    /// </summary>
    public static float MaxTeddyImpulseForce
    {
        get { return configurationData.MaxTeddyImpulseForce; }
    }

    /// <summary>
    /// Gets the min bear spawn delay
    /// </summary>
    public static int MinBearSpawnDelay
    {
        get { return configurationData.MinBearSpawnDelay; }
    }

    /// <summary>
    /// Gets the max bear spawn delay
    /// </summary>
    public static int MaxBearSpawnDelay
    {
        get { return configurationData.MaxBearSpawnDelay; }
    }

    /// <summary>
    /// Gets the number of points a bear is worth
    /// </summary>
    public static int BearPoints
    {
        get { return configurationData.BearPoints; }
    }

#endregion
```

These are essentially the same properties as the properties exposed by the `ConfigurationData` class, but the properties above are static, so they provide static access to the configuration data through the `configurationData` field.

```
#region Public methods

/// <summary>
/// Initializes the configuration data by reading the data from
/// an XML configuration file
/// </summary>
public static void Initialize()
{
    // deserialize configuration data from file into internal object
    FileStream fs = null;
    try
    {
        FileStream fs = new FileStream("ConfigurationData.xml",
            FileMode.Open);
    
```

The above lines of code open a file stream from our configuration data XML file.

```
XmlDictionaryReader reader = XmlDictionaryReader.CreateTextReader(
    fs, new XmlDictionaryReaderQuotas());
DataContractSerializer ser = new DataContractSerializer(
    typeof(ConfigurationData));
configurationData = (ConfigurationData)ser.ReadObject(
    reader, true);
```

The block of code above lets us use our data contract and the XML file to read the data from the XML file into our `configurationData` field. Feel free to read the documentation for the classes and methods we're using above to dig into the details.

```
    reader.Close();
}
finally
{
    // always close input file
    if (fs != null)
    {
        fs.Close();
    }
}
#endregion
}
```

Finally, we close our reader and file stream as we always do.

Of course, we still need to call the `Initialize` method above at the start of the game to read in the configuration data. We do that by adding a new `FishGame` script and attaching it to the Main Camera. Here's the code for that script:

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

```

/// <summary>
/// Game manager
/// </summary>
public class FishGame : MonoBehaviour
{
    /// <summary>
    /// Awake is called before Start
    /// </summary>
    void Awake()
    {
        ConfigurationUtils.Initialize();
    }
}

```

Now that we have the configuration data from the file accessible through the properties of our `ConfigurationUtils` class, we can replace the constants we have sprinkled throughout our code to use the file information instead. You should look at the accompanying code to see the details for all those changes, but here's a summary:

1. In the `TeddyBearSpawner` script, replaced the `MinSpawnDelay` constant with a `minSpawnDelay` variable and set the variable to `ConfigurationUtils.MinBearSpawnDelay` at the beginning of the `Start` method
2. In the `TeddyBearSpawner` script, replaced the `MaxSpawnDelay` constant with a `maxSpawnDelay` variable and set the variable to `ConfigurationUtils.MaxBearSpawnDelay` at the beginning of the `Start` method
3. In the `Fish` script, changed the `moveUnitsPerSecond` field from public to private and set the field to `ConfigurationUtils.FishMoveUnitsPerSecond` at the beginning of the `Start` method
4. In the `Fish` script, changed the `bearPoints` field from public to private and set the field to `ConfigurationUtils.BearPoints` at the beginning of the `Start` method
5. In the `TeddyBear` script, removed the `minImpulseForce` field and used `ConfigurationUtils.MinTeddyImpulseForce` in the `Start` method in the code that sets the magnitude of the force vector
6. In the `TeddyBear` script, removed the `maxImpulseForce` field and used `ConfigurationUtils.MaxTeddyImpulseForce` in the `Start` method in the code that sets the magnitude of the force vector

At this point, we can change any of the values in the XML file and we'll see the new values in effect in the game after we save our XML changes and rerun the game. Go ahead, try it!

# Chapter 20. Putting It All Together

Throughout the book we've learned about lots of the bits and pieces we can use to build games using C# and Unity. We've even built a couple of small games, but we haven't really developed a complete game from scratch. In this chapter, we'll build an actual game that uses many of the concepts you've learned in previous chapters (and yes, you might even learn a few new things as well).

As a warning, the game we'll be building here is pretty simple; it's about the scope you'd expect for a final project at the end of a first or second game programming course at a college or university. That means that it's got a simple menu system, straightforward gameplay, and only very rudimentary Artificial Intelligence (AI). On the other hand, it gives us a chance to put everything together into a game, and you can of course use the ideas we implement in this chapter as a foundation for more complicated games you build on your own.

## 20.1. Understand the Problem

Okay, so we're going to build a game. What will that game be, and how will it work? Here's the game description:

Implement a game called Feed the Teddies. In the game, the player moves a burger avatar around the screen, shooting french fries at teddy bears. The teddy bears aren't defenseless, though, they shoot back! The player's burger takes damage when it collides with a teddy bear or a teddy bear projectile. Selected game difficulty (Easy, Medium, or Hard) determines how smart the teddy bears are, how quickly they move, and how quickly they're spawned. When the game finishes, either because the player ran out of health or because the game timer expired, the player discovers whether or not they just achieved a new high score.

The main menu will give the player the ability to play, see their current best score, or quit. The difficulty menu will let the player select between Easy, Medium, and Hard difficulties for the game. The high score menu will simply show the current high score; the player will click a button to close the menu and return to the main menu. We'll also include a pause menu that the player can open during gameplay by pressing the Escape key. The player will be able to either resume the game or quit to the main menu from the pause menu.

## 20.2. Design a Solution (Menus)

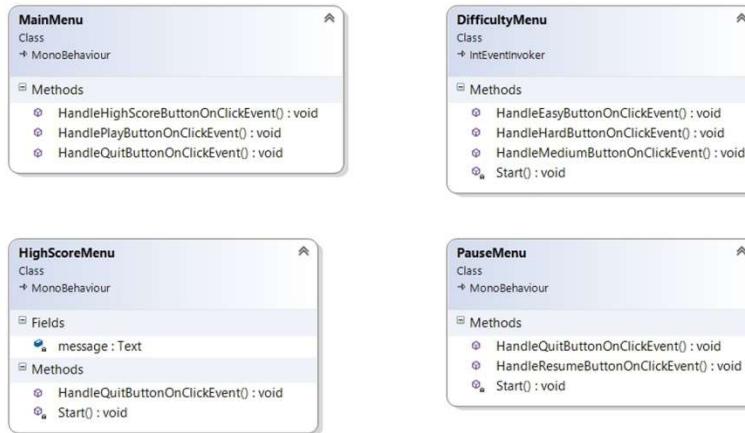
We'll approach our design, implementation, and testing in five iterations. Specifically, we'll have iterations for menus, basic gameplay, full gameplay, sounds, and XML configuration data.

It should be obvious why we're thinking of the main menu, the difficulty menu, and the pause menu as menus. Perhaps less intuitively, we're also thinking of the high score menu as a menu because it will behave just like any other menu, with a clickable button to take some action (in this case, close the menu and return to the main menu).

Just as we did in Chapter 17, we'll make our main menu and difficulty menu separate scenes in our game. Our pause menu will be a "popup menu" that appears above the gameplay scene, so we'll make that a prefab we can create and destroy as necessary. This is a different approach from the one we used in the previous chapter for our high score table, but recall that in our previous approach we included a

canvas in the scene, then modified whether or not it was displayed as appropriate. Although our game here only has one gameplay scene, you should be able to easily imagine a game with many gameplay scenes (e.g., levels). We don't want to have to add the pause menu canvas to each of those scenes in the editor, so a prefab is the better way to go. Finally, our high score menu will also be a prefab. We'll display it both from the main menu and at the end of a game, so having a prefab we can use in both places will be a good approach.

What scripts will we need? It's reasonable to plan to have `MainMenu`, `DifficultyMenu`, `PauseMenu`, and `HighScoreMenu` scripts to handle button clicks and any other required processing (like displaying the high score). The UML for those scripts is provided below.



**Figure 20.1. UML for Menu Scripts**

Now that we have our initial design for the menu system done, we can move on to our menu test cases.

### 20.3. Write Test Cases (Menus)

Our test cases include testing the behavior of all four menus.

#### Test Case 1

##### **Clicking Quit Button on Main Menu**

Step 1. Click Quit button on Main Menu. Expected Result: Exit game

#### Test Case 2

##### **Clicking High Score Button on Main Menu**

Step 1. Click High Score button on Main Menu. Expected Result: Move to High Score Menu

Step 2. Click X in corner of player. Expected Result: Exit game

#### Test Case 3

##### **Displaying High Score on High Score Menu**

Step 1. Click High Score button on Main Menu. Expected Result: Move to High Score Menu

Step 2. If no games have been played yet, the High Score Menu displays a No games played yet message. Otherwise, the High Score Menu displays the highest score achieved so far

Step 3. Click X in corner of player. Expected Result: Exit game

#### **Test Case 4**

##### **Clicking Quit Button on High Score Menu**

Step 1. Click High Score button on Main Menu. Expected Result: Move to High Score Menu

Step 2. Click Quit button on High Score Menu. Expected Result: Move to Main Menu

Step 3. Click Quit button on Main Menu. Expected Result: Exit game

#### **Test Case 5**

##### **Clicking Play Button on Main Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click X in corner of player. Expected Result: Exit game

#### **Test Case 6**

##### **Clicking Easy Button on Difficulty Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Click X in corner of player. Expected Result: Exit game

#### **Test Case 7**

##### **Clicking Medium Button on Difficulty Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Medium button on Difficulty Menu. Expected Result: Move to gameplay screen for medium game

Step 3. Click X in corner of player. Expected Result: Exit game

#### **Test Case 8**

##### **Clicking Hard Button on Difficulty Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Hard button on Difficulty Menu. Expected Result: Move to gameplay screen for hard game

Step 3. Click X in corner of player. Expected Result: Exit game

#### **Test Case 9**

##### **Clicking Resume Button on Pause Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Press Escape key. Expected Result: Game paused and Pause Menu displayed on top of game

Step 4. Press Resume button on Pause menu. Expected Result: Pause Menu removed and game unpause

Step 5. Click X in corner of player. Expected Result: Exit game

#### **Test Case 10**

##### **Clicking Quit Button on Pause Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Press Escape key. Expected Result: Game paused and Pause Menu displayed on top of game

Step 4. Press Quit button on Pause menu. Expected Result: Move to Main Menu

Step 5. Click Quit button on Main Menu. Expected Result: Exit game

## 20.4. Write the Code (Menus)

Create a new 2D Unity project and add folders for prefabs, scenes, scripts, and sprites. Save the current scene as MainMenu.

Right click the sprites folder in the Project window and select Create > Folder; call the new folder menus. We're going to keep our menu sprites separate from our gameplay sprites in this game. You'll find that as you build more complicated games, you'll need to add subfolders to organize your assets for the game. Games can have hundreds or even thousands of different sprites, and putting all those sprites into a single folder would be madness!

Next, we add the sprite in the figure below to our sprites\menus folder (our sprite is named quitmenubutton).



**Figure 20.2. Quit Button Sprite**

This sprite actually has two images. The image on the left is the unhighlighted image and the one on the right is the highlighted image. Instead of using the default Button color tinting behavior provided in Unity, we're going to have Unity change the button sprite as the player moves the mouse on to and off of the button.

To support that functionality, we actually need to treat the image as a sprite strip (like we did for fire and explosion animations in Chapter 16) because it contains two different sprites. Click the quit menu button sprite you just added in the Project window. In the Inspector, change the Sprite Mode to Multiple. Click the Sprite Editor button, select Slice near the upper left corner of the popup, change the Type to Grid by Cell Count, change C to 2, and click the Slice button. Click the Apply button near the top middle of the popup and close the sprite editor. If you expand the sprite in the Project window, you'll see that it's been split into two sprites.

Now we'll add the button to the scene. Create an Image for the Quit Button in the scene and drag the quitmenubutton\_0 sprite into the Source Image in the Inspector. Add a Button component to the Image. Place the Quit Button in a reasonable place in the scene by changing the X and Y locations of the image (remember, we'll have 3 menu buttons on the main menu). Change the name of the Image in the Hierarchy window to QuitMenuButton.

If you run the game now, you'll see the slight change in color when we move the mouse on to and off of the button. Let's make it so the button changes sprites instead. Select the QuitMenuButton, go to the Button component in the Inspector, and change Transition to Sprite Swap. Drag the quitmenubutton\_1 sprite onto the Highlighted Sprite field. When you run the game again, you'll see that the sprite changes between the unhighlighted and highlighted sprites appropriately. Cool.

Now we need to make the button do something when we click it. In Chapter 17, we wrote a `MainMenu` script that processed the clicks on each of the buttons in the menu, and we'll use that same approach here (we put it in a new scripts\menus subfolder):

```

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

/// <summary>
/// Listens for the OnClick events for the main menu buttons
/// </summary>
public class MainMenu : MonoBehaviour
{
    /// <summary>
    /// Handles the on click event from the quit button
    /// </summary>
    public void HandleQuitButtonOnClickEvent()
    {
        Application.Quit();
    }
}

```

Attach the `MainMenu` script to the canvas that contains the `QuitMenuButton` (now would be a good time to change the name of that canvas to `MainMenuCanvas`). Add the `MainMenu` `HandleQuitButtonOnClickEvent` method as a listener for the `On Click ()` event for the `QuitMenuButton` in the Inspector.

Remember, we need to use `File > Build Settings ...` and run our game in the player for the `Application` `Quit` method to work, so do that now; we put our built games into a separate `Build` folder. The game should close when you click the `Quit` button (which lets us pass Test Case 1).

Let's work on passing Test Case 2, which is the move to the High Score Menu from the Main Menu. Add a sprite for the high score menu button to the `sprites\menus` folder (our sprite is named `highscoremenubutton`) and use the Sprite Editor to slice it into two sprites like we did above for the quit menu sprite. Add an `Image` to the scene and rename the `Image` to `HighScoreMenuButton`. Set the `Source Image` to the `highscoremenubutton_0` sprite. Add a `Button` component and set it up to change to the highlighted sprite as appropriate. Run the game to confirm that highlighting works properly.

Next, we need to build the prefab for the High Score Menu. Add a new canvas to the scene like we did in the previous chapter for the high scores canvas and name the canvas `HighScoreMenu`. Add an `Image` to the canvas, center the image in the scene, change the name of the image to `Background`, set the `Width` to 400, set the `Height` to 300, and make the `Color` an opaque gray.

Add a `Text` element to the canvas for a high score label. Change the location, text, font size, and paragraph alignment appropriately to put the label near the top of the canvas. Change the name of the `Text` element to `Label`.

Add another `Text` element to the canvas for the high score message. Change the location, text, font size, and paragraph alignment appropriately to put the message near the middle of the canvas. Change the name of the `Text` element to `Message`.

When we were adding the canvas and the text elements, those were on top of the main menu buttons in both the Scene and Game views. We ran the game in the editor, though, and the canvas ended up below the main menu buttons in both those windows. Because we're going to want the High Score Menu to appear above the Main Menu when the player clicks the High Score button, we need to fix this.

There are of course a variety of ways to solve this problem, although we will point out that using Sorting Layers isn't one of them for a Canvas object (for those of you who remember Chapter 9). We're going to actually do some special processing as we move from the Main Menu to the High Score Menu (deactivating the Main Menu canvas containing the main menu buttons). It turns out that it's fairly common that we need to do some special processing as we navigate between menus in our games, especially those with complicated menu systems. Rather than spreading that processing logic out throughout the menu scripts in our game, we'll add a centralized `MenuManager` class that handles that for us.

Here's our initial code for that class (which we put in the `scripts\menus` subfolder):

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

/// <summary>
/// Manages navigation through the menu system
/// </summary>
public static class MenuManager
{
    /// <summary>
    /// Goes to the menu with the given name
    /// </summary>
    /// <param name="name">name of the menu to go to</param>
    public static void GoToMenu(MenuName name)
    {
        switch (name)
        {
            case MenuName.Difficulty:
                break;
            case MenuName.HighScore:

                // deactivate MainMenuCanvas and instantiate prefab
                GameObject mainMenuCanvas = GameObject.Find("MainMenuCanvas");
                mainMenuCanvas.SetActive(false);
                Object.Instantiate(Resources.Load("HighScoreMenu"));
                break;
            case MenuName.Main:
                break;
            case MenuName.Pause:
                break;
        }
    }
}
```

We made this a static class so the menu scripts can easily access the methods in the class. Note that we also renamed the first canvas we added to our scene to `MainMenuCanvas` so we can easily find the canvas to deactivate it when going to the High Score Menu. Also note that we added a `MenuName` enumeration to our project (also in the `scripts\menus` folder) to make our menu code more readable.

We haven't included code for most of the menus at this point, though we did include code for the `MenuName.HighScore` case. The first two lines of code in that case find the main menu canvas and

deactivate it; this works fine for now, though we'll have to modify it slightly before we're done with the game.

The third line of code uses the `Resources` class to instantiate an instance of the `HighScoreMenu` prefab. The problem we need to solve here is that we need to instantiate a prefab that isn't in the scene and hasn't been assigned to a field in any of our scripts. We don't want to add an instance of the `HighScoreMenu` prefab to each of our scenes (even as an inactive game object) because we'd have the same problem we discussed with the pause menu; we'd have to include the prefab in every single scene in the game. The `Resources Load` method looks for the asset with the given name in any folder named `Resources` in the project Assets folder, so we added a `Resources` folder under the `prefabs` folder in the Project window and created a `HighScoreMenu` prefab there by dragging the `HighScoreMenu` canvas from the Hierarchy window and dropping it into that folder. After creating the prefab, we deleted the `HighScoreMenu` canvas from the current scene. You should read the `Resources` documentation to learn more about the tradeoffs associated with using this class, but it's the right choice for us here.

Now we can add a method to the `MainMenu` class to handle clicks on the high score button:

```
/// <summary>
/// Handles the on click event from the high score button
/// </summary>
public void HandleHighScoreButtonOnClickEvent()
{
    MenuManager.GoToMenu(MenuName.HighScore);
}
```

Add the new method as a listener for the `On Click ()` event in the `HighScoreMenuItem` game object and run the game. Execute Test Case 2, clicking the High Score button on the main menu, then closing the player at the High Score Menu. The test case should pass fine.

Test Case 3 also partially works at this point because we made the default text for the `Message` element the text we want to display if we don't have a high score saved yet. We'd like to finish off this test case, though, so let's add the required functionality to retrieve a high score from player prefs and display it. To test the portion of the case where there actually is a saved high score, we'll "seed" a value into player prefs to test the score display, then clear the key so it doesn't mess up later functionality.

Drag the `HighScoreMenu` prefab from the Project window into the Hierarchy window so we can modify the prefab. Create and implement the `HighScoreMenu` script below and add it to the `HighScoreMenu` canvas in the Hierarchy window.

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

/// <summary>
/// Retrieves and displays high score and listens for
/// the OnClick events for the high score menu button
/// </summary>
public class HighScoreMenu : MonoBehaviour
{
```

```
[SerializeField]
Text message;

/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // temporary code
    PlayerPrefs.SetInt("High Score", 3000);

    // retrieve and display high score
    if (PlayerPrefs.HasKey("High Score"))
    {
        message.text = "High Score: " + PlayerPrefs.GetInt("High Score");
    }
    else
    {
        message.text = "No games played yet";
    }
}
}
```

Drag the Message element from the HighScoreMenu canvas in the Hierarchy window onto the Message field of the HighScoreMenu script in the Inspector. Click the Prefab Apply button near the top of the Inspector to apply your changes to the prefab. Delete the HighScoreMenu from the scene and run the game.

You should get a High Score Menu displaying a high score of 3000. Now change the temporary line of code in the `Start` method to

```
PlayerPrefs.DeleteKey("High Score");
```

and run the game again. You should get a High Score Menu displaying the No games played yet message.

At this point, we've tested both possibilities for a high score in Test Case 3 and the High Score key is currently not saved in player prefs (since there haven't been any games played yet). Delete the temporary code from the `Start` method.

It's actually fairly common to add temporary code that lets us test functionality that hasn't been fully implemented yet (like having a high score saved from an actual game in our current project). This is a really good approach to use to test the code we're writing as we go along, but it's of course important to delete the temporary code when you're done using it for testing.

Okay, on to Test Case 4. You've probably noticed that we're adding small bits of functionality as we go along, using our test cases to guide us to what we should do next. There's actually a development methodology called Test-Driven Development (TDD) that follows the same approach. One of the core ideas in TDD is that we start with a set of test cases that all fail (because we haven't implemented anything yet), then we focus our implementation on getting each of our test cases to pass. That's essentially what we're doing here.

**Test Case 4****Clicking Quit Button on High Score Menu**

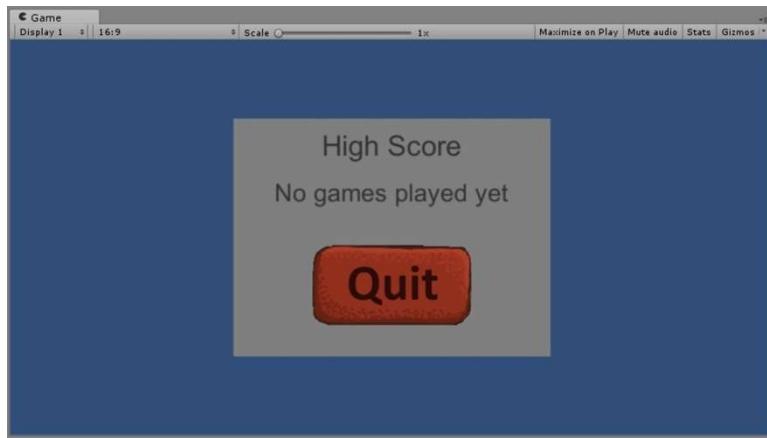
Step 1. Click High Score button on Main Menu. Expected Result: Move to High Score Menu

Step 2. Click Quit button on High Score Menu. Expected Result: Move to Main Menu

Step 3. Click Quit button on Main Menu. Expected Result: Exit game

Test Case 4 checks that the Quit button on the High Score Menu works properly, but of course it doesn't because we haven't added it yet. Drag the HighScoreMenu prefab from the Project window into the Hierarchy window so we can modify the prefab. Add a QuitMenuItem Image to the HighScoreMenu canvas and add a Button component to the image. Set up the button to highlight/unhighlight properly.

Here's what our final High Score Menu looks like.



**Figure 20.3. Final High Score Menu**

Add the following method to the `HighScoreMenu` script:

```
/// <summary>
/// Handles the on click event from the quit button
/// </summary>
public void HandleQuitButtonOnClickEvent()
{
    MenuManager.GoToMenu(MenuName.Main);
}
```

The good news is that going to the Main Menu is always the right thing to do whether we got to the High Score Menu by clicking the High Score button on the Main Menu or we got to it by finishing a game.

Add the `HandleQuitButtonOnClickEvent` method as a listener for the `On Click ()` event in the `QuitMenuItem` on the High Score Menu, then click the Prefab Apply button near the top of the Inspector to apply your changes to the prefab. Delete the `HighScoreMenu` canvas from the Hierarchy window.

Next, we need to modify our `MenuManager GoToMenu` method to handle the `MenuName.Main` case:

```

case MenuName.Main:
    // go to MainMenu scene
    SceneManager.LoadScene("MainMenu");
    break;

```

Execute Test Case 4, which should work fine.

You might be wondering why the High Score Menu disappears when we load the MainMenu scene, especially since we instantiated the HighScoreMenu prefab and added it to the scene when the player clicked the High Score menu button on the Main Menu. This happens because when we load a scene using the `SceneManager.LoadScene` method, it loads the scene as it was built in the editor. Because we don't have an instance of the HighScoreMenu prefab in the MainMenu scene in the editor, it's not included when that scene is loaded.

## **Test Case 5**

### **Clicking Play Button on Main Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click X in corner of player. Expected Result: Exit game

To pass Test Case 5, we need to add a Play button to the Main Menu that moves the game to the Difficulty Menu when it's clicked. The Difficulty Menu is a "regular" menu, not a popup menu like the High Score Menu or the Pause Menu, so we'll build a separate scene for the Difficulty Menu.

Here's what our final Main Menu looks like.



**Figure 20.4. Final Main Menu**

Right click the scenes folder in the Project window and select Create > Scene. Rename the new scene DifficultyMenu. We'll add the difficulty buttons to this menu after we're done implementing the Test Case 5 functionality.

Add a sprite for the play menu button to the `sprites\menus` folder (our sprite is named `playmenubutton`) and use the Sprite Editor to slice it into two sprites like we did for the previous menu button sprites. Add an Image to the scene (make sure you're working in the `MainMenu` scene, not the `DifficultyMenu` scene) and rename the Image to `PlayMenubutton`. Set the Source Image to the `playmenubutton_0` sprite. Add a

Button component and set it up to change to the highlighted sprite as appropriate. Run the game to confirm that highlighting works properly.

Add the following method to the `MainMenu` script

```
/// <summary>
/// Handles the on click event from the play button
/// </summary>
public void HandlePlayButtonOnClickEvent()
{
    MenuManager.GoToMenu(MenuName.Difficulty);
}
```

and add the new method as a listener for the `On Click ()` event in the `PlayMenuButton`.

Implement the `MenuName.Difficulty` case in the `MenuManager` `GoToMenu` method as follows:

```
case MenuName.Difficulty:
    // go to DifficultyMenu scene
    SceneManager.LoadScene("DifficultyMenu");
    break;
```

We now have a second scene in our game, so before running it, select File > Build Settings... from the top menu bar. Double-click the `DifficultyMenu` scene in the scenes folder in the Project window, then click the Add Open Scenes button in the build settings popup. Click the Build And Run button near the bottom right of the build settings popup and execute Test Case 5. Although it looks like the Main Menu buttons are just disappearing when we click the Play button, we're actually moving to the `DifficultyMenu`, so Test Case 5 passes.

You can actually execute all the test cases that don't use the Quit button on the Main Menu in the editor instead of building the project each time. That's really your choice, though we did want to show you how to include multiple scenes in the build. We always Build And Run each of our test cases at least once to make sure they work in "the real world", but feel free to just run the game in the editor as you go along if that's what you prefer to do.

## Test Case 6

### **Clicking Easy Button on Difficulty Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to `Difficulty Menu`

Step 2. Click Easy button on `Difficulty Menu`. Expected Result: Move to gameplay screen for easy game

Step 3. Click X in corner of player. Expected Result: Exit game

## Test Case 7

### **Clicking Medium Button on Difficulty Menu**

Step 1. Click Play button on Main Menu. Expected Result: Move to `Difficulty Menu`

Step 2. Click Medium button on `Difficulty Menu`. Expected Result: Move to gameplay screen for medium game

Step 3. Click X in corner of player. Expected Result: Exit game

## Test Case 8

### Clicking Hard Button on Difficulty Menu

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Hard button on Difficulty Menu. Expected Result: Move to gameplay screen for hard game

Step 3. Click X in corner of player. Expected Result: Exit game

Test Cases 6, 7, and 8 have us clicking the 3 difficulty buttons on the Difficulty Menu to start a game with the selected difficulty. Let's start by implementing the Easy button.

Open the DifficultyMenu scene and add an Easy menu button Image (called EasyMenuItem) to the scene. Set up the Image as we've been doing for our previous menu buttons. Rename the canvas containing the Easy menu button to DifficultyMenuCanvas. Run the game to confirm that highlighting works properly on the Easy menu button.

Our typical next step would be to add a `DifficultyMenu` script that listens for and handles a click on the Easy menu button. Before we do that, though, let's think about what should happen. We know we're going to want to move to a Gameplay scene, so right click the scenes folder in the Project window and select Create > Scene to create that new scene. Moving to that new scene will be easy to implement using the `SceneManager` class, but how to do we handle the fact that it should be an easy (rather than medium or hard) game that starts up there?

This is actually a harder problem to solve than it seems. We'll end up having a `FeedTheTeddies` script attached to the Main Camera in the Gameplay scene (as we've regularly done in the past), but we don't really want our `DifficultyMenu` script to have to know about the `FeedTheTeddies` script. That's just good Object-Oriented design, but even if we were willing to implement the functionality that way, the `FeedTheTeddies` script (really, the Main Camera in the Gameplay scene) isn't active when a difficulty menu button is pressed.

Luckily, we've solved a similar problem before with an `EventManager` class; we'll need an `EventManager` class to support scoring and the game timer during gameplay, but we can use it here as well. Our `EventManager` implementation is similar to what we implemented previously because we'll sometimes need to add listeners for events before the event invokers have been created. In our current situation, the `DifficultyMenu` script (the event invoker) won't become active until the player navigates to the Difficulty Menu, but we'll want to already have a listener added for that event so the listener hears the event when the `DifficultyMenu` script invokes it.

We'll start by adding an enum for the event names the `EventManager` script will handle (you'll see why this is helpful soon). The list of enum values is incomplete at this point, but we'll simply add more as we need them:

```
/// <summary>
/// The names of the events in the game
/// </summary>
public enum EventName
{
    GameStartedEvent
}
```

We put this enum into a new scripts\events folder in the Project window; we'll put all our event-related scripts into this folder as well.

When we implement our `EventManager` class (coming soon, we promise!), we're going to need to use an `IntEventInvoker` class that extends the `MonoBehaviour` class by adding a `UnityEvent<int>` field and an `AddListener` method. Why do we use a `UnityEvent<int>` field? By using `UnityEvent<int>` as the data type for our field, we can save any of the events we define as child classes of `UnityEvent<int>` in that field. Good thing we learned about inheritance, huh?

Here's the code for the `IntEventInvoker` class:

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

/// <summary>
/// Extends MonoBehaviour to support invoking a
/// one integer argument UnityEvent
/// </summary>
public class IntEventInvoker : MonoBehaviour
{
    protected UnityEvent<int> unityEvent;

    /// <summary>
    /// Adds the given listener for the UnityEvent
    /// </summary>
    /// <param name="listener">listener</param>
    public void AddListener(UnityAction<int> listener)
    {
        unityEvent.AddListener(listener);
    }
}
```

We make the field `protected` so child classes can set it to the appropriate class and invoke the event as appropriate.

We're actually lucky that in this game the only custom events we're going to invoke using the `EventManager` are one integer argument events. If we had lots of different custom events with different numbers and types of arguments, our `EventManager` and (multiple) `EventInvoker` classes would be much more complicated. Not a problem here, though (whew!).

Here's our `EventManager` class (with embedded comments for explanation):

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

We're using lots of namespaces here!

```
/// <summary>
/// Manages connections between event listeners and event invokers
/// </summary>
public static class EventManager
```

```
{
    #region Fields

    static Dictionary<EventName, List<IntEventInvoker>> invokers =
        new Dictionary<EventName, List<IntEventInvoker>>();
```

We're using a collection from the `System.Collections.Generic` namespace that we haven't used before: a `Dictionary`. The C# documentation tells us that a `Dictionary` "Represents a collection of keys and values." That should sound really familiar to you, because we used `PlayerPrefs` to store key/value pairs. The idea works exactly the same, because we store and retrieve values using a particular key.

The `Dictionary` class actually gives us much more flexibility than `PlayerPrefs` does. Recall that with `PlayerPrefs`, the key we use is always a `string`, and the only data types we can save for the value are `float`, `int`, or `string`. With the `Dictionary` class, the key and value can be any data type we want them to be. In the line of code above, our `Dictionary` uses `EventName` as the key (so we can use an `EventName` like `EventName.GameStartedEvent` to look up a value in the `Dictionary`) and `List<IntEventInvoker>` as the value (so we can save and retrieve a list of the `IntEventInvoker`s that invoke the given event name). We'll see how to use the `invokers` field once we start looking at the `EventManager` methods.

As you can see, we needed to define both the `EventName` enum and the `IntEventInvoker` class to implement the line of code above.

```
static Dictionary<EventName, List<UnityAction<int>>> listeners =
    new Dictionary<EventName, List<UnityAction<int>>>();
```

Our `listeners` field lets us look up a list of listeners for a particular event using an `EventName` as the key. Each of the listeners is a `UnityAction<int>` because that's the delegate type we need to use to listen for a `UnityEvent<int>`.

```
#endregion

#region Public methods

/// <summary>
/// Initializes the event manager
/// </summary>
public static void Initialize()
{
    // create empty lists for all the dictionary entries
    foreach (EventName name in Enum.GetValues(typeof(EventName)))
    {
        if (!invokers.ContainsKey(name))
        {
            invokers.Add(name, new List<IntEventInvoker>());
            listeners.Add(name, new List<UnityAction<int>>());
        }
        else
        {
            invokers[name].Clear();
            listeners[name].Clear();
        }
    }
}
```

}

The foreach loop above ensures we have empty lists as the dictionary entries in the `invokers` and `listeners` fields for each of the `EventName` values (we used foreach loops to process all the values of an enum in Section 10.5 when we used nested loops to fill a deck of cards). The value for each of those `EventName` keys is simply an empty list of the appropriate type. We know (looking ahead) that this method may get called multiple times while the game runs, and trying to add a key that already exists in a `Dictionary` throws an exception. We avoid that exception using the if statement above. If the `name` key isn't currently in the `invokers` field, the if body creates a new (empty list) dictionary entry in the `invokers` and `listeners` fields; otherwise, the else body simply clears (e.g., empties) the existing lists for those entries.

We include this method for efficiency. When it's time to add an invoker or a listener, we won't have to use an if statement to make sure we have a list for that `EventName` key in the dictionary already before adding the new invoker/listener.

```
/// <summary>
/// Adds the given invoker for the given event name
/// </summary>
/// <param name="eventName">event name</param>
/// <param name="invoker">invoker</param>
public static void AddInvoker(EventName eventName,
    IntEventInvoker invoker)
{
    // add listeners to new invoker and add new invoker to dictionary
    foreach (UnityAction<int> listener in listeners[eventName])
    {
        invoker.AddListener(listener);
    }
    invokers[eventName].Add(invoker);
}
```

When a consumer of the `EventManager` class calls the `AddInvoker` method to add an invoker for a particular `EventName`, there may already be listeners that were added to the `listeners` field to listen for that `EventName`. The foreach loop in the code above adds each of those listeners as a listener to the new invoker being added. The last line of code above adds the new invoker to the list of invokers for the given `EventName` in the `invokers` field.

```
/// <summary>
/// Adds the given listener for the given event name
/// </summary>
/// <param name="eventName">event name</param>
/// <param name="listener">listener</param>
public static void AddListener(EventName eventName,
    UnityAction<int> listener)
{
    // add as listener to all invokers and add new listener to dictionary
    foreach (IntEventInvoker invoker in invokers[eventName])
    {
        invoker.AddListener(listener);
```

```

        }
        listeners[eventName].Add(listener);
    }

#endregion
}

```

This method is similar to the `AddInvoker` method. When a consumer of the `EventManager` class calls the `AddListener` method to add a listener for a particular `EventName`, there may already be invokers that were added to the `invokers` field to invoke that `EventName`. The foreach loop in the code above adds the new listener being added as a listener to each of those invokers. The last line of code above adds the new listener to the list of listeners for the given `EventName` in the `listeners` field.

Okay, we said we included the `Initialize` method above for efficiency, but at this point we haven't called that method from anywhere. Let's add a new `GameInitializer` script that does that and attach the script to the Main Camera in the MainMenu scene (since we know that's the scene we start in when we run the game):

```

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

/// <summary>
/// Initializes the game
/// </summary>
public class GameInitializer : MonoBehaviour
{
    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
    {
        EventManager.Initialize();
    }
}

```

Boy, it feels like we've been doing a ton of work without getting to click the Easy button on the Difficulty Menu! Don't worry, we are getting closer. Even more importantly, we've been building some important infrastructure into our game to support the event system we need both for some of the menus and for gameplay.

Next, we add a `GameStartedEvent` class to the `scripts\events` folder; this is the event that the `DifficultyMenu` will invoke when one of the difficulty menu buttons is clicked:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

/// <summary>
/// An event that indicates a game should be started with
/// the given difficulty
/// </summary>

```

```
public class GameStartedEvent : UnityEvent<int>
{
}
```

For readability, we also include an enum for the game difficulty in a new scripts\gameplay folder:

```
/// <summary>
/// The different difficulties in the game
/// </summary>
public enum Difficulty
{
    Easy,
    Medium,
    Hard
}
```

Let's take a look at the new `DifficultyMenu` script we add to the scripts\menus folder:

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

/// <summary>
/// Listens for the OnClick events for the difficulty menu buttons
/// </summary>
public class DifficultyMenu : IntEventInvoker
{
```

`DifficultyMenu` is a child class of `IntEventInvoker` because we need it to add itself as an invoker in the `EventManager`.

```
/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // add event component and add invoker to event manager
    unityEvent = new GameStartedEvent();
    EventManager.AddInvoker(EventName.GameStartedEvent, this);
}
```

The `Start` method sets the `unityEvent` field (inherited from the `IntEventInvoker` class) to a new `GameStartedEvent` object, then adds itself (using `this`) to the `EventManager` as an invoker of the `EventName.GameStartedEvent`.

```
/// <summary>
/// Handles the on click event from the easy button
/// </summary>
public void HandleEasyButtonOnClickEvent()
{
    unityEvent.Invoke((int)Difficulty.Easy);
}
```

Notice that when we invoke the event, we cast `Difficulty.Easy` to an `int`. We can do this because the default underlying data type for an `enum` is `int`.

Attach the `DifficultyMenu` script to the `DifficultyMenuCanvas` in the `DifficultyMenu` scene. Add the `HandleEasyButtonOnClickEvent` method as a listener for the `On Click ()` event in the `EasyMenuButton` on the `Difficulty Menu`.

Okay, now we have the `DifficultyMenu` invoking the event when the Easy button is clicked, but at this point no one is actually listening for that event. Because there are a variety of gameplay characteristics that depend on difficulty, we'll write a static `DifficultyUtils` class in the `scripts\gameplay` folder. For now, this class will listen for the event above to set the gameplay difficulty and start the game; by the time we're done, this class will also expose methods to provide the difficulty-dependent gameplay values to consumers of the class. Here's the code:

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

/// <summary>
/// Provides difficulty-specific utilities
/// </summary>
public static class DifficultyUtils
{
    #region Fields

    static Difficulty difficulty;

    #endregion
}
```

The `difficulty` field stores the gameplay difficulty so we can provide difficulty-specific values to the consumers of the class (through methods we'll write later) during gameplay.

```
#region Public methods

/// <summary>
/// Initializes the difficulty utils
/// </summary>
public static void Initialize()
{
    EventManager.AddListener(EventName.GameStartedEvent,
        HandleGameStartedEvent);
}

#endregion
```

The `Initialize` method adds the `HandleGameStartedEvent` method (discussed next) to the `EventManager` as a listener for the `EventName.GameStartedEvent`.

```
#region Private methods
```

```

/// <summary>
/// Sets the difficulty and starts the game
/// </summary>
/// <param name="intDifficulty">int value for difficulty</param>
static void HandleGameStartedEvent(int intDifficulty)
{
    difficulty = (Difficulty)intDifficulty;
    SceneManager.LoadScene("Gameplay");
}

#endregion
}

```

Because the default underlying data type for an `enum` is `int`, we can cast the `intDifficulty` parameter as a `Difficulty` to save it into the `difficulty` field. The second line of code above moves the game to the `Gameplay` scene.

We also need to add the following line of code at the end of the `GameInitializer` `Start` method:

```
DifficultyUtils.Initialize();
```

This line of code has to come after the call to the `EventManager` `Initialize` method because the `DifficultyUtils` `Initialize` method uses the `EventManager`.

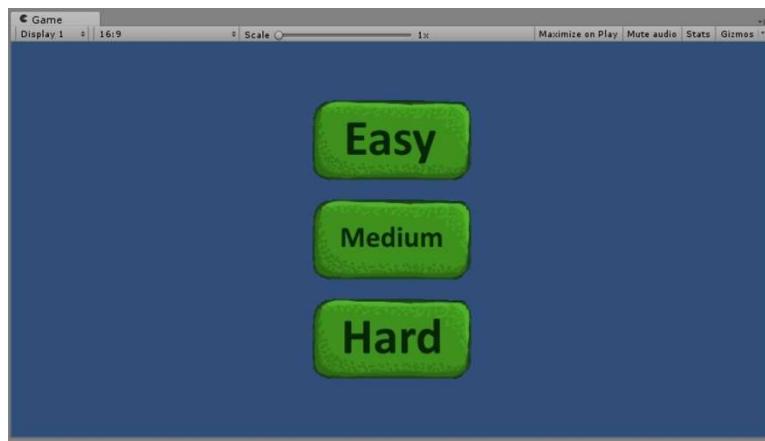
We're almost there! Right click the scenes folder in the Project window and add a new `Gameplay` scene; open the new scene. Select `File > Build Settings...` from the top menu bar and add the new `Gameplay` scene to the build. Click the `Build And Run` button near the bottom right and execute Test Case 6.

Well, we're definitely getting to the `Gameplay` scene when we click the `Easy` button on the `Difficulty` Menu, but how do we know that the game difficulty is being set to `Easy`?

This will be easiest to check running the game in the editor. In Visual Studio, set a breakpoint in the `DifficultyUtils` `HandleGameStartedEvent` method on the line of code that loads the `Gameplay` scene. Select `Debug > Attach Unity Debugger` and attach to the `Unity Instance` for the project. In the editor, run the game and execute Test Case 6. When the game stops at the breakpoint, look at the value of the `difficulty` field in Visual Studio to confirm that the difficulty is set to `Easy`.

That's it for Test Case 6! Add the `Medium` and `Hard` buttons to the `DifficultyMenuCanvas`, add `HandleMediumButtonOnClickEvent` and `HandleHardButtonOnClickEvent` methods to the `DifficultyMenu` script to handle clicks on those buttons, use the editor to add those methods as listeners for `On Click ()` on `MediumMenuButton` and `HardMenuButton`, and execute Test Cases 7 and 8. Use the debugger to confirm the difficulty is being set correctly in `DifficultyUtils`.

Here's our final `Difficulty` Menu:



**Figure 20.5. Final Difficulty Menu**

The last menu we need to add to our game is the Pause Menu, another popup menu. Start by opening the Gameplay scene. We'll build a prefab for the pause menu like we did for the high score menu. Add a new canvas to the scene and name the new canvas PauseMenu. Add an Image to the canvas, center the image in the scene, change the name of the image to Background, set the Width to 400, set the Height to 300, and make the Color an opaque gray.

### **Test Case 9**

#### **Clicking Resume Button on Pause Menu**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu
- Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game
- Step 3. Press Escape key. Expected Result: Game paused and Pause Menu displayed on top of game
- Step 4. Press Resume button on Pause menu. Expected Result: Pause Menu removed and game unpause
- Step 5. Click X in corner of player. Expected Result: Exit game

Let's work toward passing Test Case 9, where we click the Resume button on the Pause Menu to resume a paused game. Add a Resume button to the PauseMenu canvas and set it up like our other menu buttons.

You should probably realize that we're going to write a `PauseMenu` script and attach it to the PauseMenu canvas. Before we do that, though, we need to think about what should happen when the player clicks the Resume button. There are really only two things that should happen here: the game should be unpause (game objects start moving again, timers start running again, and so on) and the Pause Menu should be removed from the scene. Removing the Pause Menu from the scene is appropriately done from within the `PauseMenu` script, but who controls when the game is paused and unpause?

If we think about how Test Case 9 works, in Step 3 we press the Escape key to pause the game. We mentioned earlier that we knew we'd have a `FeedTheTeddies` script attached to the Main Camera in the Gameplay scene, and it's appropriate for us to detect the Escape key being pressed to instantiate the PauseMenu prefab in the scene within that script. We could pause the game in that script before we instantiate the prefab, or we can have the `PauseMenu` script pause the game when it's instantiated. Which approach is better?

In our opinion, having the `PauseMenu` script pause the game when it's instantiated and unpause the game when either menu button is clicked is the better approach, because that contains all the pause/unpause functionality in a single script. This also makes our implementation a bit easier, because if we had the `FeedTheTeddies` script unpause the game when the Resume button is clicked, the best approach would be to have the `PauseMenu` script invoke an event (a `GameResumedEvent`, say) that the `FeedTheTeddies` script listens for. We certainly understand how to use our event system properly by this point, but that event would be added complexity that we don't really need.

Here's the `PauseMenu` script we add to the `scripts\menus` folder:

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

/// <summary>
/// Pauses and unpauses the game. Listens for the OnClick
/// events for the pause menu buttons
/// </summary>
public class PauseMenu : MonoBehaviour
{
    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
    {
        // pause the game when added to the scene
        Time.timeScale = 0;
    }

    /// <summary>
    /// Handles the on click event from the Resume button
    /// </summary>
    public void HandleResumeButtonOnClickEvent()
    {
        // unpause game and destroy menu
        Time.timeScale = 1;
        Destroy(gameObject);
    }
}
```

Attach the `PauseMenu` script to the `PauseMenu` canvas and use the editor to add the `HandleResumeButtonOnClickEvent` method as a listener for `On Click ()` on the `ResumeMenuButton`. Drag the `PauseMenu` canvas onto the `prefabs\Resources` folder in the Project window to create the prefab and delete the `PauseMenu` canvas from the Gameplay scene.

All we need now is the `FeedTheTeddies` script to detect the Escape key being pressed to instantiate the `PauseMenu` prefab in the scene. Here's the script to add to the `scripts\gameplay` folder:

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

```

/// <summary>
/// Game manager
/// </summary>
public class FeedTheTeddies : MonoBehaviour
{
    /// <summary>
    /// Update is called once per frame
    /// </summary>
    void Update()
    {
        // check for pausing game
        if (Input.GetKeyDown("escape"))
        {
            MenuManager.GoToMenu(MenuName.Pause);
        }
    }
}

```

We originally tried to use the `Input.GetAxis` method with a new Pause Input axis to detect that the Escape key had been pressed, but we just ended up infinitely instantiating the PauseMenu prefab as though the Escape key was interpreted as being pressed on every `Update` (the `.GetAxisRaw` method didn't work either). Everything worked fine if we had a breakpoint set on the line that calls the `MenuManager GoToMenu` method and continued on from that breakpoint each time, but it didn't work running normally. That's why we're using the `Input.GetKeyDown` method instead.

Speaking of the `MenuManager GoToMenu` method, we needed to implement the `MenuName.Pause` case in that method:

```

case MenuName.Pause:
    // instantiate prefab
    Object.Instantiate(Resources.Load("PauseMenu"));
    break;

```

Attach the `FeedTheTeddies` script attached to the Main Camera in the Gameplay scene and execute Test Case 9 starting from the MainMenu scene. Everything should look like it's working fine, but because there's nothing moving in the Gameplay scene we can't really tell if the game is getting paused and unpause correctly for this test case.

In Visual Studio, set breakpoints in the `PauseMenu` script on both lines of code that change `Time.timeScale`. Select Debug > Attach Unity Debugger and attach to the Unity Instance for the project. In the editor, run the game and execute Test Case 9. When the game stops at the first breakpoint, press F10 to Step Over the line of code in Visual Studio to make sure that line of code doesn't crash. Press F5 in Visual Studio, then click the Resume button in the Game view of the editor. When the game stops at the second breakpoint, press F10 to Step Over the line of code in Visual Studio to make sure that line of code doesn't crash. Select Run > Stop from the top menu bar in Visual Studio, then stop the game in the editor. Everything should work fine, so Test Case 9 passes.

## Test Case 10

### Clicking Quit Button on Pause Menu

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Press Escape key. Expected Result: Game paused and Pause Menu displayed on top of game

Step 4. Press Quit button on Pause menu. Expected Result: Move to Main Menu

Step 5. Click Quit button on Main Menu. Expected Result: Exit game

Okay, only one more menu test case left. In Test Case 10, we click the Quit button on the Pause Menu to quit the game and return to the Main Menu.

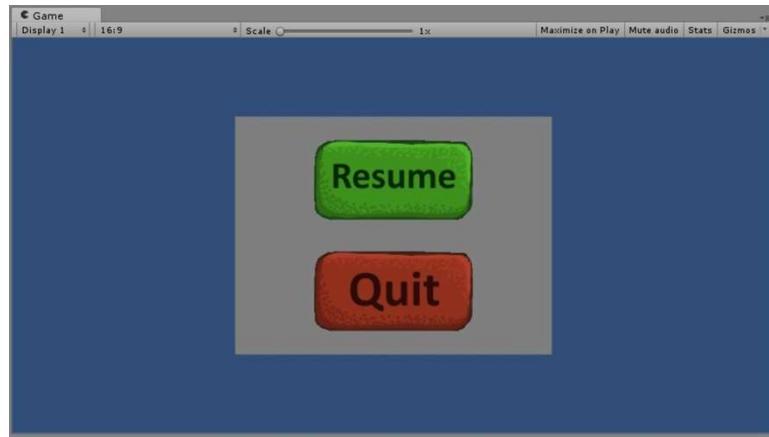
Add the following method to the `PauseMenu` script:

```
/// <summary>
/// Handles the on click event from the Quit button
/// </summary>
public void HandleQuitButtonOnMouseEvent()
{
    // unpause game, destroy menu, and go to main menu
    Time.timeScale = 1;
    Destroy(gameObject);
    MenuManager.GoToMenu(MenuName.Main);
}
```

You might be wondering why we bother unpausing the game when we're quitting the game anyway. We do this so that if we return to the Main Menu, then play another game, the new game doesn't start as paused. That's another good reason to encapsulate all the pause/unpause functionality in the `PauseMenu` script.

Drag the `PauseMenu` prefab from the Project window into the Hierarchy window. Add the Quit button to the `PauseMenu` canvas and add the `HandleQuitButtonOnMouseEvent` method as a listener for On Click () on the new Quit button. Click the Prefab Apply button on the `PauseMenu` canvas to apply the changes to the `PauseMenu` prefab and delete the `PauseMenu` from the Gameplay scene.

Here's the final Pause Menu:



**Figure 20.6. Final Pause Menu**

Execute Test Case 10, using the debugger to confirm that the code that sets `Time.timeScale` in the `HandleQuitButtonOnMouseEvent` method doesn't crash. Test Case 10 also passes.

We're finally done implementing our menu system! Huzzah.

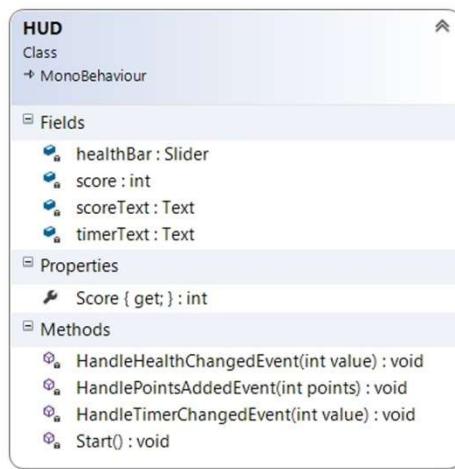
## 20.5. Test the Code (Menus)

We've been running our menu test cases throughout the previous section, confirming that they all work correctly. If you haven't run each test case in the player yet, though, now would be the time to do that. Now that we've added all the scenes in our game to the build, you can just use File > Build & Run to run the game in the player.

## 20.6. Design a Solution (Basic Gameplay: Stupid Teddies)

Let's think about how the game objects will interact in our game to implement basic gameplay. We're going to design, implement, and test gameplay in a number of steps, because solving our problems iteratively is even more important as our problems get larger. We'll define basic gameplay as the burger moving around and shooting french fries, teddy bears being periodically spawned and (stupidly) shooting teddy bear projectiles, and displaying score, health, and the game timer in a HUD.

The key game objects in our game are the burger avatar for the player, the french fries that the burger shoots, the teddy bears that are periodically spawned, and the teddy bear projectiles the teddy bears shoot. We should include the HUD as a game object as well, since it will also interact with other game objects. The UML for the new `HUD` script is shown below.

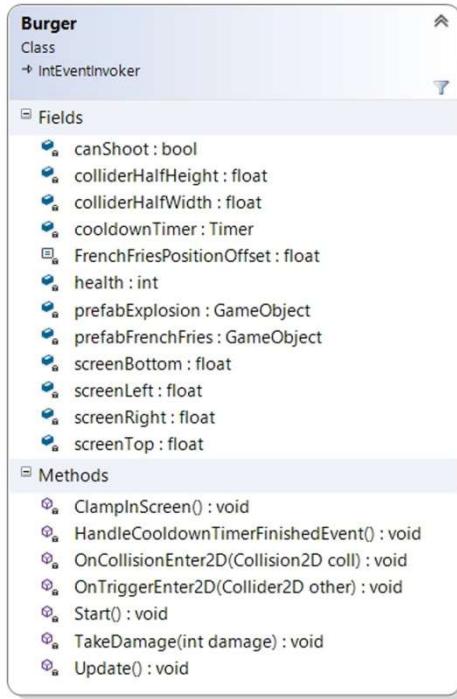


**Figure 20.7. HUD Script UML**

So, what are the object interactions for the burger? There are actually four: the burger shooting (instantiating) french fries, the burger colliding with french fries, the burger colliding with a teddy bear, and the burger colliding with a teddy bear projectile.

The burger shooting french fries really doesn't require further detail, though we'll of course have to implement the shooting functionality in a `Burger` script. If the burger collides with french fries, we'll simply destroy the french fries. If the player is bad enough to run into their own french fries, those french fries don't get to go kill something else in the game, but we're nice enough not to damage the burger when that happens. We'll implement this functionality in the `Burger` script as well.

What should happen when the burger collides with a teddy bear or a teddy bear projectile? The burger will take damage from the collision, with the burger's health indicated by a health bar in the HUD. The teddy bear or teddy bear projectile will be destroyed as a result of the collision. Some of this processing will occur in the `Burger` script, while updating the health bar will happen in a `HUD` script. Because the `Burger` script shouldn't know about the `HUD` script, we'll use the event system to indicate that the burger's health has changed. The UML for the `Burger` script is shown below.



**Figure 20.8. Burger Script UML**

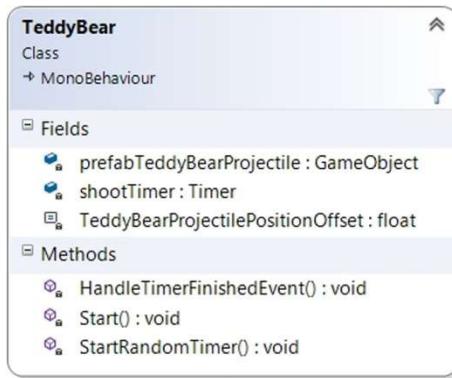
French fries will collide with teddy bears, other french fries, or teddy bear projectiles. If the french fries collide with other french fries or teddy bear projectiles, both participants in the collision will be destroyed. This punishes the player if they're foolish enough to shoot their own french fries, but also gives them access to a strategy where they shoot the teddy bear projectiles with their french fries to protect their burger. This processing will happen in a `FrenchFries` script.

When french fries collide with a teddy bear, both participants in the collision will be destroyed. This collision is worth points for the player, so the score in the HUD will be increased as well. Some of this processing will happen in the `FrenchFries` script, but because the `FrenchFries` script shouldn't know about the `HUD` script, we'll use the event system to indicate that points should be added to the score. Here's the UML for the `FrenchFries` script:



**Figure 20.9. FrenchFries Script UML**

Teddy bears collide with the burger, other teddy bears, teddy bear projectiles, or french fries. Collisions between a teddy bear and a burger and a teddy bear and french fries have already been discussed above. When a teddy bear collides with another teddy bear, they'll bounce off each other using the Unity physics system. When a teddy bear collides with a teddy bear projectile, the teddy bear projectile will be destroyed; this processing will happen in a `TeddyBear` script. The UML for the `TeddyBear` script is shown below.



**Figure 20.10. TeddyBear Script UML**

Teddy bear projectiles collide with the burger, teddy bears, french fries, and other teddy bear projectiles. The first three collisions have already been discussed above. When a teddy bear projectile collides with another teddy bear projectile, both participants in the collision will be destroyed. This processing will happen in a `TeddyBearProjectile` script. The UML for the `TeddyBearProjectile` script is shown below. Although this looks almost identical to the UML for the `FrenchFries` script, we'll find a number of important differences between them when we Write the Code.



**Figure 20.11. TeddyBearProjectile Script UML**

We'll undoubtedly include other components in our solution when we Write the Code, but this is a good analysis of how the objects in our game will interact.

## 20.7. Write Test Cases (Basic Gameplay: Stupid Teddies)

Here are the test cases for the basic gameplay.

### Test Case 11

#### **Watch Game Timer Count Down**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu
- Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game
- Step 3. Wait for game timer to reach 0. Expected Result: High Score Menu displayed above Gameplay scene
- Step 4. Click Quit button on High Score Menu. Expected Result: Move to Main Menu
- Step 5. Click X in corner of player. Expected Result: Exit game

### Test Case 12

#### **Moving Burger Around Screen**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu
- Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game
- Step 3. Move burger around screen using arrow keys. Expected Result: Burger moves around screen, staying in the borders of the screen.
- Step 4. Click X in corner of player. Expected Result: Exit game

### Test Case 13

#### **Watch Teddy Bears Spawning and Moving Around Screen**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu
- Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game
- Step 3. Watch teddy bears spawning and moving around screen. Expected Result: Teddy bears move around screen, staying in the borders of the screen. Teddy bears bounce off each other on collision
- Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 14****Collide Burger with Teddy Bear**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game  
 Step 3. Move burger to collide with teddy bear. Expected Result: Teddy bear explodes. Health bar shows reduced health  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 15****Shoot French Fries**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game  
 Step 3. Shoot french fries using the space bar. Expected Result: French fries move straight up from burger when shot. Firing rate controlled when space bar held down  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 16****Collide Burger with French Fries**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game  
 Step 3. Move burger to collide with french fries. Expected Result: French fries explode. Health bar doesn't change  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 17****Collide French Fries with Teddy Bear**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game  
 Step 3. Shoot french fries into collision with teddy bear. Expected Result: French fries and teddy bear explode. Score increases  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 18****Watch Teddy Bears Shoot Teddy Bear Projectiles**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game  
 Step 3. Watch teddy bears periodically shoot teddy bear projectiles. Expected Result: Teddy bear projectiles move straight down from teddy bear when shot  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 19****Collide Burger with Teddy Bear Projectile**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game  
 Step 3. Move burger to collide with teddy bear projectile. Expected Result: Teddy bear projectile explodes. Health bar shows reduced health  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 20****Damage Burger until Health is 0**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game  
 Step 3. Move burger to collide with teddy bears and teddy bear projectiles until health is 0. Expected Result: High Score Menu displayed above Gameplay scene. All moving objects are paused  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 21****Collide French Fries with Teddy Bear Projectile**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game  
 Step 3. Shoot french fries into collision with teddy bear projectile. Expected Result: French fries and teddy bear projectile explode. No change in score  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 22****Watch Teddy Bear Collide with Teddy Bear Projectile**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Hard button on Difficulty Menu. Expected Result: Move to gameplay screen for hard game  
 Step 3. Watch until a teddy bear collides with a teddy bear projectile. Expected Result: Teddy bear projectile explodes  
 Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 23****Watch Teddy Bear Projectile Collide with Teddy Bear Projectile**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu  
 Step 2. Click Hard button on Difficulty Menu. Expected Result: Move to gameplay screen for hard game  
 Step 3. Watch until a teddy bear projectile collides with a teddy bear projectile. Expected Result: Both teddy bear projectiles explode  
 Step 4. Click X in corner of player. Expected Result: Exit game

The test cases above include testing all the interactions we identified in the Design a Solution step except for french fries colliding with french fries. If you think about this, because all french fries move at the same speed in the same direction (up), there's actually no way to make one french fries collide with another one. We therefore won't implement or test that interaction. We also included several test cases (11 and 20) that capture the requirement that the High Score Menu is displayed when the game is over.

## 20.8. Write the Code and Test the Code (Basic Gameplay: Stupid Teddies)

Before moving on, remove the effects of gravity from the game world by selecting Edit > Project Settings >Physics 2D and setting the Y component of Gravity to 0.

To pass Test Case 11, we'll need to implement a HUD containing the timer text, a game timer, and two timer-related events: a `TimerChangedEvent` and a `TimerFinishedEvent`. This work is very similar to what we did in Section 19.4., so we don't duplicate that discussion here. The most significant changes we made were to support using our new event system. Look at the code accompanying this chapter for the details if you'd like.

As we added the game timer support to our `FeedTheTeddies` script (which we changed to an `IntEventInvoker`, by the way), we encountered our first constant, which determines how long the game lasts. We know that before we're done with our game we'll be using XML configuration data, so we decided to ease our transition to that approach by implementing a `ConfigurationUtils` class to return all the configurable data for our game. At this point in our implementation, the properties in that class will simply return hard-coded values, but by the time we're done it will return values from the XML file. Each time we get ready to declare a constant in our code, we'll think about whether it should be a tunable value (and therefore in `ConfigurationUtils`) or whether it should be a non-tunable constant hard-coded into the game.

After doing the work discussed above, our test case works fine until it crashes in the following code in the `MenuManager` `GoToMenu` method:

```
case MenuName.HighScore:

    // deactivate MainMenuCanvas and instantiate prefab
    GameObject mainMenuCanvas = GameObject.Find("MainMenuCanvas");
    mainMenuCanvas.SetActive(false);
    Object.Instantiate(Resources.Load("HighScoreMenu"));
    break;
```

Recall that we wrote this code when we were going to the High Score Menu from the Main Menu, so we deactivated the main menu canvas before instantiating the HighScoreMenu prefab. When we go to the High Score Menu from the Gameplay scene, there is no main menu canvas to deactivate. That means the `GameObject Find` method returns null and we get a `NullReferenceException` when we try to call the `SetActive` method on that (null) object. There's an easy fix, of course, to make the code work in both scenarios:

```
case MenuName.HighScore:

    // deactivate MainMenuCanvas and instantiate prefab
    GameObject mainMenuCanvas = GameObject.Find("MainMenuCanvas");
    if (mainMenuCanvas != null)
    {
        mainMenuCanvas.SetActive(false);
    }
    Object.Instantiate(Resources.Load("HighScoreMenu"));
    break;
```

Although Test Case 11 now passes, we want to point out that the location of the timer text on the screen doesn't look right when we run at full screen or don't pick exactly the resolution we've been testing at when using Build Settings ... to run the game in the player. Of course, anyone you distribute your game to will be using the player (ignoring a web deployment), so we need to fix this problem.

This is an easy fix. Select the timer text element in the Hierarchy window and click the gray box above Anchors in the Rect Transform component in the Inspector. Set the Anchor to the top right anchor preset and you're good to go.

Well, maybe not quite yet. Although setting the anchor properly locates the text element correctly, what about its size? Shouldn't the text be larger at higher resolutions so it takes up the same amount of "screen real estate"?

The answer is yes, of course. This is also an easy fix. Select the HUD canvas in the Hierarchy window and set the UI Scale Mode in the Canvas Scaler component to Scale with Screen Size. The only default value you should really change is Match, which you should set to 0.5. Basically, that makes sure scaling works properly across different aspect ratios.

If you execute Test Case 11 with a variety of resolutions, you'll see that the timer text is now consistently placed in a reasonable location and is the appropriate size as well. In fact, before moving on you should go make these changes to all the menu canvases in the game (including prefabs) to make sure the menus also scale properly based on the selected resolution.

### **Test Case 12**

#### **Moving Burger Around Screen**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

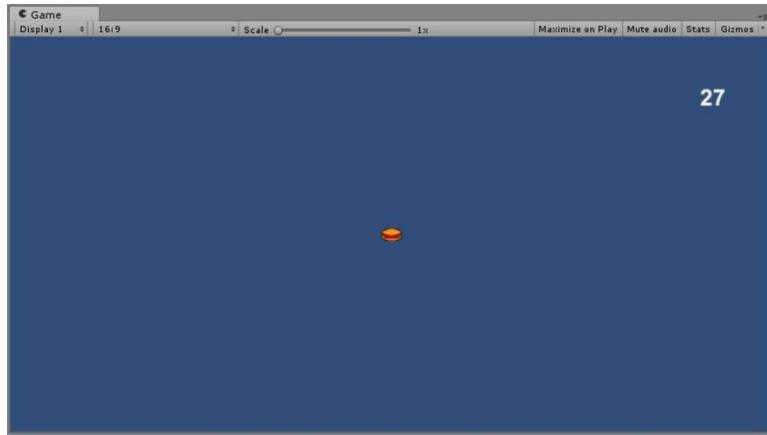
Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Move burger around screen using arrow keys. Expected Result: Burger moves around screen, staying in the borders of the screen.

Step 4. Click X in corner of player. Expected Result: Exit game

This is a lot like the work we did to set up a fish we could move around the screen in previous games, so we won't go over the details again here. After you've added the burger sprite, added it to the scene, added a Box Collider 2D, and implemented and attached a new `Burger` script to the game object to handle movement, execute Test Case 12.

Your game should look something like the figure below at this point.



**Figure 20.12. Burger and Timer**

### **Test Case 13**

#### **Watch Teddy Bears Spawning and Moving Around Screen**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Watch teddy bears spawning and moving around screen. Expected Result: Teddy bears move around screen, staying in the borders of the screen. Teddy bears bounce off each other on collision

Step 4. Click X in corner of player. Expected Result: Exit game

Before we start working on our `TeddyBear` and `TeddyBearSpawner` scripts, we want to point out a flaw in the way we've used edge colliders to keep game objects on the screen up to this point. You may have noticed that everything seems to work fine in the Game view, but if you run the game in the player the game objects may not stay on the screen as they should (sometimes bouncing too soon, sometimes leaving the screen before bouncing) depending on the resolution the player selected. That means that anyone playing our game can mess up the gameplay just by changing the player resolution! This is obviously a bad thing, so let's solve that problem here.

We add a new `ScreenUtils` script to add the edge colliders on the four sides of the screen at runtime and attach the script to the Main Camera in the Gameplay scene. Here's the code we have at the beginning of the `Start` method:

```
// cache reusable values for efficiency
PhysicsMaterial2D screenEdgeMaterial = Resources.Load("ScreenEdgeMaterial")
    as PhysicsMaterial2D;
float zLocation = -Camera.main.transform.position.z;

// add left edge collider
EdgeCollider2D collider = gameObject.AddComponent<EdgeCollider2D>();
Vector3[] screenEndPoints = new Vector3[2];
Vector3[] worldEndPoints = new Vector3[2];
Vector2[] colliderEndPoints = new Vector2[2];
screenEndPoints[0].x = 0;
screenEndPoints[0].y = Screen.height;
screenEndPoints[0].z = zLocation;
worldEndPoints[0] = Camera.main.ScreenToWorldPoint(screenEndPoints[0]);
colliderEndPoints[0].x = worldEndPoints[0].x;
colliderEndPoints[0].y = worldEndPoints[0].y;
screenEndPoints[1].x = screenEndPoints[0].x;
screenEndPoints[1].y = 0;
screenEndPoints[1].z = zLocation;
worldEndPoints[1] = Camera.main.ScreenToWorldPoint(screenEndPoints[1]);
colliderEndPoints[1].x = worldEndPoints[1].x;
colliderEndPoints[1].y = worldEndPoints[1].y;
collider.points = colliderEndPoints;
collider.sharedMaterial = screenEdgeMaterial;
```

We figure out each end point for the collider in screen coordinates, then convert to world coordinates, then set the `Vector2` components of the collider end point appropriately. At the end of the block of code, we set the end points of the actual collider and set the Physics 2D material for the collider as well. Adding the right, top, and bottom colliders is similar.

Our `TeddyBear` and `TeddyBearSpawner` scripts and our `TeddyBear` prefab are almost identical to those we used in the fish game in the previous chapter; the only difference is that we use some properties in the `ConfigurationUtils` class instead of constants in the scripts. The `ConfigurationUtils` properties sometimes use new properties in the `DifficultyUtils` class to return the appropriate difficulty-specific values. For example, because teddy bears spawn faster at higher difficulties, the minimum spawn delay is no longer constant. Here's the new `MinSpawnDelay` property in the `DifficultyUtils` class:

```
/// <summary>
/// Gets the min spawn delay for teddy bear spawning
/// </summary>
```

```

/// <value>minimum spawn delay</value>
public static float MinSpawnDelay
{
    get
    {
        switch (difficulty)
        {
            case Difficulty.Easy:
                return ConfigurationUtils.EasyMinSpawnDelay;
            case Difficulty.Medium:
                return ConfigurationUtils.MediumMinSpawnDelay;
            case Difficulty.Hard:
                return ConfigurationUtils.HardMinSpawnDelay;
            default:
                return ConfigurationUtils.EasyMinSpawnDelay;
        }
    }
}

```

It might seem a little awkward to you that the property above needs to access the `EasyMinSpawnDelay`, `MediumMinSpawnDelay`, or `HardMinSpawnDelay` property in the `ConfigurationUtils` class, but we do it that way so that the `ConfigurationUtils` class is the only access point for the configuration data from the XML file. Here's the new `MinSpawnDelay` property in the `ConfigurationUtils` class:

```

/// <summary>
/// Gets the min spawn delay for teddy bear spawning
/// </summary>
/// <value>minimum spawn delay</value>
public static float MinSpawnDelay
{
    get { return DifficultyUtils.MinSpawnDelay; }
}

```

See the code accompanying the chapter to look at the changes in detail as you see fit.

Test Case 13 now passes, but there is one gameplay tweak we want to make. At this point there's no limit to how many bears can appear on the screen. Allowing unlimited teddies is of course one reasonable game design decision, but we prefer having a difficulty-specific limit instead. We added the appropriate properties to `ConfigurationUtils` and `DifficultyUtils` to support that, we tagged the `TeddyBear` prefab with a new `TeddyBear` tag, and we made sure the spawner doesn't spawn a new bear if the scene already contains the max number of bears for the game difficulty.

## Test Case 14

### **Collide Burger with Teddy Bear**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Move burger to collide with teddy bear. Expected Result: Teddy bear explodes. Health bar shows reduced health

Step 4. Click X in corner of player. Expected Result: Exit game

We can of course get to Step 3 in Test Case 14 and run the burger into a teddy bear, but neither one of the expected results occurs. Let's blow up the teddy bear first, then work on reducing the burger's health.

Start by building an Explosion prefab like we did in Section 8.2. We can detect the collision in either the `Burger` script or the `TeddyBear` script. Because the collision affects the burger's health, let's detect it in the `Burger` script. We've of course detected collisions between fish and teddy bears before, so the following method should look familiar to you:

```
/// <summary>
/// Processes collisions with other game objects
/// </summary>
/// <param name="coll">collision info</param>
void OnCollisionEnter2D(Collision2D coll)
{
    // if colliding with teddy bear, destroy teddy and reduce health
    if (coll.gameObject.CompareTag("TeddyBear"))
    {
        Instantiate(prefabExplosion,
                    coll.gameObject.transform.position, Quaternion.identity);
        Destroy(coll.gameObject);
    }
}
```

Run the game now and you should be able to explode teddy bears by running them over with your burger (come on, how many times do you get to say a sentence like that?).

The other expected result we have is that the health bar should show reduced health. Of course, it's hard for that to happen here because we don't even have a health bar yet! Let's get to work on that now.

Luckily, Unity provides a Slider UI component that will work great for this. Right click the HUD canvas in the Gameplay scene and select UI > Slider. Use the Anchor Presets and the Y location in the Rect Transform component to center the slider at the top center of the game screen, aligned vertically with the timer text. Change the name of the slider to HealthBar.

Sliders can actually be interactable, like when you have the player use a slider to adjust music volume in a game, but that's not how we're using the slider here. Expand the HealthBar in the Hierarchy window, right click Handle Slide Area, and select Delete. In the Slider component in the Inspector, uncheck the Interactable check box. Change Transition just below the Interactable check box to None, because that transition setting is for when the player is interacting with the component.

Next, change both the Max Value and Value to 100, since health will go from 0 to 100 and the player starts with 100 health. You should also check the Whole Numbers check box because our player health will be an integer.

Clearly, the `Burger` script (which detects the collision with the teddy bear) shouldn't know anything about the HUD, so we need to add a new `HealthChangedEvent` and link the burger as an invoker and the HUD as a listener for that event through the `EventManager`. First, we add a new `HealthChangedEvent` value to the `EventName` enumeration. On the `Burger` side, we change `Burger` to be a child class of `IntEventInvoker`, add a `health` field

```
// health support
int health = 100;
```

set the `unityEvent` field to a new `HealthChangedEvent` object and add the script as an event invoker in the `Start` method

```
// add as event invoker
unityEvent = new HealthChangedEvent();
EventManager.AddInvoker(EventName.HealthChangedEvent, this);
```

and add two more lines to the if body for a collision with a teddy bear (using a new `ConfigurationUtils` property as well)

```
health = Mathf.Max(0, health - ConfigurationUtils.BearDamage);
unityEvent.Invoke(health);
```

The first line of code subtracts the damage the bear inflicts from the current health value, then sets health to whichever is higher, that value or 0. Basically, the code adjusts the health properly without letting it go below 0. The second line of code invokes the event with the new health value.

On the `HUD` side, we add a new field to hold a reference to the health bar slider

```
// health support
[SerializeField]
Slider healthBar;
```

write a new method to change the value of the health bar when the health changes

```
/// <summary>
/// Handles the health changed event by changing
/// the health bar value
/// </summary>
/// <param name="value">health value</param>
void HandleHealthChangedEvent(int value)
{
    healthBar.value = value;
}
```

and add the new method as a listener in the `Start` method

```
// add listener for HealthChangedEvent
EventManager.AddListener(EventName.HealthChangedEvent,
    HandleHealthChangedEvent);
```

Select the `HUD` canvas in the Hierarchy window, then drag the `HealthBar` onto the `Health Bar` field of the `HUD` script in the Inspector.

If you execute Test Case 14 again, you'll see that we now get both expected results. Be sure to execute Test Case 14 starting at the Main Menu scene to make sure the `Burger` `Start` method doesn't crash trying to add itself as an invoker to an uninitialized `EventManager`.



**Figure 20.13. Health Bar (with some damage)**

### Test Case 15

#### Shoot French Fries

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Shoot french fries using the space bar. Expected Result: French fries move straight up from burger when shot. Firing rate controlled when space bar held down

Step 4. Click X in corner of player. Expected Result: Exit game

If we're going to have the burger shoot french fries, we're going to want a french fries prefab we can instantiate as the burger shoots. Add a french fries sprite to the sprites\gameplay folder in the Project window, drag the sprite into the Hierarchy window, and rename the game object in the Hierarchy window FrenchFries. Add a new FrenchFries tag to the tags in the game and add that tag to the FrenchFries game object. We're going to apply an impulse force to get the french fries moving when we shoot them, so add a Rigidbody2D component, freeze rotation in Z, and set the Interpolate field to Interpolate. We're also going to need to detect collisions between the french fries and other game objects, so add a Box Collider 2D component as well. We don't actually want the french fries to bounce off the edge colliders surrounding the screen or the other game objects, so check the Is Trigger checkbox.

As we indicated in the Design a Solution step, we're going to be using a `FrenchFries` script for some of our game functionality, so we'll add that script now. At this point, the script will just get the french fries moving when they're instantiated, but we'll add more functionality as we go along.

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

/// <summary>
/// French fries
/// </summary>
public class FrenchFries : MonoBehaviour
{
    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
```

```

    {
        // apply impulse force to get projectile moving
        GetComponent<Rigidbody2D>().AddForce(
            new Vector2(0, ConfigurationUtils.FrenchFriesImpulseForce),
            ForceMode2D.Impulse);
    }
}

```

Drag the `FrenchFries` script onto the FrenchFries game object in the Hierarchy window, drag the FrenchFries game object from the Hierarchy window onto the prefabs folder in the Project window, and delete the FrenchFries game object from the Hierarchy window.

Now that we have the prefab built we can move over to the `Burger` script to add the french fries shooting functionality. We need to change the Positive Button in the Fire1 input axis to space (instead of the default left ctrl), add a `prefabFrenchFries` field to the `Burger` script, mark that field with `[SerializeField]`, and populate that field in the Inspector with the FrenchFries prefab. Then we add the following code at the end of the `Burger` Update method:

```

// shoot french fries
if (Input.GetAxisRaw("Fire1") != 0)
{
    Instantiate(prefabFrenchFries, transform.position,
                Quaternion.identity);
}

```

Go ahead and run the test case. The first part of Step 3 works, with french fries shooting straight up from the burger, but the firing rate is definitely NOT being controlled!

We'll control the firing rate by using a cooldown timer. When the player presses the space bar, we'll instantiate a french fries and start the cooldown timer. While the cooldown timer is running, we won't instantiate any more french fries. Once the cooldown timer finishes, we'll enable shooting again.

That approach gives the player an "automatic burger" with a particular firing rate, but we can also support a different player tactic where the player repeatedly presses and releases the space bar as quickly as they can (spam, spam, spam, spam, ...). This is more work for the player, but they'll be able to achieve a faster firing rate shooting that way.

Almost of our code will be in the `Burger` script. We'll keep track of whether or not the player can shoot using a Boolean flag, and we'll also need a countdown timer as a field:

```

// firing support
bool canShoot = true;
Timer countdownTimer;

```

We write a new listener for the `TimerFinishedEvent`

```

/// <summary>
/// Reenables shooting when the cooldown timer finishes
/// </summary>
void HandleCooldownTimerFinished()
{
    canShoot = true;
}

```

```
}
```

and create the countdown timer and add the listener at the end of the `Start` method:

```
// set up cooldown timer
cooldownTimer = gameObject.AddComponent<Timer>();
cooldownTimer.Duration = ConfigurationUtils.BurgerCooldownSeconds;
cooldownTimer.AddTimerFinishedEventListner(
    HandleCooldownTimerFinishedEvent);
```

Before we finish off the `Burger` code, we need to add a `Stop` method to our `Timer` class. That way, when the player releases the space bar we can immediately stop the timer and reenable shooting. Here's the `Timer` `Stop` method:

```
/// <summary>
/// Stops the timer
/// </summary>
public void Stop()
{
    started = false;
    running = false;
}
```

We need to set both the `started` and the `running` flags to `false` to ensure the `Timer` `Finished` property still works properly.

Finally, we change the shooting code at the end of the `Burger` `Update` method:

```
// reenable shooting on Fire1 axis release
if (!canShoot &&
    Input.GetAxisRaw("Fire1") == 0)
{
    cooldownTimer.Stop();
    canShoot = true;
}
```

The block of code above stops the cooldown timer and immediately reenables shooting.

```
// shoot french fries
if (canShoot &&
    Input.GetAxisRaw("Fire1") != 0)
{
    cooldownTimer.Run();
    canShoot = false;
    Instantiate(prefabFrenchFries, transform.position, Quaternion.identity);
}
```

We added another condition to the Boolean expression for our if statement so we only shoot french fries if the `canShoot` flag is `true`. At the start of the if body, we start the cooldown timer and set `canShoot` to `false` to disable shooting until the cooldown timer finishes or the player releases the space bar.

The test case now passes fine, but we actually have a huge inefficiency in our game. You may not have realized it yet, but every french fries we fire stays in the game world forever, even after it leaves the

screen (you can actually see this in the Hierarchy window as you fire french fries, because none of them are removed from the Hierarchy window as you play). This is really bad because we waste CPU time updating all those projectiles even though they're really out of the game. We can solve this problem by adding the following method to the `FrenchFries` script:

```
// Called when the french fries become invisible
void OnBecameInvisible()
{
    // destroy the game object
    Destroy(gameObject);
}
```

Unity automatically calls the `OnBecameInvisible` method when the game object the script is attached to can no longer be seen by the camera, so this is exactly what we need. Execute Test Case 15 again to verify that the `FrenchFries` objects are destroyed when they leave the screen.

## Test Case 16

### **Collide Burger with French Fries**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Move burger to collide with french fries. Expected Result: French fries explode. Health bar doesn't change

Step 4. Click X in corner of player. Expected Result: Exit game

In our default tuning configuration, the burger will be slower than the french fries, so we'll never actually be able to collide the burger with the french fries. Because we're going to include the configuration data in an XML file, someone could change the settings so the burger is faster than the french fries, making this collision possible. We'll implement and test this test case by changing the `ConfigurationUtils.FrenchFriesImpulseForce` to slow down the french fries to support that collision, then change that value back to what we think is more reasonable for actual gameplay.

We said we'd do this work in the `Burger` script ... but we have a problem! We want to handle the processing in the inherited `MonoBehaviour.OnTriggerEnter2D` method, which is called when "another object enters a trigger collider attached to this object". Although the `Burger` game object has a Box Collider 2D attached to it, that collider is not marked as a trigger because we use that collider to keep the burger on the screen. The `FrenchFries` game object does have a trigger collider attached to it, though, so we could add the required processing to the `FrenchFries` script instead, but looking ahead, that's probably not the correct solution.

Why not? Because when we get to detecting collisions between the `Burger` and a teddy bear projectile, it will be much cleaner if the `Burger` script handle reducing the player's health rather than having the `TeddyBearProjectile` script do that. Let's go with our original design decision to have the `Burger` handle this processing. Add another Box Collider 2D component to the `Burger` game object, but check the Is Trigger checkbox for the new collider. Now add the following method to the `Burger` script:

```
/// <summary>
/// Processes trigger collisions with other game objects
/// </summary>
/// <param name="other">information about the other collider</param>
void OnTriggerEnter2D(Collider2D other)
```

```

{
    // if colliding with french fries, destroy french fries
    if (other.gameObject.CompareTag("FrenchFries"))
    {
        Instantiate(prefabExplosion,
                    other.gameObject.transform.position, Quaternion.identity);
        Destroy(other.gameObject);
    }
}

```

If you try to execute Test Case 16 now, you'll see an explosion on top of the burger every time you fire. We have a problem that we didn't notice before because the french fries move so fast. Recall that in our [Burger](#) Update code, we set the new french fries location to

```
transform.position
```

which is the center of the burger. Unfortunately, that means that the french fries are immediately colliding with the burger, so they're immediately destroyed. The best way to solve this problem is to actually instantiate the french fries slightly above the burger when they're fired so they don't actually start out in collision with the burger. We changed our french fries instantiation code to

```
Vector3 frenchFriesPos = transform.position;
frenchFriesPos.y += FrenchFriesPositionOffset;
Instantiate(prefabFrenchFries, frenchFriesPos, Quaternion.identity);
```

where `FrenchFriesPositionOffset` is a new constant we added to the [Burger](#) script.

After executing Test Case 16 to confirm that it passes, change the `ConfigurationUtils.FrenchFriesImpulseForce` to a more reasonable value.

## Test Case 17

### Collide French Fries with Teddy Bear

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Shoot french fries into collision with teddy bear. Expected Result: French fries and teddy bear explode. Score increases

Step 4. Click X in corner of player. Expected Result: Exit game

As we mentioned above, some of this processing will happen in the [FrenchFries](#) script, but we'll use the event system to indicate that points should be added to the score. We'll start by creating a new `PointsAddedEvent`, adding [FrenchFries](#) as an invoker for that event, and adding the [HUD](#) as a listener for that event. We also add score text to the [HUD](#) to display the current score. This is almost identical to work we've done in previous chapters, so refer to the code accompanying the chapter for the details.

Next, we add a new field to the [FrenchFries](#) script, mark that field with `[SerializeField]`, and populate that field in the Inspector with the Explosion prefab. Then we add the following method to the [FrenchFries](#) script:

```

/// <summary>
/// Processes trigger collisions with other game objects
/// </summary>

```

```

/// <param name="other">information about the other collider</param>
void OnTriggerEnter2D(Collider2D other)
{
    // if colliding with teddy bear, add score and destroy teddy bear and self
    if (other.gameObject.CompareTag("TeddyBear"))
    {
        unityEvent.Invoke(ConfigurationUtils.BearPoints);
        Instantiate(prefabExplosion,
                    other.gameObject.transform.position, Quaternion.identity);
        Destroy(other.gameObject);
        Instantiate(prefabExplosion,
                    transform.position, Quaternion.identity);
        Destroy(gameObject);
    }
}

```

We instantiate two explosions because we're thinking of both the teddy bear and the french fries as being stuffed with C4!

We've actually added another potential inefficiency with the changes we just made for this new functionality, though we won't see that inefficiency in our implementation. Remember, the `EventManager` holds a dictionary of the invokers for each event, including the `PointsAddedEvent`. When the `EventManager` `AddListener` method is called, that method adds the listener to each of the invokers for the event. Once a particular instance of a french fries game object leaves the scene, we destroy that game object, but it stays in the `EventManager` as an invoker for the `PointsAddedEvent`. We don't see that inefficiency here because the `HUD` script is the only listener for this event and it adds itself before any invokers are added, but it's in general risky to make assumptions about the order in which invokers and listeners will be added.

To fix this problem, we need to add a new method to the `EventManager`:

```

/// <summary>
/// Removes the given invoker for the given event name
/// </summary>
/// <param name="eventName">event name</param>
/// <param name="invoker">invoker</param>
public static void RemoveInvoker(EventName eventName,
    IntEventInvoker invoker)
{
    // remove invoker from dictionary
    invokers[eventName].Remove(invoker);
}

```

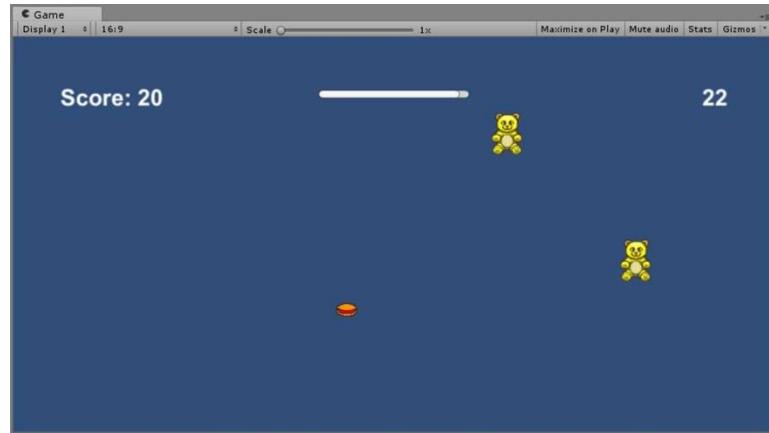
Now we can change the `FrenchFries` `OnBecameInvisible` method to

```

// Called when the french fries become invisible
void OnBecameInvisible()
{
    // destroy the game object
    EventManager.RemoveInvoker(EventName.PointsAddedEvent, this);
    Destroy(gameObject);
}

```

and we've removed that inefficiency. Execute Test Case 17 to see that it passes.



**Figure 20.14. Score Included**

### **Test Case 18**

#### **Watch Teddy Bears Shoot Teddy Bear Projectiles**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Watch teddy bears periodically shoot teddy bear projectiles. Expected Result: Teddy bear projectiles move straight down from teddy bear when shot

Step 4. Click X in corner of player. Expected Result: Exit game

It's time for the teddy bears to fight back. We start by adding a `TeddyBearProjectile` prefab that moves straight down when it's added to the scene. Here's the `TeddyBearProjectile` script:

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

/// <summary>
/// A teddy bear projectile
/// </summary>
public class TeddyBearProjectile : MonoBehaviour
{
    /// <summary>
    /// Use this for initialization
    /// </summary>
    void Start()
    {
        // apply impulse force to get projectile moving
        GetComponent<Rigidbody2D>().AddForce(
            new Vector2(0,
                -ConfigurationUtils.TeddyBearProjectileImpulseForce),
            ForceMode2D.Impulse);
    }

    /// <summary>
    /// Called when the teddy bear projectile becomes invisible
    /// </summary>
    void OnBecameInvisible()
```

```
        {
            // destroy the game object
            Destroy(gameObject);
        }
    }
```

Note that the Y component of the `Vector2` we use to apply the impulse force is negative so the force pushes the teddy bear projectile down. We of course also destroy teddy bear projectiles when they leave the game just like we did for french fries for the same efficiency reason.

Next, we add a `Timer` field to the `TeddyBear` script so we can have the teddy bear periodically shoot teddy bear projectiles, and we also add a field to hold a `TeddyBearProjectile` prefab we instantiate when it's time for the teddy bear to shoot one.

```
// shooting support  
[SerializeField]  
GameObject prefabTeddyBearProjectile;  
Timer shootTimer;
```

We add a new `HandleTimerFinishedEvent` method as well to shoot a teddy bear projectile and restart the timer with a random duration:

```
/// <summary>
/// Shoots a teddy bear projectile, resets the timer
/// duration, and restarts the timer
/// </summary>
void HandleTimerFinishedEvent()
{
    // shoot a teddy bear projectile
    Vector3 projectilePos = transform.position;
    projectilePos.y -= TeddyBearProjectilePositionOffset;
    Instantiate(prefabTeddyBearProjectile, projectilePos,
        Quaternion.identity);

    // change timer duration and restart
    shootTimer.Duration = Random.Range(ConfigurationUtils.BearMinShotDelay,
        ConfigurationUtils.BearMaxShotDelay);
    shootTimer.Run();
}
```

We realized that we needed to offset the teddy bear projectile from the teddy bear because (later) we'd have the same problem we had shooting french fries from the center of the burger.

Finally, we add the timer initialization code at the end of the `Start` method:

```
// create and start timer
shootTimer = gameObject.AddComponent<Timer>();
shootTimer.AddTimerFinishedEventListenner(HandleTimerFinishedEvent);
shootTimer.Duration = Random.Range(ConfigurationUtils.BearMinShotDelay,
    ConfigurationUtils.BearMaxShotDelay);
shootTimer.Run();
```

We actually realized at this point that the last two lines of code in the `HandleTimerFinishedEvent` method and in the block of code above are identical. Because we hate duplicated code, we pulled those lines out into a separate `StartRandomTimer` method and called that method from the two places in the code above. We also refactored similar code in the `TeddyBearSpawner` script the same way.

Because the teddy bear firing rate is difficulty-dependent, we added our usual properties to the `ConfigurationUtils` and `DifficultyUtils` classes to support that.

Execute Test Case 18 and you'll see the teddy bears starting to fight back.

## Test Case 19

### **Collide Burger with Teddy Bear Projectile**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Move burger to collide with teddy bear projectile. Expected Result: Teddy bear projectile explodes. Health bar shows reduced health

Step 4. Click X in corner of player. Expected Result: Exit game

Of course, at this point the teddy bears' attempts to fight back are futile since the teddy bear projectiles don't damage the burger. We can fix that by adding an else if clause to our `Burger` `OnTriggerEnter2D` method:

```
else if (other.gameObject.CompareTag("TeddyBearProjectile"))
{
    // if colliding with teddy bear projectile, destroy projectile and
    // reduce health
    Instantiate(prefabExplosion,
        other.gameObject.transform.position, Quaternion.identity);
    Destroy(other.gameObject);
    health = Mathf.Max(0, health - ConfigurationUtils.BearProjectileDamage);
    unityEvent.Invoke(health);
}
```

Getting Test Case 19 to pass was easy!

## Test Case 20

### **Damage Burger until Health is 0**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Move burger to collide with teddy bears and teddy bear projectiles until health is 0. Expected Result: High Score Menu displayed above Gameplay scene. All moving objects are paused

Step 4. Click X in corner of player. Expected Result: Exit game

If you try to execute Test Case 20 now, you'll see that the game keeps going even when the burger health reaches 0. Before we add the code to go to the High Score Menu in that case, let's do a little refactoring. At this point, we inflict burger damage in two places: in the `OnCollisionEnter2D` method on a collision with a teddy bear and in the `OnTriggerEnter2D` method on a collision with a teddy bear projectile. Let's pull that code out into a new `TakeDamage` method that we call from both those places. We'll also check if the game is over in that new method.

What should we do if the game is over? The easiest thing (so, obviously, the wrong thing!) to do would be to have the method call the `MenuManager GoToMenu` method to move to the High Score Menu. But the `Burger` shouldn't really know about the `MenuManager` class, so this isn't the best choice. As usual, we'll use the event system for this, with a new `GameOverEvent` that the `Burger` script invokes in the `TakeDamage` method if health is 0 and the `FeedTheTeddies` script listens for. The `FeedTheTeddies` script calls the `MenuManager GoToMenu` method when the `GameOverEvent` is invoked, which is fine because this script manages the overall game and already calls that method to go to the pause menu as appropriate.

There are couple of interesting details to point out here. The `GameOverEvent` has to be a child class of `UnityEvent<int>` even though we don't really need to pass an `int` when we invoke this event. We need to do it this way because the parameter for the listener in the `EventManager AddListener` method is a `UnityAction<int>`, so we can only add listeners that listen for `UnityEvent<int>` events. It would certainly be possible to make a more robust `EventManager` that allows different versions of `UnityEvent`, but because almost all of our events are `UnityEvent<int>`, we might as well save ourselves the complexity and just live with a parameter in the listener method that we don't actually use.

The other thing to point out is that we need to declare a new `GameOverEvent` field in the `Burger` class even though it's an `IntEventInvoker` containing a `unityEvent` field. That's because we're already using the `unityEvent` field for the `HealthChangedEvent`, so we can't use it for the `GameOverEvent` as well.

Although that all made sense to us when we implemented it, it didn't actually work in practice. When we ran the code after making the changes described above, the High Score Menu popped up the first time the burger took damage. Here's why.

Even though our dictionaries in the `EventManager` have different keys for each `EventName`, we lose that distinction when we call the `IntEventInvoker AddListener` method from within the `EventManager AddListener` method. When the `FeedTheTeddies` script calls the `EventManager AddListener` method for the `GameOverEvent`, the listener actually gets added for the `Burger unityEvent` field, which as we mentioned above is for the `HealthChangedEvent`. Everything worked fine for `IntEventInvokers` that only invoked a single event, but now our stupid `Burger` script needs to invoke two events. Ugh!

Although we could probably come up with a really ugly workaround for this, we have to imagine that we'd run into this situation a lot for more complex games with more events. Let's actually implement a robust, general solution to this problem. We start by changing our `IntEventInvoker` class:

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

/// <summary>
/// Extends MonoBehaviour to support invoking
/// one integer argument UnityEvents
/// </summary>
public class IntEventInvoker : MonoBehaviour
{
```

```

protected Dictionary<EventName, UnityEvent<int>> unityEvents =
    new Dictionary<EventName, UnityEvent<int>>();

/// <summary>
/// Adds the given listener for the given event name
/// </summary>
/// <param name="eventName">event name</param>
/// <param name="listener">listener</param>
public void AddListener(EventName eventName, UnityAction<int> listener)
{
    // only add listeners for supported events
    if (unityEvents.ContainsKey(eventName))
    {
        unityEvents[eventName].AddListener(listener);
    }
}
}

```

Instead of having a single `unityEvent` field, the `IntEventInvoker` class now has a dictionary of events keyed by the event name. That lets each instance of the `IntEventInvoker` class invoke a number of different events.

Of course, the change above breaks tons of our existing code! The easiest compilation error to fix is in the `EventManager` `AddInvoker` and `AddListener` methods, where we simply add the event name as an argument when we call the `IntEventInvoker` `AddListener` method. The other two compilation errors occur in two different places: whenever we invoke an event and whenever we create the event object. Let's fix those problems in the `FrenchFries` script as an example.

The `FrenchFries` script currently invokes the `PointsAddedEvent` in the `OnTriggerEnter2D` method using

```
unityEvent.Invoke(ConfigurationUtils.BearPoints);
```

This doesn't work any more, because we've changed our field (now called `unityEvents`) to a dictionary. Here's the revised code:

```
unityEvents[EventName.PointsAddedEvent].Invoke(
    ConfigurationUtils.BearPoints);
```

We're using the event name to key into our `unityEvents` dictionary to invoke the required event. We're definitely "glass half full" kinds of authors, so we'll observe that this might actually make our event invocation code more readable than it was before because now it explicitly says what event is being invoked.

We also need to fix where we create our event object, which is currently implemented in the `FrenchFries` `Start` method as

```
unityEvent = new PointsAddedEvent();
```

This is also a straightforward fix, because all we need to do is add the new event object to the dictionary with the appropriate key:

```
unityEvents.Add(EventName.PointsAddedEvent, new PointsAddedEvent());
```

Okay, go fix all the compilation errors. It always hurts to break a bunch of our code to make our design more robust, but this was definitely the right thing for us to do here.

When we execute Test Case 20 now, it almost works. Our expected results say that all moving objects are paused when the High Score Menu is displayed, but that doesn't happen because we haven't paused the game when we display the High Score Menu. Also, the High Score Menu says that no games have been played yet, though we just finished a game. We realize at this point that we need to add an expected result to our test case:

## Test Case 20

### **Damage Burger until Health is 0**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Move burger to collide with teddy bears and teddy bear projectiles until health is 0. Expected Result: High Score Menu displayed above Gameplay scene with a high score displayed (either from this game or a previous game with a higher score). All moving objects are paused

Step 4. Click X in corner of player. Expected Result: Exit game

We already implemented pause and unpause functionality in our `PauseMenu` script, so we'll use the same ideas to pause the game when the High Score Menu is added to the scene and to unpause the game when the player clicks the Quit button.

Now let's fix the incorrect message on the High Score Menu. To do that, we need to add a new field in the `FeedTheTeddies` script for the HUD and populate that field in the Inspector. We also need to write a new `SetHighScore` method in that script and call it from both the `HandleGameTimerFinishedEvent` and the `HandleGameOverEvent` methods before going to the High Score Menu:

```
/// <summary>
/// Sets the saved high score if we have a new or first high score
/// </summary>
void SetHighScore()
{
    HUD hudScript = hud.GetComponent<HUD>();
    int currentScore = hudScript.Score;
    if (PlayerPrefs.HasKey("High Score"))
    {
        if (currentScore > PlayerPrefs.GetInt("High Score"))
        {
            PlayerPrefs.SetInt("High Score", currentScore);
            PlayerPrefs.Save();
        }
    }
    else
    {
        PlayerPrefs.SetInt("High Score", currentScore);
        PlayerPrefs.Save();
    }
}
```

Of course, when we did this we realized that the bodies of the `HandleGameTimerFinishedEvent` and the `HandleGameOverEvent` methods are identical. We created a new `EndGame` method that sets the high score and goes to the High Score Menu and just call the `EndGame` method from both those methods. We actually thought about replacing those methods with the `EndGame` method, but we couldn't because the listener for the `TimerFinishedEvent` from the game timer doesn't have any parameters and the listener for the `GameOverEvent` has a single `int` parameter.

Test Case 20 now passes. Because we made changes to the `HighScoreMenu` script, it also makes sense at this point to run Test Cases 3 and 4 again to make sure the High Score Menu still works properly when we go to it from the Main Menu.

## **Test Case 21**

### **Collide French Fries with Teddy Bear Projectile**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game

Step 3. Shoot french fries into collision with teddy bear projectile. Expected Result: French fries and teddy bear projectile explode. No change in score

Step 4. Click X in corner of player. Expected Result: Exit game

To implement this functionality, we add the following `else if` clause to the `FrenchFries` `OnTriggerEnter2D` method:

```
else if (other.gameObject.CompareTag("TeddyBearProjectile"))
{
    // if colliding with teddy bear projectile, destroy projectile and self
    Instantiate(prefabExplosion, other.gameObject.transform.position,
               Quaternion.identity);
    Destroy(other.gameObject);
    Instantiate(prefabExplosion, transform.position, Quaternion.identity);
    Destroy(gameObject);
}
```

Test Case 21 passes with the above code included, though we admit we had to slow down the teddy bear projectiles so we could hit them with french fries!

## **Test Case 22**

### **Watch Teddy Bear Collide with Teddy Bear Projectile**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Hard button on Difficulty Menu. Expected Result: Move to gameplay screen for hard game

Step 3. Watch until a teddy bear collides with a teddy bear projectile. Expected Result: Teddy bear projectile explodes

Step 4. Click X in corner of player. Expected Result: Exit game

We said in the Design a Solution step that "When a teddy bear collides with a teddy bear projectile, the teddy bear projectile will be destroyed; this processing will happen in a `TeddyBear` script." We're not actually going to do it that way because the `TeddyBearProjectile` prefab has a trigger collider but the `TeddyBear` prefab doesn't. Adding an additional collider to the `TeddyBear` prefab would be a bad choice because it would increase the number of collision checks the Unity engine has to do each frame by a multiple of the number of teddy bears in the scene. Instead, we'll add a `prefabExplosion` field and an `OnTriggerEnter2D` method to the `TeddyBearProjectile` script to do the required processing:

```

/// <summary>
/// Processes trigger collisions with other game objects
/// </summary>
/// <param name="other">information about the other collider</param>
void OnTriggerEnter2D(Collider2D other)
{
    // if colliding with teddy bear, destroy self
    if (other.gameObject.CompareTag("TeddyBear"))
    {
        Instantiate(prefabExplosion, transform.position,
                    Quaternion.identity);
        Destroy(gameObject);
    }
}

```

It's not at all unusual to make changes to the design as we work through our implementation steps. Nobody ever gets the design perfect the first time, so don't feel badly that we didn't.

Before executing Test Case 22, we made the Burger a prefab and removed it from the Gameplay scene. That way, the only exploding collisions we'd see would be teddy bears colliding with teddy bear projectiles. After making the `TeddyBearProjectilePositionOffset` value in the `TeddyBear` script a little larger (teddy bear projectiles were exploding when they were fired) the test case passes fine.

### Test Case 23

#### **Watch Teddy Bear Projectile Collide with Teddy Bear Projectile**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Hard button on Difficulty Menu. Expected Result: Move to gameplay screen for hard game

Step 3. Watch until a teddy bear projectile collides with a teddy bear projectile. Expected Result: Both teddy bear projectiles explode

Step 4. Click X in corner of player. Expected Result: Exit game

We implement this functionality by adding an else if clause to the `OnTriggerEnter2D` method in the `TeddyBearProjectile` script:

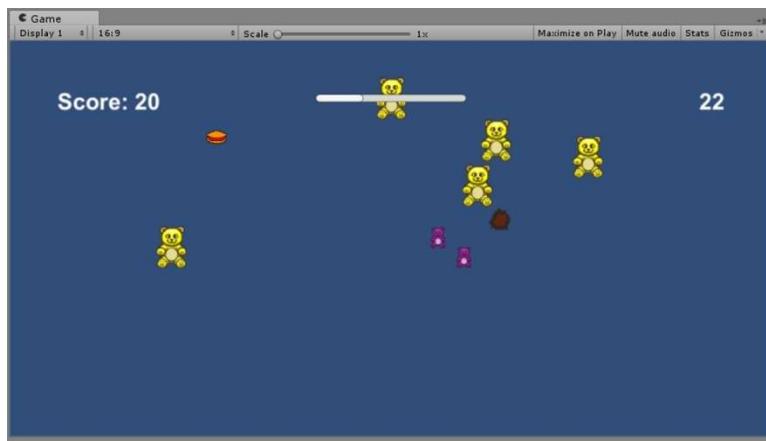
```

else if (other.gameObject.CompareTag("TeddyBearProjectile"))
{
    // if colliding with teddy bear projectile, destroy projectile and self
    Instantiate(prefabExplosion, other.gameObject.transform.position,
                Quaternion.identity);
    Destroy(other.gameObject);
    Instantiate(prefabExplosion, transform.position, Quaternion.identity);
    Destroy(gameObject);
}

```

To execute this test case, we left the burger out of the Gameplay scene and commented out the body of the if statement in the `OnTriggerEnter2D` method. That way, the only exploding collisions we'd see would be teddy bears projectiles colliding with teddy bear projectiles.

Test Case 23 passes fine, though it took a long time to actually see two teddy bear projectiles collide!



**Figure 20.15. Basic Gameplay in Action**

We've now finished all the basic gameplay functionality for the game. We've been running our test cases as we went along, confirming that they all work correctly. As for the menu test cases, if you haven't run each test case in the player yet you should do that now.

## 20.9. Design a Solution (Full Gameplay: AI Teddies)

It's time to make our teddy bears, and even our teddy bear projectiles, a little smarter. How can we do that? Let's discuss teddy bears first.

We're going to implement very rudimentary artificial intelligence (AI) here, where a teddy bear moves toward the burger rather than just moving randomly through the Gameplay scene. Smarter teddy bears will follow the burger more closely, even as the burger moves around, while less smart teddy bears can be more easily avoided by the burger.

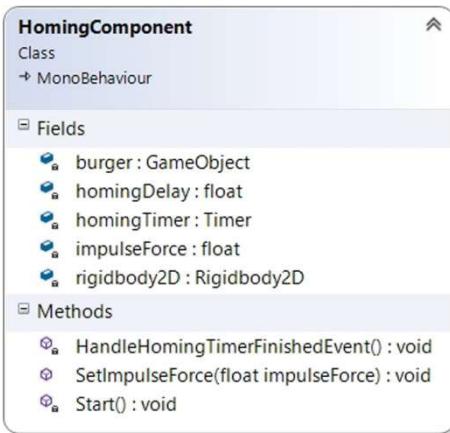
There are several ways to implement that behavior. One way would be to have the teddy bear move toward the burger with some probability on every update. Smarter bears would have a higher probability of doing this (a probability of 1 would mean the teddy bear reorients to the burger on every update) and less smart teddy bears would have a lower probability. This is certainly a workable solution, but given the way random number generation works, we might have a smart teddy bear that acts stupid for a while, then suddenly acts very smart for a while (and vice versa for a less smart teddy bear). This behavior might seem more erratic than is believeable to a player who selected a difficulty level (Hard) that implied the enemies would be consistently smart.

The approach we'll use instead leads to steady intelligence at a particular level throughout the game. We'll have the teddy bears use a timer to indicate when they should reorient toward the burger. Smart bears will use a shorter timer duration, so they'll follow the burger more closely, while less smart bears will use a longer timer duration. This is actually a reasonably general approach to use to affect the strength of AI; in our Battle Paddles game, we have the AI plan a set of actions to take, where the planning takes place more regularly (at shorter timer durations) for the smarter opponents.

If we use the current game difficulty to affect how often teddy bears reorient toward the burger, how should we use the game difficulty to affect how teddy bear projectiles behave? In exactly the same way! If we think of the teddy bear projectiles as "smart projectiles", they should also move toward the burger,

changing course as the burger changes direction. On harder difficulties, the projectiles will reorient toward the burger more often, perhaps because they have better sensors and control systems.

We'll implement the homing functionality in a `HomingComponent` script. The great news is that because Unity uses a component-based approach, we can attach that script to both the `TeddyBear` and `TeddyBearProjectile` prefabs to get the behavior we want from both those prefabs using a single script. Here's the UML for the `HomingComponent` script:



**Figure 20.16. HomingComponent Script UML**

## 20.10. Write Test Cases (Full Gameplay: AI Teddies)

### Test Case 24

#### **Watch Easy Teddy Bear and Teddy Bear Projectile Homing**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu
- Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game
- Step 3. Watch teddy bears and teddy bear projectiles periodically move toward the burger. Expected Result: Teddy bears and teddy bear projectiles periodically move toward the burger
- Step 4. Click X in corner of player. Expected Result: Exit game

### Test Case 25

#### **Watch Medium Teddy Bear and Teddy Bear Projectile Homing**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu
- Step 2. Click Medium button on Difficulty Menu. Expected Result: Move to gameplay screen for medium game
- Step 3. Watch teddy bears and teddy bear projectiles periodically move toward the burger. Expected Result: Teddy bears and teddy bear projectiles periodically move toward the burger more often than the easy game
- Step 4. Click X in corner of player. Expected Result: Exit game

**Test Case 26****Watch Hard Teddy Bear and Teddy Bear Projectile Homing**

Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu

Step 2. Click Hard button on Difficulty Menu. Expected Result: Move to gameplay screen for hard game

Step 3. Watch teddy bears and teddy bear projectiles periodically move toward the burger. Expected Result: Teddy bears and teddy bear projectiles periodically move toward the burger more often than the medium game

Step 4. Click X in corner of player. Expected Result: Exit game

It's unusual for us to have test cases that talk about relative performance compared to previous test cases (e.g., "more often than the easy game"), but we need to structure the test cases that way because all three difficulties exhibit the same behavior, just at different speeds.

## 20.11. Write the Code and Test the Code (Full Gameplay: AI Teddies)

To give the game tuners finer control of gameplay, we want them to be able to independently control the timing for the teddy bears and the teddy bear projectiles. Assuming we've added 6 new properties to the `ConfigurationUtils` class for those values, here's the new method we add to the `DifficultyUtils` class:

```
/// <summary>
/// Gets the homing delay for the given tag for
/// the current game difficulty
/// </summary>
/// <returns>homing delay</returns>
/// <param name="tag">tag</param>
public static float GetHomingDelay(string tag)
{
    if (tag == "TeddyBear")
    {
        switch (difficulty)
        {
            case Difficulty.Easy:
                return ConfigurationUtils.EasyBearHomingDelay;
            case Difficulty.Medium:
                return ConfigurationUtils.MediumBearHomingDelay;
            case Difficulty.Hard:
                return ConfigurationUtils.HardBearHomingDelay;
            default:
                return ConfigurationUtils.EasyBearHomingDelay;
        }
    }
    else
    {
        switch (difficulty)
        {
            case Difficulty.Easy:
                return ConfigurationUtils.EasyBearProjectileHomingDelay;
            case Difficulty.Medium:
                return ConfigurationUtils.MediumBearProjectileHomingDelay;
            case Difficulty.Hard:
                return ConfigurationUtils.HardBearProjectileHomingDelay;
        }
    }
}
```

```
        default:  
            return ConfigurationUtils.EasyBearProjectileHomingDelay;  
    }  
}
```

Although we could have the `HomingComponent` script call this method directly, up to this point in our implementation we've used `ConfigurationUtils` as the single access point for configuration data. We'll continue that approach here by adding a `GetHomingDelay` method to the `ConfigurationUtils` class as well:

```
    /// <summary>
    /// Gets the homing delay for the given tag
    /// </summary>
    /// <returns>homing delay</returns>
    /// <param name="tag">tag</param>
    public static float GetHomingDelay(string tag)
    {
        return DifficultyUtils.GetHomingDelay(tag);
    }
```

Now we can implement the `HomingComponent` script:

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

/// <summary>
/// A homing component
/// </summary>
public class HomingComponent : MonoBehaviour
{
    GameObject burger;
    new Rigidbody2D rigidbody2D;
    float impulseForce;
    float homingDelay;
    Timer homingTimer;
```

The `burger`, `rigidbody2D`, and `homingDelay` fields are to cache (store) values for efficiency; that way we don't have to retrieve those values each time we need them. We store the `impulseForce` so the object the script is attached to keeps moving at the same speed even when it changes direction to move toward the burger. The `homingTimer` is a standard `Timer` component we use to determine when it's time to "home" again.

```
/// <summary>
/// Use this for initialization
/// </summary>
void Start()
{
    // save values for efficiency
    burger = GameObject.FindGameObjectWithTag("Burger");
    homingDelay = ConfigurationUtils.GetHomingDelay(gameObject.tag);
    rigidbody2D = GetComponent<Rigidbody2D>();
```

We added a Burger tag to the Burger because finding objects by tag is faster (in general) than finding objects by name in Unity.

```
// create and start timer
homingTimer = gameObject.AddComponent<Timer>();
homingTimer.Duration = homingDelay;
homingTimer.AddTimerFinishedEventListerner(
    HandleHomingTimerFinishedEvent);
homingTimer.Run();
}

/// <summary>
/// Sets the impulse force
/// </summary>
/// <value>impulse force</value>
public void SetImpulseForce(float impulseForce)
{
    this.impulseForce = impulseForce;
}
```

Both the `TeddyBear` script and the `TeddyBearProjectile` script call this method from their `Start` methods. That way, the homing object maintains its original speed even when changing direction to move toward the burger.

```
/// <summary>
/// Handles the homing timer finished event
/// </summary>
void HandleHomingTimerFinishedEvent() {

    // stop moving
    rigidbody2D.velocity = Vector2.zero;

    // calculate direction to burger and start moving toward it
    Vector2 direction = new Vector2(
        burger.transform.position.x - transform.position.x,
        burger.transform.position.y - transform.position.y);
    direction.Normalize();
    rigidbody2D.AddForce (direction * impulseForce,
        ForceMode2D.Impulse);

    // restart timer
    homingTimer.Run ();
}
}
```

This is where we actually change our direction and start moving toward the burger. First we stop moving; otherwise, we'll keep adding force and the game object will move faster and faster over time. The block of code to start moving toward the burger is the same as the code we used in the Ted the Collector game in Chapter 9 to move the teddy bear toward the target pickup. Finally, we restart the homing timer (yes, we forgot to do that when we first wrote the method!) so we "home" again after the appropriate delay.

We can now execute Test Cases 24, 25, and 26. We temporarily reduced the max number of bears for each difficulty level to a single bear so we could easily observe the homing behavior of both the TeddyBear and the TeddyBearProjectiles in the scene. All three test cases pass.

## 20.12. Design a Solution (Sound)

Let's add some sound effects to our game. We won't add background music to the game (though we showed how to do that in Chapter 15), but we will add sound effects. Specifically, we'll add sounds for when the player pauses the game, menu button clicks, burger and teddy bear shots, burger damage, explosions, and if the player loses by running out of health.

Although we'll use the knowledge we gained about audio sources and audio clips in Chapter 15, it's actually amazing how quickly implementing sound effects can get complicated. In Chapter 15, we attached audio sources to the game objects that most logically played the sound effects we were including in the game, then had scripts attached to those game objects actually play the sound effects. Unfortunately, that approach won't work here.

We admit that we originally implemented our audio using that approach, but when we got to our explosions, the explosion sound effect was getting cut off. This was happening because we were destroying the Explosion game object once its animation finished playing, but that also destroys the Audio Source component attached to that game object (as it should). Destroying the Audio Source component stops playing the clip as well, so we didn't get the complete explosion sound effect.

This is, of course, a general game development problem. We were recently writing a small Asteroids game, and we wanted to play a sound when a bullet destroys an asteroid. We were destroying both of the game objects involved in that collision, so we couldn't have either one of them play the required sound effect. The approach we'll use here works in that scenario as well.

We'll implement a static `AudioManager` class that will hold an audio source and a dictionary of audio clips for the sound effects in the game. This class will expose an `Initialize` method to load all the audio clips, an `Initialized` property so we can tell whether or not the audio manager has been initialized, and a `Play` method the other classes in our game will call to play particular sound effects.

We'll also need an  `AudioSource` that persists across all the scenes in the game. We'll see how we can implement that when we Write the Code.

## 20.13. Write Test Cases (Sound)

### Test Case 27

#### User Interface Sound Effects

Step 1. Click High Score button on Main Menu. Expected Result: Move to High Score Menu. Menu button click sound plays

Step 2. Click Quit button on High Score Menu. Expected Result: Move to Main Menu. Menu button click sound plays

Step 3. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu. Menu button click sound plays

Step 4. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game. Menu button click sound plays

- Step 5. Press Escape key. Expected Result: Game paused and Pause Menu displayed on top of game. Pause game sound plays
- Step 6. Press Resume button on Pause menu. Expected Result: Pause Menu removed and game unpause. Menu button click sound plays
- Step 7. Press Escape key. Expected Result: Game paused and Pause Menu displayed on top of game. Pause game sound plays
- Step 8. Press Quit button on Pause menu. Expected Result: Move to Main Menu. Menu button click sound plays
- Step 9. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu. Menu button click sound plays
- Step 10. Click Medium button on Difficulty Menu. Expected Result: Move to gameplay screen for medium game. Menu button click sound plays
- Step 11. Press Escape key. Expected Result: Game paused and Pause Menu displayed on top of game. Pause game sound plays
- Step 12. Press Quit button on Pause menu. Expected Result: Move to Main Menu. Menu button click sound plays
- Step 13. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu. Menu button click sound plays
- Step 14. Click Hard button on Difficulty Menu. Expected Result: Move to gameplay screen for hard game. Menu button click sound plays
- Step 15. Press Escape key. Expected Result: Game paused and Pause Menu displayed on top of game. Pause game sound plays
- Step 16. Press Quit button on Pause menu. Expected Result: Move to Main Menu. Menu button click sound plays
- Step 17. Click Quit button on Main Menu. Expected Result: Exit game (no menu button click sound plays because the application quits so quickly)

## **Test Case 28**

### **Gameplay Sound Effects**

- Step 1. Click Play button on Main Menu. Expected Result: Move to Difficulty Menu
- Step 2. Click Easy button on Difficulty Menu. Expected Result: Move to gameplay screen for easy game
- Step 3. Shoot french fries using the space bar. Expected Result: French fries move straight up from burger when shot. Firing rate controlled when space bar held down. French fries shooting sound plays
- Step 4. Move burger to collide with teddy bear. Expected Result: Teddy bear explodes. Health bar shows reduced health. Explosion sound and burger damage sound plays
- Step 5. Watch teddy bears periodically shoot teddy bear projectiles. Expected Result: Teddy bear projectiles periodically move toward the burger when shot. Teddy bear shooting sound plays
- Step 6. Move burger to collide with teddy bears and teddy bear projectiles until health is 0. Expected Result: High Score Menu displayed above Gameplay scene. All moving objects are paused. Player running out of health sound plays
- Step 7. Click Quit button on High Score Menu. Expected Result: Move to Main Menu
- Step 8. Click Quit button on Main Menu. Expected Result: Exit game

### **20.14. Write the Code and Test the Code (Sound)**

Create a new audio folder in the Project window and create a Resources folder in the new audio folder. Copy the sound effect files into that folder.

We'll start with the pause game sound effect, but getting that sound effect to play will require getting the whole audio system set up.

First, we'll implement the `AudioManager` class:

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

/// <summary>
/// The audio manager
/// </summary>
public static class AudioManager
{
    static bool initialized = false;
    static AudioSource audioSource;
```

The `initialized` field is used to keep track of whether or not the audio manager has been initialized yet. The `audioSource` field holds the  `we'll use to play all the sound effects.`

```
static Dictionary<AudioClipName, AudioClip> audioClips =
    new Dictionary<AudioClipName, AudioClip>();
```

We use a dictionary of audio clips so we can look up the audio clips to play by audio clip name.

```
/// <summary>
/// Gets whether or not the audio manager has been initialized
/// </summary>
public static bool Initialized
{
    get { return initialized; }
}
```

At this point in the book, you should be able to easily understand this property!

```
/// <summary>
/// Initializes the audio manager
/// </summary>
/// <param name="source">audio source</param>
public static void Initialize(AudioSource source)
{
    initialized = true;
    audioSource = source;
```

The code above captures the fact that the audio manager has now been initialized and saves the provided  `AudioSource` to use to play clips later.

```
audioClips.Add(AudioClipName.BurgerDamage,
    Resources.Load<AudioClip>("BurgerDamage"));
audioClips.Add(AudioClipName.BurgerDeath,
    Resources.Load<AudioClip>("BurgerDeath"));
```

```

        audioClips.Add(AudioClipName.BurgerShot,
            Resources.Load<AudioClip>("BurgerShot"));
        audioClips.Add(AudioClipName.Explosion,
            Resources.Load<AudioClip>("Explosion"));
        audioClips.Add(AudioClipName.MenuButtonClick,
            Resources.Load<AudioClip>("ButtonClick"));
        audioClips.Add(AudioClipName.PauseGame,
            Resources.Load<AudioClip>("ButtonClick"));
        audioClips.Add(AudioClipName.TeddyShot,
            Resources.Load<AudioClip>("TeddyShot"));
    }
}

```

The rest of the method loads each of the audio clips for the game and puts them in the dictionary using the audio clip name as the key. Notice that we're using the `ButtonClick` audio clip for both the menu button click and pause game sound effects.

One additional benefit of using the dictionary of audio clips is that we can implement all the game functionality using the in-game audio clip names independent of the names of the actual sound files you've included in your Unity project. That way, if someone renames an asset (yes, that's going to happen to you!) you can simply change the name of the audio clip you're loading in this method without having to change any of your other code. Trust us, that's a win!

```

/// <summary>
/// Plays the audio clip with the given name
/// </summary>
/// <param name="name">name of the audio clip to play</param>
public static void Play(AudioClipName name)
{
    audioSource.PlayOneShot(audioClips[name]);
}
}

```

The method above plays a particular audio clip from the dictionary. We first retrieve the audio clip from the dictionary using `audioClips[name]`, then we play it using the  `AudioSource PlayOneShot` method. Why aren't we using the `Play` method we used in Chapter 15?

The `PlayOneShot` method is like a "fire and forget" way to play an audio clip. The downside is that we can't pause or stop the clip once we start playing it, but that's okay, we don't need that functionality anyway. The big win here is that our audio source can play multiple clips simultaneously, which we definitely want in our game. We didn't have to worry about this in Chapter 15 because each game object had its own audio source playing its own clip, so we could have many of those clips playing at the same time. We only have one audio source in this game, though, so using the `PlayOneShot` method to play multiple clips simultaneously is the way to go.

The next thing we need is an  `AudioSource` that persists across all the scenes in the game. As we discussed above, when a game object is destroyed, the Audio Source component for that game object is destroyed as well. Because our `audioSource` field in the  `AudioManager` class is a reference to an Audio Source component, we need to make sure that component (and the game object it's attached to) stays around for the life of the game.

Start by opening the MainMenu scene, right clicking in the Project window, and selecting Create Empty. Change the name of the new game object to Game AudioSource. In the Inspector, click the Add Component button and select Audio > Audio Source. There's no need to assign a clip to the Audio Source because our audio manager will select the appropriate clips to play as the game runs.

Now we need to attach a script to the Game AudioSource game object to initialize the audio manager and to make the game object persist across all the scenes in the game:

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

/// <summary>
/// An audio source for the entire game
/// </summary>
public class Game AudioSource : MonoBehaviour
{
    /// <summary>
    /// Awake is called before Start
    /// </summary>
    void Awake()
    {
        // make sure we only have one of this game object
        // in the game
        if (!AudioManager.Initialized)
        {

```

We actually need to make sure we only have one instance of the Game AudioSource game object in the entire game because we don't want our audio manager to end up with multiple audio sources. There are a variety of different ways we can check this, so we decided to simply check to see if the audio manager has already been initialized. If it hasn't, this is the first time we've called the `Awake` method for a Game AudioSource game object, so we know this is the first instance of that game object being added to the game.

You might be wondering how we could end up with multiple copies of the Game AudioSource game object in the game anyway, since we've added it to the MainMenu scene, which is the scene our game starts in. The issue is that any time we go to the Main Menu we'll try to create an instance of the Game AudioSource game object, which will happen whenever we return to Main Menu from the High Score Menu or by quitting a game from the Pause Menu.

```
// initialize audio manager and persist audio source across scenes
AudioSource audioSource = gameObject.AddComponent<AudioSource>();
AudioManager.Initialize(audioSource);
DontDestroyOnLoad(gameObject);
```

The first two lines of code above add an Audio Source component to the Game AudioSource game object and pass that component in to the audio manager. The last line of code calls the `DontDestroyOnLoad` method, which keeps the game object alive as we move between the scenes in the game.

```
        }
    else
    {
        // duplicate game object, so destroy
        Destroy(gameObject);
    }
}
```

The else clause above makes sure we only have one instance of the Game AudioSource game object in the game. We only get to the else clause if the audio manager has already been initialized, so we know there's already an instance of the Game AudioSource game object in the game. In that case, we simply destroy the additional instance of the Game AudioSource game object that was just created.

Okay, we're finally ready to play the pause game sound effect! Given all the up-front work we've done, we can do that by adding the following line of code to the `FeedTheTeddies` Update method at the end of the if clause:

```
AudioManager.Play(AudioClipName.PauseGame);
```

Go ahead and run Test Case 27. You won't hear any of the menu button clicks yet, but you should hear the pause game sound effect in Steps 5, 7, 11, and 15.

To get the rest of Test Case 27 to pass, we need to add the menu button clicks every time we click a menu button. To do that, add the following line of code at the beginning of each of the `Handle<ButtonName>ButtonOnClickEvent` methods in the `DifficultyMenu`, `HighScoreMenu`, `MainMenu`, and `PauseMenu` scripts:

```
AudioManager.Play(AudioClipName.MenuButtonClick);
```

Run Test Case 27 again. Everything should work fine, so we can move on to Test Case 28.

Although Test Case 28 has us testing all the gameplay sound effects, we'll add them one at a time in the order they're tested in the test case. That means we'll start with the french fries shooting sound.

Add the following code to the `Burger` `Update` method right after we instantiate a new french fries object:

```
AudioManager.Play(AudioClipName.BurgerShot);
```

Run the game now and you'll hear the sound effect whenever the burger shoots french fries.

You'll actually notice if you pause and resume or quit the game, the burger fires french fries when you click a Pause Menu button. That's because the left mouse button is included as the Alt Positive Button in the Fire1 axis in the Input manager. Go delete mouse 0 from that value in the Input settings and rerun the game to see that's solved the problem.

Let's add the explosion sound next. Modify the `Explosion` script to tell the audio manager to play the `AudioClipName.Explosion` clip at the end of the `Start` method. Run the burger into a teddy bear in the game; you should hear the explosion sound when you do.

For the burger damage sound, add code to the `Burger` `TakeDamage` method right after reducing the burger health to tell the audio manager to play the `AudioClipName.BurgerDamage` clip. Run the burger into a teddy bear in the game; you should hear the burger damage sound when you do. The explosion sound is also playing at the same time, though it's a little hard to hear both. You can certainly shoot a teddy bear with french fries to confirm the explosion sound is still working if you'd like.

Modify the `TeddyBear` script to tell the audio manager to play the `AudioClipName.TeddyShot` clip when the teddy bear instantiates a teddy bear projectile. Execute up to Step 5 of Test Case 28 to hear the teddy bears shooting.

The last sound effect we need to add is the player running out of health sound. Add code to the `Burger` `TakeDamage` method to tell the audio manager to play the `AudioClipName.BurgerDeath` clip if the burger runs out of health. Execute Test Case 28 in its entirety to confirm all the sound effects are working properly.

## 20.15. Design a Solution (XML Configuration Data)

Up to this point, we've hard-coded configuration values into the properties the `ConfigurationUtils` class exposes. We did that so we wouldn't have to change the code in any of the consumers of those properties once we implemented our XML configuration data approach. We will of course have to change the implementation of the `ConfigurationUtils` class itself, but we won't change any of its external interfaces. Isn't information hiding a beautiful thing?

We're going to use the same structure for this part of our project as we did in Section 19.5. Specifically, we'll have a `ConfigurationData` XML file to store our values, a `ConfigurationData` script to retrieve those values from the XML file, and the `ConfigurationUtils` class we already have to expose those values to the rest of the game.

## 20.16. Write Test Cases (XML Configuration Data)

We have 35 different configuration data values and we need to confirm that for each one a change in the value in the XML file leads to an appropriate change when we run the game. It doesn't make sense to write 35 more test cases, one for each configuration data value, so instead we write a single test case to be executed 35 times:

### Test Case 29

#### **XML Configuration Data**

Step 0. Before running the game, change a single configuration data value in the `ConfigurationData.xml` file

Step 1. Play the game. Expected Result: Game behaves appropriately based on the configuration data value

The Expected Result clearly violates our rule of having a specific result, but in this case pragmatism wins.

## 20.17. Write the Code and Test the Code (XML Configuration Data)

Create a new `ConfigurationData` script in the scripts\util folder in the Project window and add fields and properties for all 35 configuration data values like we did in Section 19.5. Add a static `configurationData` field to the `ConfigurationUtils` class and change all the properties that return a hard-coded value to return the corresponding property from the `configurationData` field instead.

Instead of manually creating our XML file, let's just generate one instead. Add the following code at the end of the `ConfigurationData` class:

```
#region Temporary development support

/// <summary>
/// Temporary private constructor
/// </summary>
ConfigurationData()
{
}

/// <summary>
/// Creates an XML file that contains the default values for the
/// configuration data
///
/// NOTE: This method should only be used during development
/// </summary>
public static void SaveDefaultValues()
{
    ConfigurationData defaultData = new ConfigurationData();
    FileStream writer = new FileStream("ConfigurationData.xml",
        FileMode.Create);
    DataContractSerializer ser =
        new DataContractSerializer(typeof(ConfigurationData));
    ser.WriteObject(writer, defaultData);
    writer.Close();
}

#endregion
```

The above code writes the default values for all the fields in the `ConfigurationData` class to the `ConfigurationData.xml` file using the appropriate format.

Add an `Initialize` method to the `ConfigurationUtils` class and call this method in the `Start` method in the `GameInitializer` class right before calling the `DifficultyUtils Initialize` method. Now add the following temporary code to the `ConfigurationUtils Initialize` method:

```
// temporary to write initial configuration data xml file
ConfigurationData.SaveDefaultValues();
```

Run the game, being sure to start from the Main Menu scene. You can just exit the game right after starting it up. Use your OS to navigate to your project folder. You'll see that the `ConfigurationData.xml` file has been created there. Although it's not necessary for our game to work properly, you'll probably want to add line breaks to the file to make it easier to edit as you change the configuration values.

Now that we've generated the XML file, comment out the region of code above in the `ConfigurationData` class (there's no reason for us to ship this code when we distribute our game) and the code in the `ConfigurationUtils` `Initialize` method that calls the `ConfigurationData` `SaveDefaultValues` method. You can of course delete that code instead, it's your choice, but we wanted to save it in case we need to use it again in the future.

Add the following using directives to the `ConfigurationUtils` class

```
using System.IO;
using System.Runtime.Serialization;
using System.Xml;
```

and add the following code to the body of the `ConfigurationUtils` `Initialize` method:

```
// deserialize configuration data from file into internal object
FileStream fs = null;
try
{
    FileStream fs = new FileStream("ConfigurationData.xml", FileMode.Open);
    XmlDictionaryReader reader = XmlDictionaryReader.CreateTextReader(
        fs, new XmlDictionaryReaderQuotas());
   DataContractSerializer ser =
        new DataContractSerializer(typeof(ConfigurationData));
    configurationData = (ConfigurationData)ser.ReadObject(reader, true);
    reader.Close();
}
finally
{
    // always close input file
    if (fs != null)
    {
        fs.Close();
    }
}
```

Run the game to make sure everything works fine. Now you can run Test Case 29 for each of the configuration data values to make sure they work properly. Of course, because we've been using `ConfigurationUtils` as the single access point for the rest of the game throughout our development, you shouldn't find any problems.

You've probably realized that someone editing the XML file could include values that don't really make any sense, like -5 for the number of points a teddy bear is worth. Although you as the programmer probably wouldn't do that, non-programmers editing the file could do so either by mistake or maliciously.

Although we didn't include any data validation in our `ConfigurationData` class, that's certainly something we could have done. In fact, the Battle Paddles game we give away on the Burning Teddy web site comes with an XML file containing 52 different configuration data values that can be used to tune the game. Because anyone downloading that game can edit that file (we want people to play with the game tuning!), we included data validation in the `ConfigurationData` class for that game to help protect XML editors from themselves. That's actually a MonoGame game, not a Unity game, but the same ideas apply.

## 20.18. Conclusion

Well, that was a LOT of work to build a very simple game! The bad news is that it is a lot of work to program games. The good news is that it's a lot of fun doing it. Even better, you now have a solid foundation for turning your own game ideas into reality using C# and Unity. Rock on.

## Appendix: Setting Up Your Development Environment

Learning to program requires that you actually DO lots of programming – not just read about it, hear about it, or watch someone else do it, but actually do it yourself. That means that you'll need to set up a programming environment so you can learn all the stuff in this book by actually slinging code. For the topics covered in this book, you'll need to install Visual Studio Community 2019 and the Unity game engine. Doing this will let you use Visual Studio to write C# console applications and will also let you write C# scripts in Unity as you develop Unity games. We'll discuss how to install each below.

### A.1. Visual Studio Community 2019

Microsoft's commercial Integrated Development Environments (IDEs) are called Visual Studio, and they come in various capability (and cost) levels. Because there are lots of people who want to program as a hobby rather than as a commercial endeavor, Microsoft also distributes free IDEs. We used the free Visual Studio Community 2019 IDE to develop all the code in this book. Although the source code provided on the accompanying web site can be used in any IDE, if you have Visual Studio Community 2019 you can just open up and use the projects as is.

Here's how you can get this IDE installed:

1. Download Visual Studio Community 2019 at <http://www.visualstudio.com/>.
2. Run the install file you downloaded.

Be sure to check the following items in the Workloads area of the installer:

Windows: .NET desktop development

Mobile & Gaming: Game development with Unity

You should also check the following in the Individual components area of the installer:

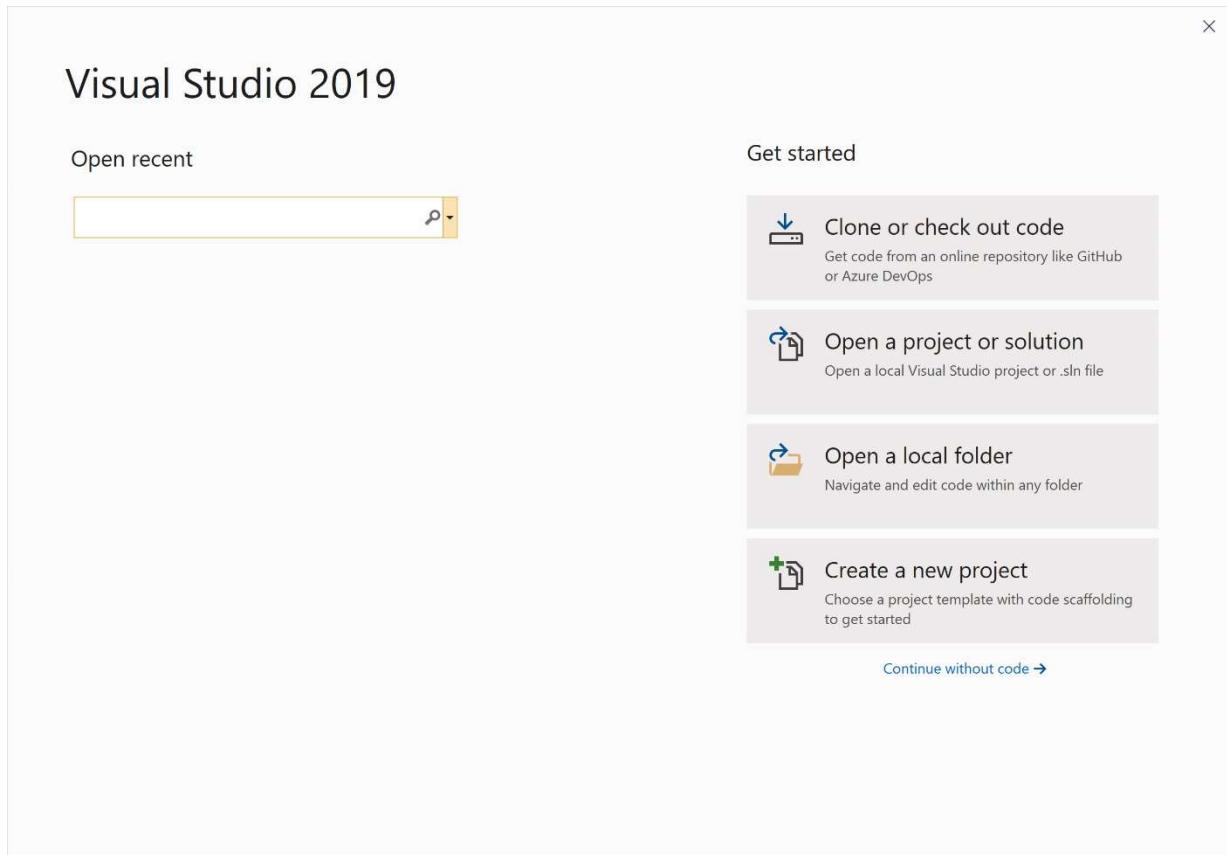
Code tools: Help Viewer

You can certainly select other Workloads and Individual components if you'd like, but we just installed the three items above. After selecting those items, click the Install button and wait patiently while everything is installed.

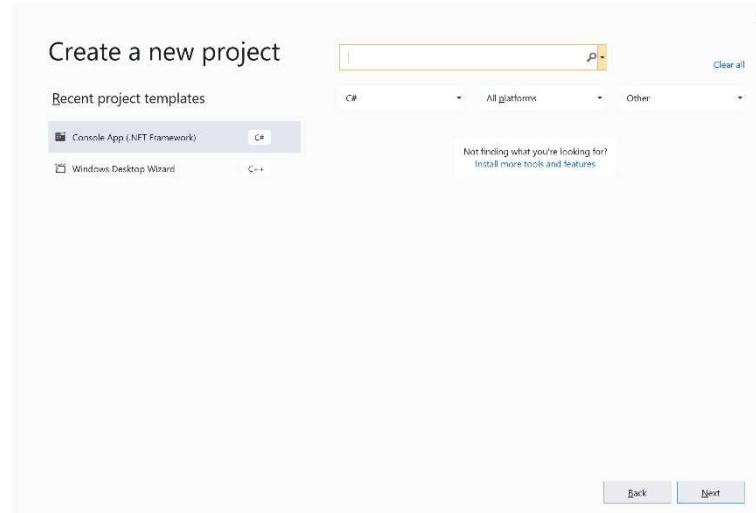
At this point, you should have a working version of the IDE installed. Let's check to make sure you do.

We put a shortcut on our desktop to make launching the IDE easier; for us, the IDE is installed at C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\devenv.exe. Of course, you can also use Windows to search on Visual Studio and click the Visual Studio 2019 result instead if you prefer.

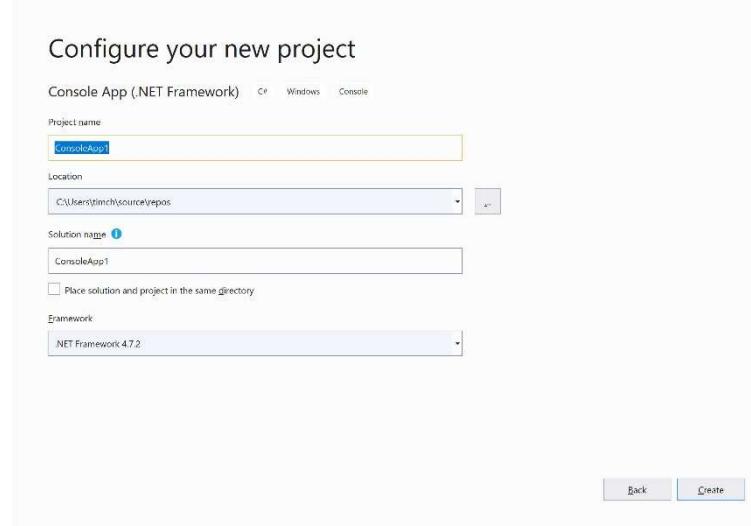
During the first startup of Visual Studio, be sure to change Development Settings to Visual C# before clicking the Start Visual Studio button on the bottom right. When Visual Studio starts up, you should end up with something like the window below. Yours may not match exactly, but it should be similar.



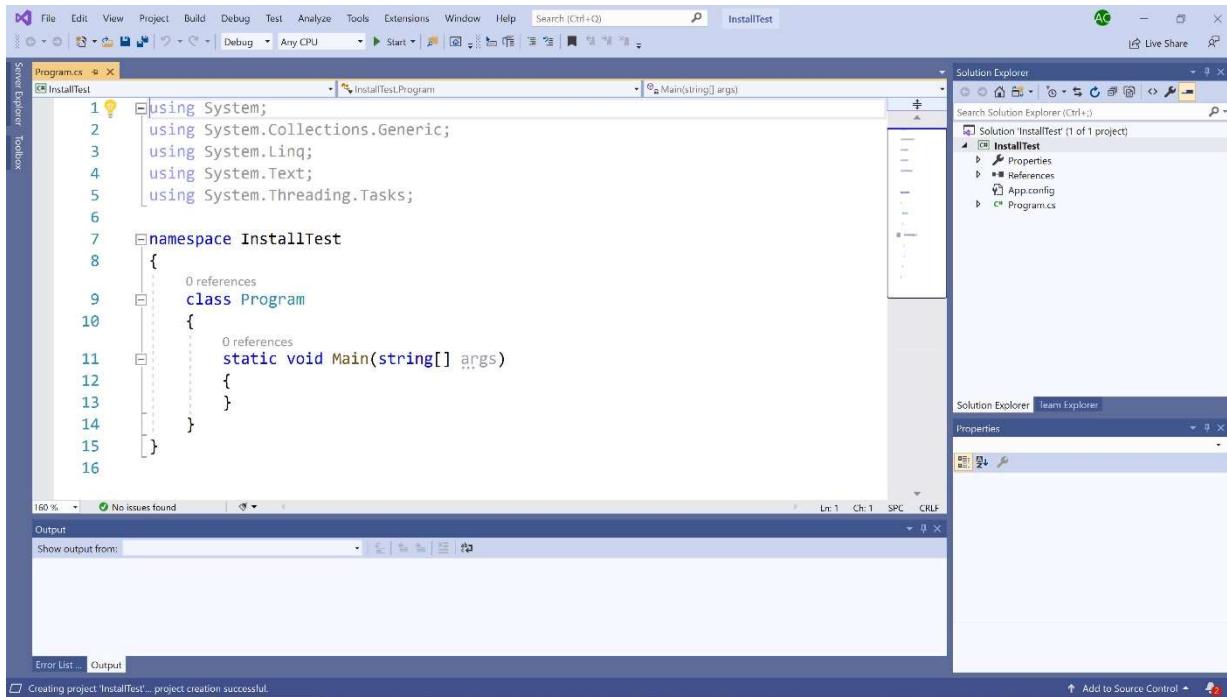
We're going to compile and run a simple program to make sure the IDE installed correctly. Click the Create a new project box on the lower right, which will bring you to a screen like the one below. Select the Console App (.NET Framework) C# button on the upper left, then click the Next button on the lower right.



You'll end up at a screen like the one shown below. Change the Project name from `ConsoleApp1` to `InstallTest` in the text box at the top left of the screen, change the Location to put the project wherever you want it to be saved, then click the Create button at the bottom right.



After a moment or two, you'll come back to the main screen of the IDE, which should look like this (we zoomed in on the Code window in the upper left using Ctrl + middle mouse wheel):



If you have the Toolbox pane displayed on the left-hand side, you can simply click the Auto Hide "pin" just to left of the X in the upper right corner of the Toolbox pane to make it hide itself on the left.

When we told the IDE we wanted a new console app project, it automatically gave us a template for that kind of application. Compile the application by pressing F6 or by selecting Build > Build Solution from the top menu bar<sup>35</sup>. Once you see a "Build succeeded" message in the bar at the bottom of the IDE, you

---

<sup>35</sup> We've actually seen the keyboard shortcut for building the solution as F7 and Ctrl+Shift+B also. If you select Build > Build Solution from the top menu bar, the keyboard shortcut for that action is shown to the right. You can also customize the keyboard shortcuts in Visual Studio if you'd like; see <https://msdn.microsoft.com/en-us/library/5zwses53.aspx>

can actually run the application. Do that by pressing Ctrl+F5; when you do, you should get the Command Prompt window below. Just press any key to return to the IDE.



Your Command Prompt window probably has the default black background with light grey text on it rather than a white background with black text. Although that works fine on a computer screen, it looks horrible in an electronic or printed book! If that's the only difference between your output and the output above, you're good to go. If you decide you want to change your defaults, open a Command Prompt window, right click near the top of the window, select Defaults, click the Colors tab, and change away. We also changed the Font, so feel free to customize that window as much as you want.

Exit the IDE. You certainly don't need to save the project if you don't want to – you'll get plenty of practice creating and saving projects – but feel free to do so if you want to.

If everything worked as described here, Visual Studio Community 2019 is installed and you're ready to move on to setting up the documentation. If your IDE isn't working, you should try working through the install process again or get online with Microsoft to ask for help.

When we're developing code, it's really useful to have the help documentation available directly within the IDE. The default for help documentation is to simply access help (through the IDE) online. If your computer always has connectivity to the Internet, this should work fine. If you're sometimes disconnected while you're programming, though, you'll probably want to set up your environment to use local help instead.

Start up the IDE, then click Help > Add and Remove Help Content. The IDE starts up the Microsoft Help Viewer, which lets you add local content. At a minimum, you should add the following topics:

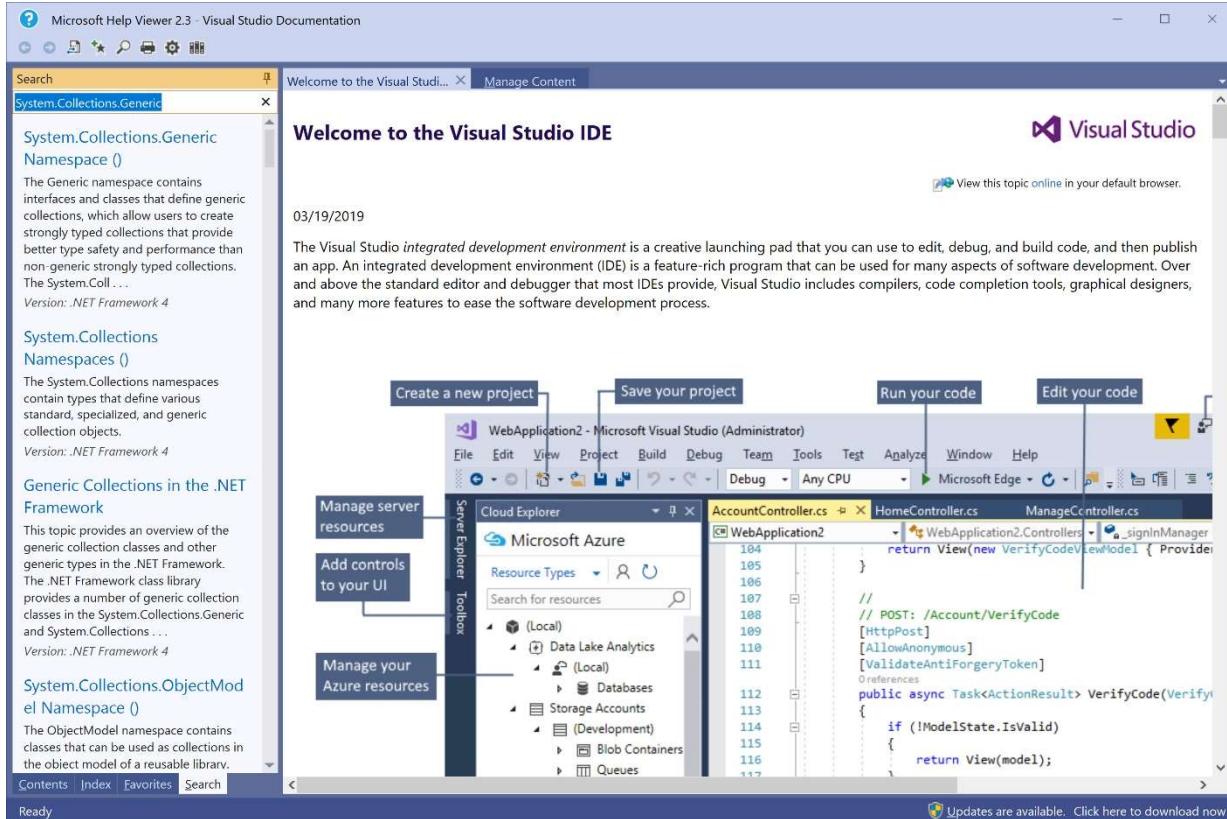
- .NET API Reference (already set to be added by default)
- Visual C# (already set to be added by default)

Click the Update button near the lower right of the Help Viewer and wait patiently while the update packages are downloaded and installed.

Finally, you can read the help documentation through a browser (which will access the online documentation) or through the Microsoft Help Viewer (which will access the local help content). Both approaches provide access to the same help content, they just look slightly different – and, of course, if you're offline you can only access the local help. If you prefer using a browser, you don't need to make

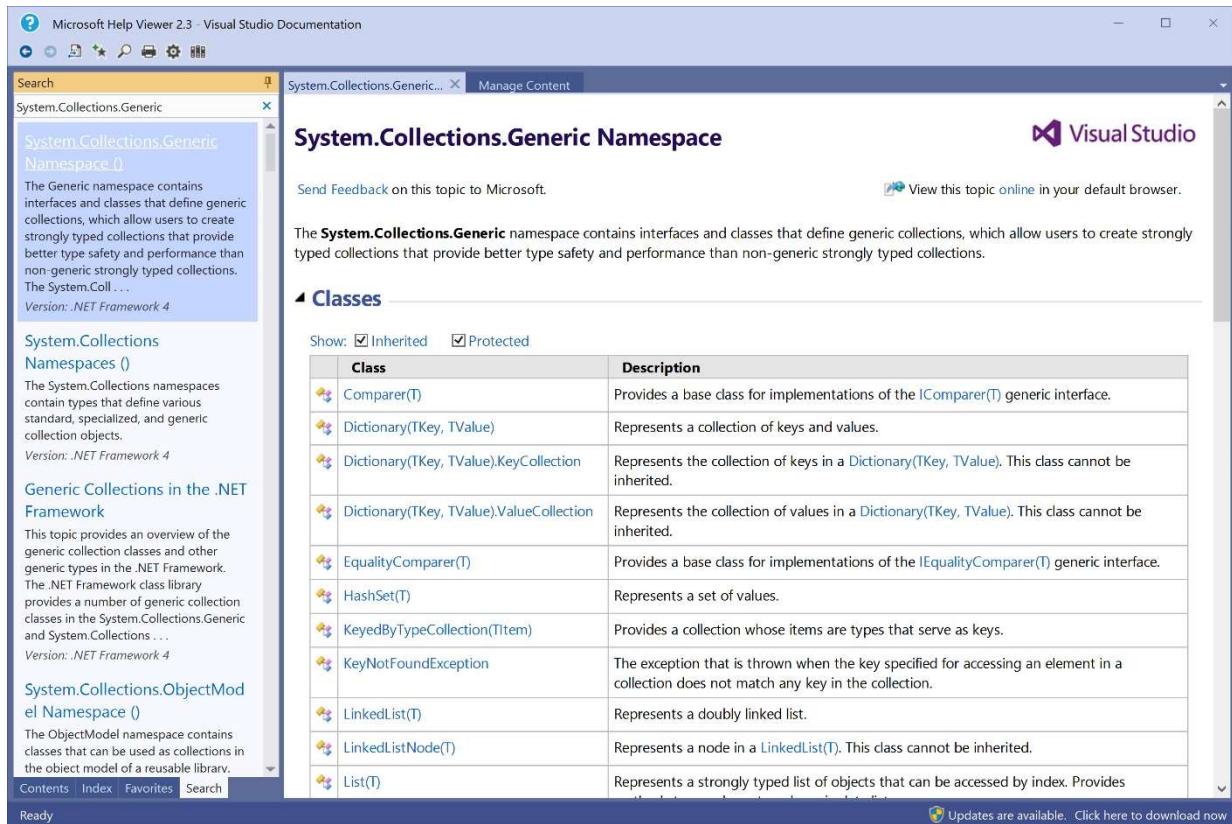
any changes because that's the default. If you prefer using the help viewer (like we do), select Help > Set Help Preference > Launch in Help Viewer.

Let's make sure the local documentation installed properly. Click Help > View Help. Enter System.Collections.Generic in the search box and press Enter. You should get a page of search results like the one shown below.



Your page will look different if you're using online help, but you should see a set of results somewhere on the page.

Click on the link that say System.Collections.Generic Namespace () and you should get a window of documentation like the one shown below.



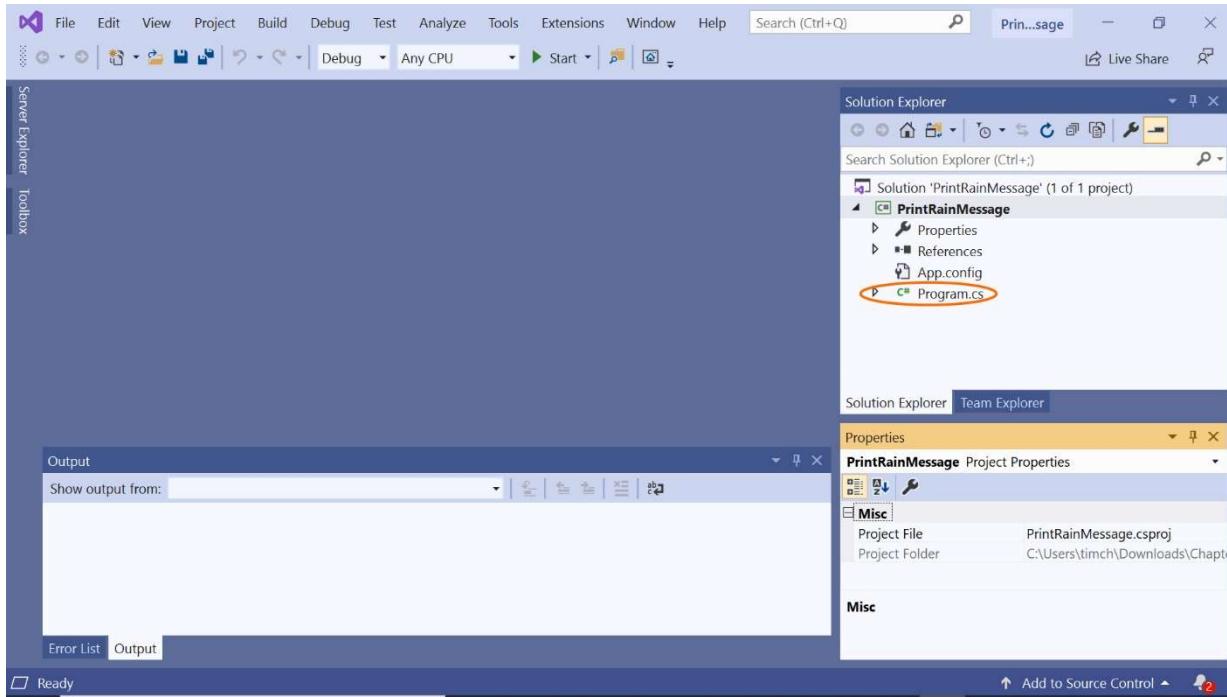
You should also know that you can always look at the C# documentation online at <https://msdn.microsoft.com/en-us/library/>; just click on the magnifying glass near the upper right of the page and use the search box to look for a specific C# topic.

### *Opening Downloaded Visual Studio Projects*

When you download a Visual Studio project, you can open it in Visual Studio by double clicking the solution file. The solution file has a .sln file extension, but in case you're not showing file extensions in Windows Explorer it's the file marked "Microsoft Visual St...", "Visual Studio Solu...", or something similar as the Type. Let's see how that works.

Download the Chapter 2 code from the Burning Teddy web site and extract the zip file somewhere on your computer.

Navigate into the folder called PrintRainMessage (it will be wherever you extracted the Chapter 2 code zip file) and double click the solution file. When the project opens, you might think something is wrong because the code pane doesn't show any code. Simply click Program.cs in the Solution Explorer to see the code in that file.

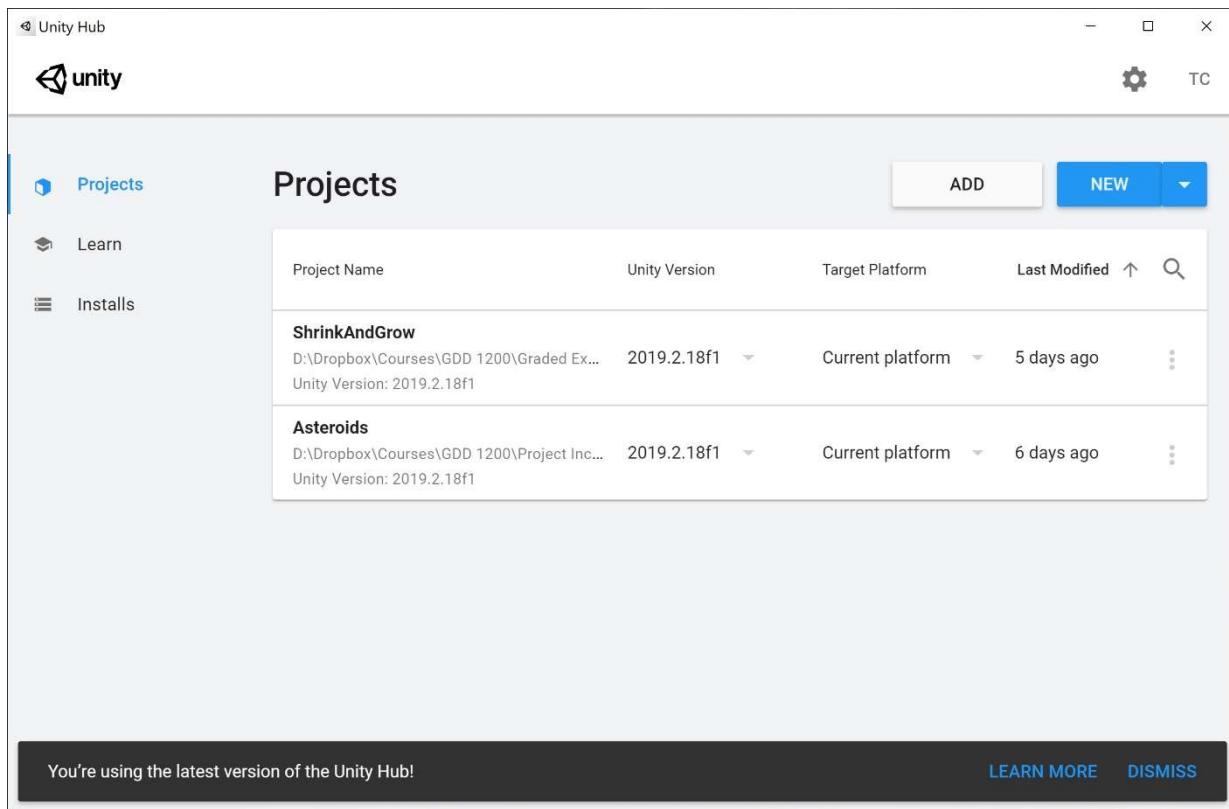


## A.2. Unity

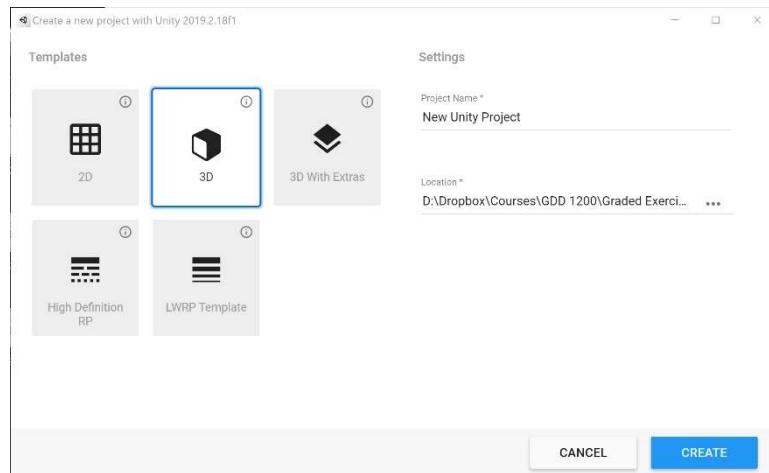
Here's how you can get Unity installed:

1. Download Unity Hub from <http://unity.com/>. At the time of this writing, you'll need to click Get started, select the Individual tab, click Get started under Individual, click Start here under First-time Users, and click Agree and download. The sequence of steps may be different by the time you're doing this, but the bottom line is you're downloading Unity Hub (even though the web pages don't say you're doing that, your downloaded file should be named something like UnityHubSetup.exe)
2. Run the install file you downloaded. Just accepting the default values as you go along should work fine, but you can of course change them if you want

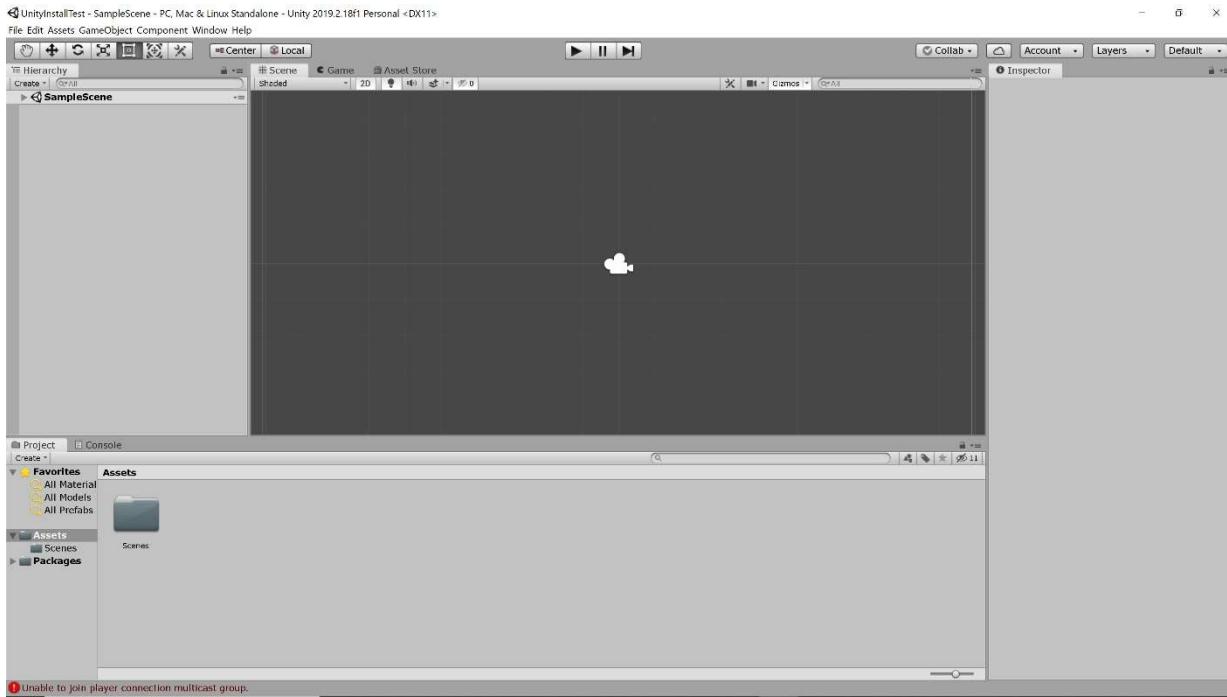
Unity Hub lets you have multiple versions of Unity installed on your computer and using Unity Hub is the standard way to install new versions of Unity. By default, Unity installs a shortcut to Unity Hub on your desktop when it installs. Double click that shortcut; you should end up with a window like the one below (though you won't have the list of recent projects, of course!). You may actually get a popup asking you to sign into your Unity account first; you can choose to create an account and sign into it or work offline, your choice.



Click New near the upper right corner; you should end up with a window like the one below.

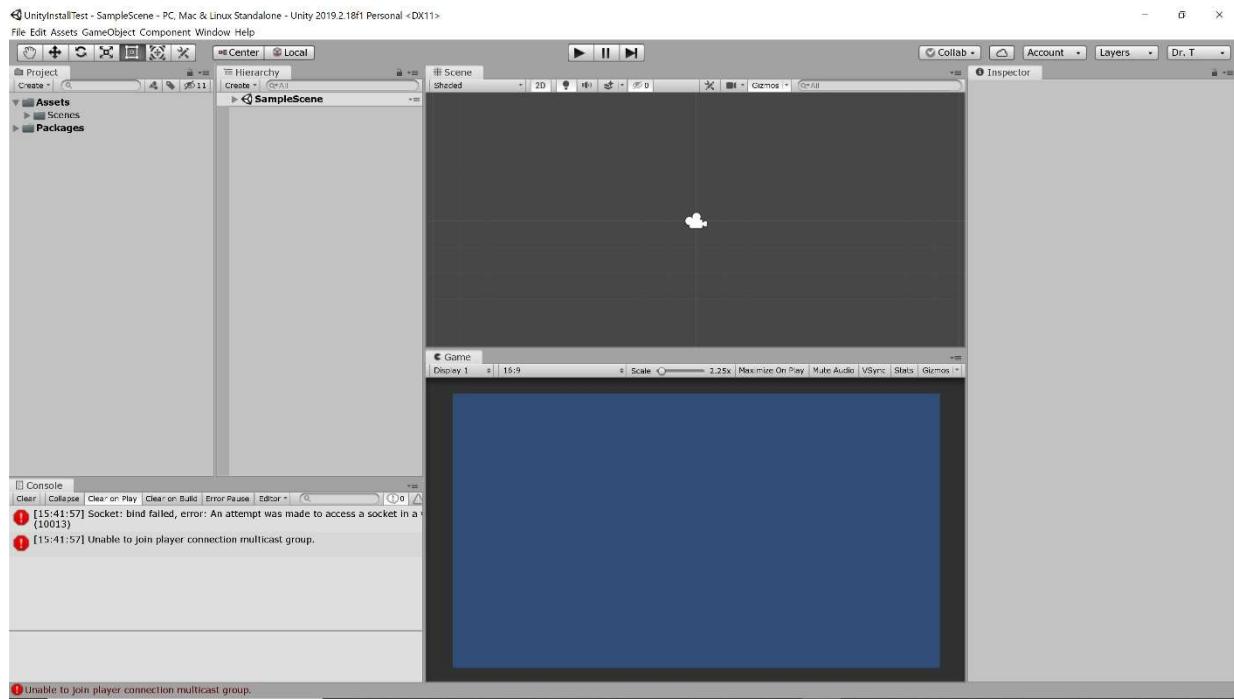


Click the 2D button to create a 2D project, change the Project Name to UnityInstallTest, change the Location to wherever you want to save the project, then click the Create button at the lower right. After a moment of two, you'll come to the Unity editor, which should look like this:

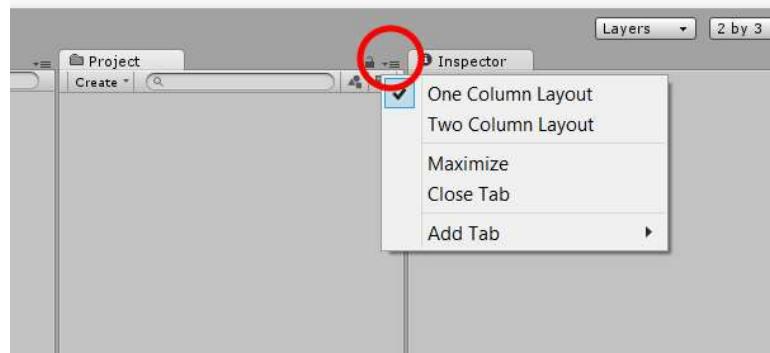


If the editor has a Services tab on the right, you can just right click on that tab and select Close Tab.

This is the default layout for the Unity editor, but you can change the layout by clicking the layout dropdown near the upper right and selecting a different layout. You can even configure and save your own layout; because we prefer a different layout when we develop in Unity, we'll show you how to configure and save the layout we'll use throughout the book (a layout that's also used in Jeremy Gibson's excellent *Introduction to Game Design, Prototyping, and Development* book). You don't have to do this – you can use any layout you want! – but go ahead and follow along if you want to use the layout we use. Once we're done our layout will look like this (don't worry about the errors showing up in the image below, they won't affect us):

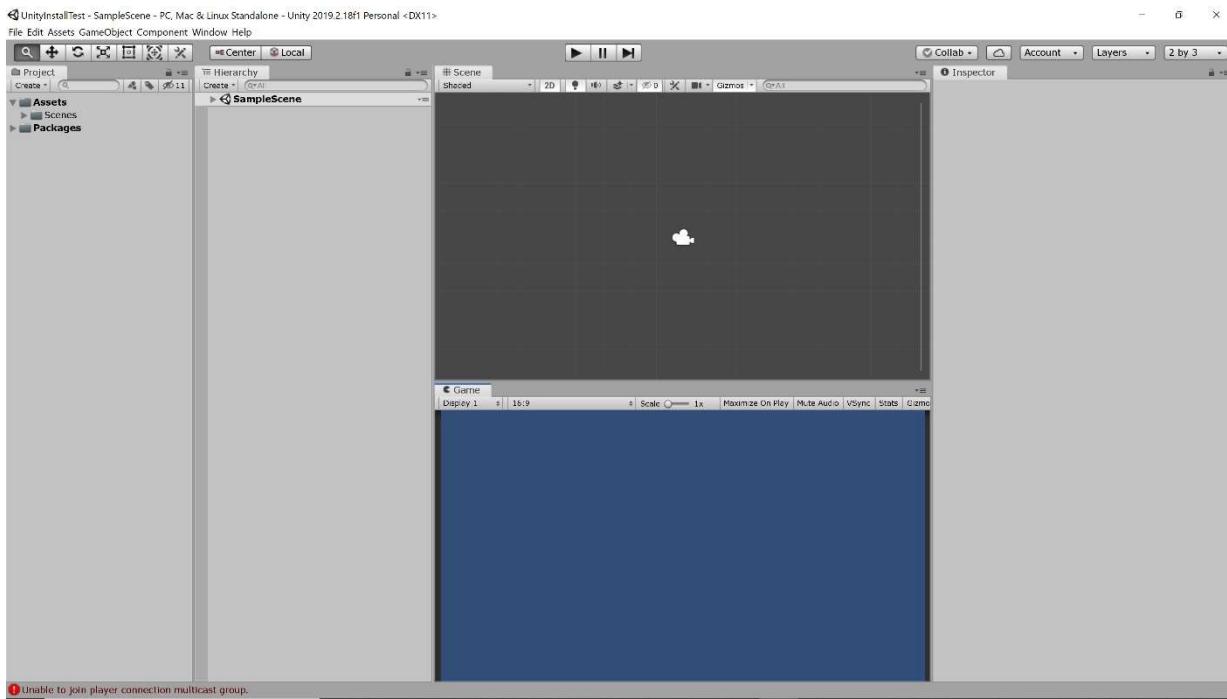


We'll use the 2 by 3 layout as our starting point, so select 2 by 3 from the layout dropdown on the upper right. Left click on the options selector on the Project window (circled in red in the figure below) and select One Column Layout.



We can drag windows around in the Unity by holding the left mouse button down on the tab for the window, then dragging the window to the desired location. This might take a little practice to make sure the window is "docked" where you want it rather than just floating free, but you should be able to get the hang of this pretty easily.

Click the dropdown that says Free Aspect at the top of the Game view and select 16:9. Move the windows around until your setup looks like the image below. As is typical in many apps, you can move the borders of particular windows by dragging their edges.



The last thing we want to add to our layout is a Console window (which acts much like the command prompt window in our console apps). Select Window > General > Console from the menu bar at the top of the Unity window and drag the resulting window onto the bottom of the Project window. After doing that, you'll need to drag the Hierarchy window onto the right side of the Project window so they appear side-by-side above the Console window.

Finally, click the layout dropdown (which currently shows 2 by 3) and select Save Layout... Call the layout whatever you want (we called ours Dr. T) and click the Save button. Now you can use this layout whenever you want by selecting it from the layout dropdown after you create a new project.

Before we finish, let's make sure Unity is set to use Visual Studio Community 2019 as our script editor. Select Edit > Preferences ... from the top menu bar, then click External Tools in the pane on the left. Click the dropdown box next to External Script Editor; if Visual Studio 2019 is an option, select it and close the dialog. If Visual Studio 2019 isn't an option in the dropdown, select Browse... Navigate to where your devenv.exe file for Visual Studio Community 2019 is installed and double-click it. Close the dialog.

We'll actually add and test our first C# script in Unity in Chapter 2, so you're done setting up your development environment.

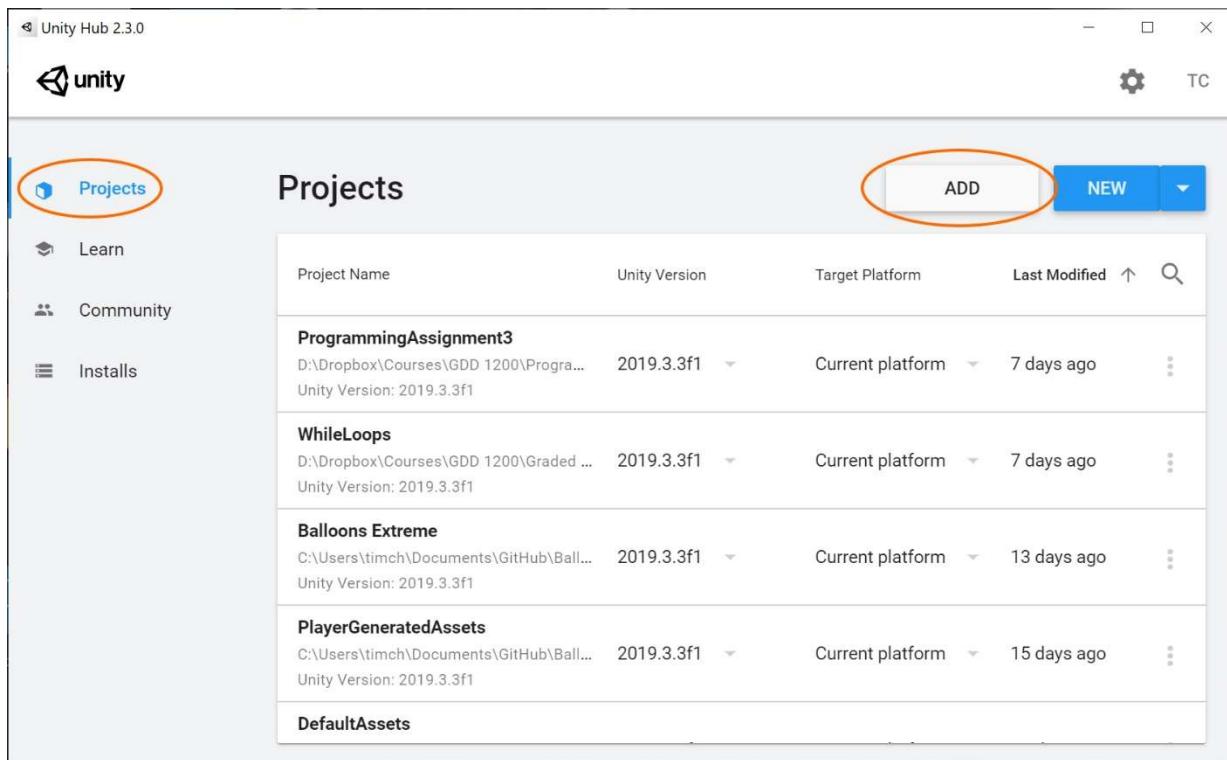
### *Opening Downloaded Unity Projects*

**Disclaimer:** Although projects built in earlier Unity versions will theoretically work in later versions, we can only guarantee that all the code provided on the Burning Teddy web site will work on the version it was created with.

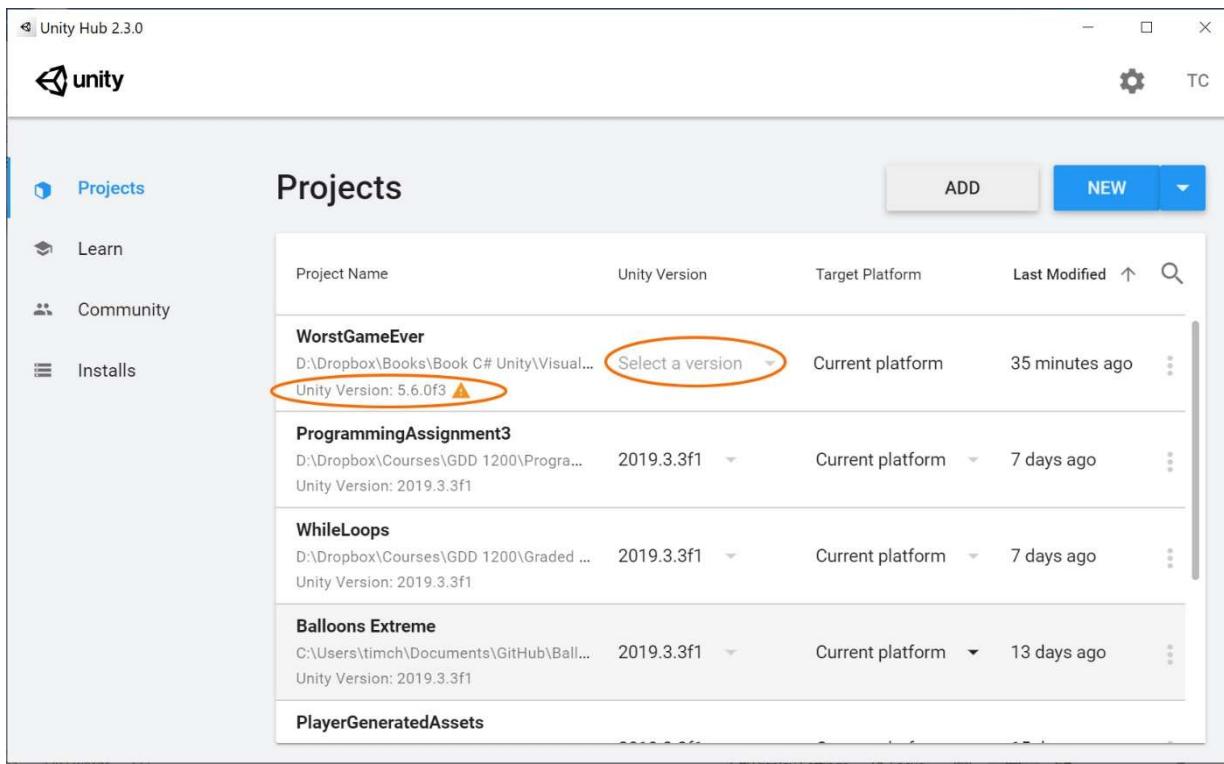
When you download a Unity project, you need to add it to your projects in Unity Hub and (probably) update the Unity version as well. Let's see how that works.

Download the Chapter 6 code from the Burning Teddy web site and extract the zip file somewhere on your computer.

In Unity Hub, select Projects on the left and click the Add button.



Navigate to the folder called WorstGameEver (it will be wherever you extracted the Chapter 6 code zip file) and click the Select Folder button. Unless you happen to have the exact same version of Unity installed as I used to create the project, your WorstGameEver project will look like the image below. Notice that the version I used to create the project is shown below the project name and Unity Hub is asking you to Select a version of Unity to use for the project.



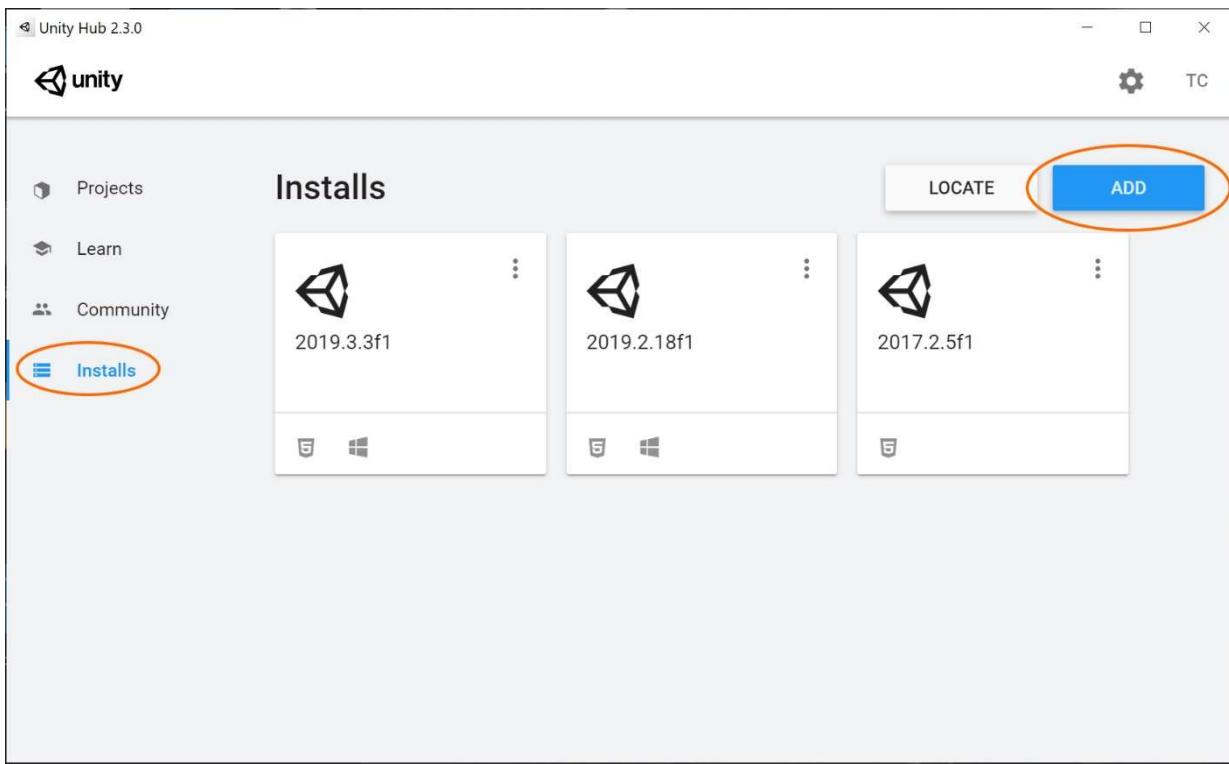
Click the down arrow next to Select a version to select the Unity version you want to use for the project; the list of versions that are shown are the Unity versions you currently have installed on your computer.

To open the project, click the project name in Unity Hub. Unity will ask you if you want to upgrade the project; you should click Confirm and Yes buttons in the dialog boxes that appear as necessary to complete the upgrade.

### *Installing New Unity Versions*

Unity releases new versions pretty regularly; you don't have to install each new version, but you'll probably want to install the latest version of Unity periodically. Here's how you do that.

Open Unity Hub, click Installs on the left and click the Add button on the upper right.



Select the Unity version you want to install from the dialog box that appears (I usually pick the first choice in the Latest Official Releases section) and click the Next button. Check the modules you want to be installed; I usually check WebGL Build Support, Windows Build Support (IL2CPP) and Documentation (to get local documentation installed). Click the Done button, confirm that Unity Hub can make changes to your computer (if necessary), and wait patiently while your selected version is installed.