# The C# Player's Guide

3 THIRD EDITION

Using C# 7.0 and Visual Studio 2017

RB Whitaker

# The C# Player's Guide

**Third Edition**

RB Whitaker

Starbound Software

RB Whitaker
rbwhitaker@outlook.com

ISBN-10: 0-9855801-3-5
ISBN-13: 978-0-9855801-3-1

# Contents at a Glance

## Part 6: Wrapping Up

# Table of Contents

# Acknowledgements

The task of writing a book is like writing software. When you start, you're sure it's only going to take a few weeks. *It'll be easy*, you think. But as you start working, you start seeing that you're going to need to make changes, and lots of them. You need to rearrange entire chapters, add topics you hadn't even thought about, and you discover that there's not even going to be a place in your book for that chapter called *Muppets of the Eastern Seaboard*.

I couldn't have ever finished this book without help. I'll start by thanking Jack Wall, Sam Hulick, Clint Mansell, and the others who wrote the music for Mass Effect. (You think I'm joking, don't you?) I listened to their music nearly incessantly as I wrote this book. Because of them, every moment of the creation of this book felt absolutely epic.

I need to also thank the many visitors to my game development tutorials site, who provided feedback on the early versions of this work. In particular, I want to thank Jonathan Loh, Thomas Allen, Daniel Bickler, and Mete ÇOM, who went way above and beyond, spending hours of their own personal time, reading through this book and provided detailed critique and corrections. With their help, this book is far more useful and valuable.

I also need to thank my mom and dad. Their confidence in me and their encouragement to always do the best I can has caused me to do things I never could have done without them.

Most of all, I want to thank my beautiful wife, who was there to lift my spirits when the weight of writing a book became unbearable, who read through my book and gave honest, thoughtful, and creative feedback and guidance, and who lovingly pressed me to keep going on this book, day after day. Without her, this book would still be a random scattering of Word documents, buried in some obscure folder on my computer, collecting green silicon-based mold.

To all of you, I owe you my sincerest gratitude.

-RB Whitaker

# Introduction

**In a Nutshell**

- Describes the goals of this book, which is to function like a player's guide, not a comprehensive cover-everything-that-ever-existed book.
- Breaks down how the book is organized from a high-level perspective, as well as pointing out some of the extra "features" of the book.
- Provides some ideas on how to get the most out of this book for programmers, beginners, and anyone who is short on time.

## The Player's Guide

This book is not about playing video games. (Though programming is as fun as playing video games for many people.) Nor is it about *making* video games, specifically. (Though you definitely can make video games with C#.)

Instead, think of this book like a player's guide, but for a programming language. A player's guide is a popular kind of book that is written to help game players:

- learn the basics of the game,
- prevent them from getting stuck,
- understand how the world they're playing in works,
- learn how to overcome the obstacles and enemies they face,
- point out common pitfalls they may face and locate useful items,
- and master the tools they're given.

This book accomplishes those same goals for the C# programming language. I'll walk you through the language from the ground up, point out places where people get stuck, provide you with hands-on examples to explore, give you quizzes to ensure you're on the right track, and describe how to use the tools that you'll need to create programs. I'll show you the ins and outs of the many features of C#, describing *why* things work the way they do, rather than just simple mechanics and syntax.

My goal is to provide you with the “dungeon map” to direct you as you begin delving into C#, while still allowing you to mostly explore whatever you want, whenever you want.

I want to point out that this book is intentionally *not* called *Everything you Need to Know about C#,* or *The Comprehensive Guide to C#*. (Note that if books with those titles actually exist, I’m not referring to them specifically, but rather, to just the general idea of an all-encompassing book.) I’m here to tell you, when you’re done with this book, you’ll still have lots to learn about C#.

But guess what? That’s going to happen with *any* book you use, including those all-encompassing books. Programming languages are complex creations, and there are enough dark corners and strange combinations that nobody can learn everything there is to know about them. In fact, I’ve even seen the people who designed the C# language say they just learned something new about it! For as long as you use C#, you’ll constantly be learning new things about it, and that’s actually one of the things that makes programming interesting.

I’ve tried to cover a lot of ground in this book, and with roughly 400 pages, anyone would expect that to be quite a bit. And it is. But there are plenty of other books out there that are 800 or even 1200 pages long. A book so heavy, you’ll need a packing mule to carry it anywhere. That, or permanently place it on the central dais of an ancient library, with a single beam of dusty light shining in on it through a hole in the marble ceiling. Instead of all that, the goal of this book is effectiveness and clarity, not comprehensiveness. Something that will fit both on your shelf and in your brain.

It is important to point out that this book is focused on the C# programming language, rather than libraries for building certain specific application types. So while you can build desktop applications, web pages, and computer games with C#, we won’t be discussing WPF, ASP.NET, DirectX, or any other platform- or framework-specific code. Instead, we’ll focus on core C# code, without bogging you down with those additional libraries at first. Once you’ve got the hang of C#, heading into one of those areas will be much easier.

## How This Book is Organized

This book is divided into six parts. Part 1 describes what you need to get going. You’ll learn how to get set up with the free software that you need to write code and make your first C# program.

Part 2 describes the basics of procedural programming—how to tell the computer, step-by-step, what to do to accomplish tasks. It covers things like how information is stored (in variables), how to make decisions, loop over things repeatedly, and put blocks of code that accomplish specific tasks into a reusable chunk called a method. It also introduces the type system of the C# language, which is one of the key pieces of C# programming.

Part 3 goes into object-oriented programming, introducing it from the ground up, but also getting into a lot of the details that make it so powerful. Chapter 20, in my opinion, is the critical point of the book. By Chapter 19, we’ve introduced all of the key concepts needed to make almost any C# program, including classes, which is the most powerful way C# provides for building your own data types. Chapter 20 contains the task (and solution) to making a simple but complete game of Tic-Tac-Toe, which will put all of the knowledge from the earlier chapters to the test. Everything we do after this chapter is simply fleshing out details and giving you better tools to get specific jobs done faster.

Part 4 covers some common programming tasks, as well as covering some of the more advanced features of C#. For the most part, these topics are independent of each other, and once you’ve made it past that critical point in Chapter 20, you should be able to do these at any time you want.

Part 5 changes gears, and covers more details about Visual Studio, which you use to create C# programs, additional information about the .NET Platform, and some tools, tricks, and information you can use as you program.

Finally, Part 6 wraps up the book with some larger scale programs for you to try making, a chapter on where to go next as you continue to learn C#, and a glossary of words that are defined throughout the book, which you can use as a reference when you run across a word or phrase that you are unfamiliar with or have forgotten about.

## Try It Out!

Scattered throughout the book are a variety of sections labeled *Try It Out!* These sections give you simple challenge problems and quizzes that give you a chance to play around with the new concepts in the chapter and test your understanding. If this were a class, these would be the homework.

The purpose of these *Try It Out!* sections is to help you get some real world practice with the new information. You can't learn to drive a car by reading the owner's manual, and you can't learn to program without writing any code.

I strongly encourage you to spend at least a few minutes doing each of these challenges to help you understand what you're reading and ensure that you've learned what you needed to.

If you have something else you want to explore with the new concepts instead of the challenges I've provided, all the better. The only thing better than playing around with this stuff is doing something with it that you have a personal interest in. If you want to explore a different direction, go for it!

At the end of the book, in Chapter 50, I have an entire chapter full of larger, tougher challenge problems for you to try out. These problems involve combining concepts from many chapters together into one program. Going through some or all of these as you're finishing up will be a great way to make sure you've learned the most important things you needed to.

The most important thing to remember about these *Try It Out!* sections is that the answers are all online. If you get stuck, or just want to compare your solution to someone else's, you can see mine at **starboundsoftware.com/books/c-sharp/try-it-out/**. I should point out that just because your solution is different from mine (or anyone else's) doesn't necessarily mean it is wrong. That's one of the best parts about programming—there's always more than one way to do something.

## In a Nutshell

At the beginning of each chapter, I summarize what it contains. These sections are designed to do the following:

- Summarize the chapter to come.
- Show enough of the chapter so that an experienced programmer can know if they already know enough to skip the chapter or if they need to study it in depth.
- Review the chapter enough to ensure that you got what you needed to from the chapter. For instance, imagine you're about to take a test on C#. You can jump from chapter to chapter, reading the *In a Nutshell* sections, and anything it describes that you didn't already know, you can then go into the chapter and review it.

## In Depth

On occasion, there are a few topics that are not critical to your understanding of C#, but they are an interesting topic that is related to the things you're learning. You'll find this information pulled out into *In Depth* sections. These are never required reading, so if you're busy, skip ahead. If you're not too busy, I think you'll find this additional information interesting, and worth taking the time to read.

### Glossary

As you go through this book, you're going to learn a ton of new words and phrases. Especially if you're completely new to programming in general. At the back of this book is a glossary that contains the definitions for these words. You can use this as a reference in case you forget what a word means, or as you run into new concepts as you learn C#.

## Getting the Most from This Book

### For Programmers

If you are a programmer, particularly one who already knows a programming language that is related to C# (C, C++, Java, Visual Basic .NET, etc.) learning C# is going to be relatively easy for you.

C# has a lot in common with all of these languages. In fact, it's fair to say that all programming languages affect and are inspired by other languages, because they evolve over time. C# looks and feels like a combination of Java and C++, both of which have roots that go back to the C programming language. Visual Basic .NET (VB.NET) on the other hand, looks and feels quite different from C# (it's based on Visual Basic, and Basic before that) but because both C# and VB.NET are designed and built for the .NET Platform, they have many of the same features, and there's almost a one-to-one correspondence between features and keywords.

Because C# is so closely tied to these other languages, and knowing that many people may already know something about these other languages, you'll see me point out how C# compares to these other languages from time to time.

If you already know a lot about programming, you're going to be able to move quickly through this book, especially the beginning, where you may find very few differences from languages you already know. To speed the process along, read the *In a Nutshell* section at the start of the chapter. If you feel like you already know everything it describes, it's probably safe to skip to the next chapter.

I want to mention a couple of chapters that might be a little dangerous to skip. Chapter 6 introduces the C# type system, including a few concepts that are key to building types throughout the book. Also, Chapter 16 is sort of a continuation on the type system, describing value and reference types. It's important to understand the topics covered in those chapters. Those chapters cover some of the fundamental ways that C# is different from these other languages, so don't skip them.

### For Busy People

One of the best parts about this book is that you don't need to read it all. Yes, that's right. It's not all mandatory reading to get started with C#. You could easily get away with only reading a part of this book, and still understand C#. In fact, not only understand it, but be able to make just about any program you can dream up. This is especially true if you already know a similar programming language.

At a minimum, you should start at the beginning and read through Chapter 20. That covers the basics of programming, all the way up to and including an introduction to making your own classes. (And if you're already a programmer, you should be able to fly through those introductory chapters quickly.)

The rest of the book could theoretically be skipped, though if you try to use someone else's code, you're probably going to be in for some surprises.

Once you've gone through those 20 chapters, you can then come back and read the rest of the book in more or less any order that you want, as you have extra time.

### For Beginners

If you've never done any programming before, be warned: learning a programming language can be hard work. The concepts in the first 20 chapters of this book are the most important to understand. Take whatever time is necessary to really feel like you understand what you're seeing in these chapters. This gets you all of the basics, and gets you up to a point where you can make your own types using classes. Like with the *For Busy People* section above, Chapter 20 is the critical point that you've got to get to, in order to really understand C#. At that point, you can probably make any program that you can think of, though the rest of the book will cover additional tools and tricks that will allow you to do this more easily and more efficiently.

After reading through these chapters, skim through the rest of the book, so that you're aware of what else C# has. That's an important step if you're a beginner. It will familiarize you with what C# has to offer, and when you either see it in someone else's code or have a need for it, you'll know exactly where to come back to. A lot of these additional details will make the most sense when you have an actual need for it in a program of your own creation. After a few weeks or a few months, when you've had a chance to make some programs on your own, come back and go through the rest of the book in depth.

## I Genuinely Want Your Feedback

Writing a book is a huge task, and no one has ever finished a huge task perfectly. There's the possibility of mistakes, plenty of chances to inadvertently leave you confused, or leaving out important details. I was tempted to keep this book safe on my hard drive, and never give it out to the world, because then those limitations wouldn't matter. But alas, my wife wouldn't let me follow Gandalf's advice and "keep it secret; keep it safe," and so now here it is in your hands.

If you ever find any problems with this book, big or small, or if you have any suggestions for improving it, I'd really like to know. After all, books are a lot like software, and there's always the opportunity for future versions that improve upon the current one. Also, if you have positive things to say about the book, I'd love to hear about that too. There's nothing quite like hearing that your hard work has helped somebody.

To give feedback of any kind, please visit **starboundsoftware.com/books/c-sharp/feedback**.

## This Book Comes with Online Content

On my website, I have a small amount of additional content that you might find useful. For starters, as people submit feedback, like I described in the last section, I will post corrections and clarifications as needed on this book's errata page: **starboundsoftware.com/books/c-sharp/errata**.

Also on my site, I will post my own answers for all of the *Try It Out!* sections found throughout this book. If you get stuck, or just want something to compare your answers with, you can visit this book's site and see a solution. To see these answers, go to: **starboundsoftware.com/books/c-sharp/try-it-out/**.

The website also contains some extra problems to work on, beyond the ones contained in this book. I've been frequently asked to add more problems to the book than what it currently has. Indeed, this version contains more than any previous version. But at the same time, most people don't actually do these problems. To avoid drowning out the actual content with more and more problems, I've provided additional problems on the website. This felt like a good compromise. These can be found at **starboundsoftware.com/books/c-sharp/additional-problems**.

Additional information or resources may be found at **starboundsoftware.com/books/c-sharp**.

# Part 1

# Getting Started

The world of C# programming lies in front of you, waiting to be explored. In Part 1 of this book, within just a few short chapters, we'll do the following:

- Get a quick introduction to what C# is (Chapter 1).
- Get set up to start making C# programs (Chapter 2).
- Write our first program (Chapter 3).
- Dig into the fundamental parts of C# programming (Chapters 3 and 4).

1

# The C# Programming Language

**In a Nutshell**

- Describes the general idea of programming, and goes into more details about why C# is a good language.
- Describes the core of what the .NET Platform is.
- Gives some history on the C# programming language for context.

I'm going to start off this book with a very brief introduction to C#. If you're already a programmer, and you've read the Wikipedia pages on C# and the .NET Framework, skip ahead to the next chapter.

On the other hand, if you're new to programming in general, or you're still a little vague on what exactly C# or the .NET Platform is, then this is the place for you.

I should point out that we'll get into a lot of detail about how the .NET Platform functions, and what it gives you as a programmer in Chapter 44. This chapter just provides a quick overview of the basics.

## What is C#?

Computers only understand binary: 1's and 0's. All of the information they keep track of is ultimately nothing more than a glorified pile of bits. All of the instructions they run and all of the data they process are binary.

But humans are notoriously bad at doing anything with a giant pile of 1's and 0's. So rather than doing that, we created programming languages, which are based on human languages (usually English) and structured in a way that allows you to give instructions to the computer. These instructions are called *source code*, and are simple text files.

When the time is right, your source code will be handed off to a special program called a *compiler*, which is able to take it and turn it into the binary 1's and 0's that the computer understands, typically in the form of an EXE file. In this sense, you can think of the compiler as a translator from your source code to the binary machine instructions that the computer knows.

There are thousands, maybe tens of thousands of programming languages, each good at certain things, and less good at other things. C# is one of the most popular. C# is a simple general-purpose programming language, meaning you can use it to create pretty much anything, including desktop applications, server-side code for websites, and even video games.

C# provides an excellent balance between ease of use and power. There are other languages that provide less power and are easier to use (like Java) and others that provide more power, giving up some of its simplicity (like C++). Because of the balance it strikes, it is the perfect language for nearly everything that you will want to do, so it's a great language to learn, whether it's your first or your tenth.

## What is the .NET Platform?

C# relies heavily on something called the .NET Platform. It is also commonly also called the .NET Framework, though we'll make a subtle distinction between the two later on. The .NET Platform is a large and powerful platform, which we'll discuss in detail in Chapter 44. You can go read it as soon as you're done with this chapter, if you want.

The .NET Platform is vast, with many components, but two stand out as the most central. The first part is the *Common Language Runtime*, often abbreviated as the *CLR*. The CLR is a software program that takes your compiled C# code and runs it. When you launch your EXE file, the CLR will start up and begin taking your code and translating it into the optimal binary instructions for the physical computer that it is running on, and your code comes to life.

In this sense, the CLR is a middle-man between your code and the physical computer. This type of program is called a virtual machine. We'll get into more of the specifics in Chapter 44. For now, it's only important to know that the .NET Platform itself, specifically the CLR runtime, play a key role in running your application—and in making it so your application can run on a wide variety of computer architectures and operating systems.

The second major component of the .NET Platform is the .NET Standard Library. The Standard Library is frequently called the Base Class Library. The Standard Library is a massive collection of code that you can reuse within your own programs to accelerate the development of whatever you are working on. We will cover some of the most important things in the Standard Library in this book, but it is huge, and deserves a book of its own. More detail on the Standard Library and the Base Class Library can be found in Chapter 44.

Built on top of the .NET Standard Library is a collection of *app models*. An app model is another large library designed for a specific type of application. This includes things like WPF and Windows Forms for GUI applications, ASP.NET for web development, and Xamarin for iOS and Android development. Game frameworks or engines like MonoGame and Unity could also be considered app models, though these are not maintained directly by Microsoft.

This book, unfortunately, doesn't cover these app models to any serious extent. There are two reasons for this. Each app model is gigantic. You could write multiple books about each of these app models (and indeed, there are many books out there). Trying to pack them all into this book would make it a 5000 page book.

Second, the app models are, true to their name, specific to a certain type of application. This means that the things that are important to somebody doing desktop development are going to be wildly different from somebody doing web development. This book focuses on the C# language itself, and the aspects of the .NET Platform that are useful to everybody. Once you've finished this book, you could then proceed on to other books that focus on specific app models. (Those books all generally assume you know C# anyway.)

We will cover how the .NET Platform is organized and how it functions in depth in Chapter 44.

## C# and .NET Versions

C# has gone through quite a bit of evolution over its history. The first release was in 2002, and established the bulk of the language features C# still has today.

A little over a year later, in 2003, C# 2.0 was released, adding in a lot of other big and powerful features, most of which will get quite a bit of attention in this book (generics, nullable types, delegates, static classes, etc.)

C# 3.0 expanded the language in a couple of very specific directions: LINQ and lambdas, both of which get their own chapters in this book.

The next two releases were somewhat smaller. C# 4.0 added dynamic typing, as well as named and optional method arguments. C# 5.0 added greatly expanded support for asynchronous programming.

In the C# 5 era, a new C# compiler was introduced: Roslyn. This compiler has a number of notable features: it's open source, it's written in C# (written in the language it's for), and it is available while your program is running (so you can compile additional code dynamically). Something about its construction also allows for people to more easily tweak and experiment with new features, which led to the features added in C# 6.0 and 7.0.

C# 6.0 and 7.0 added a whole slew of little additions and tweaks across the language. While previous updates to the language could usually be summed up in a single bullet point or two, and are given their own chapters in this book, the new features in C# 6.0 and 7.0 are small and numerous. I try to point out what these new features are throughout this book, so that you are aware of them.

Alongside the C# language itself, both Visual Studio and the Standard Library have both been evolving and growing. This book has been updated to work with Visual Studio 2017 and C# 7.0 at the time of publishing.

Future versions will, of course, arrive before long. Based on past experience, it's a safe bet that everything you learn in this book will still apply in future versions.

2

# Installing Visual Studio

**In a Nutshell**

- To program in C#, we will need a program that allows us to write C# code and run it. That program is Microsoft Visual Studio.
- A variety of versions of Visual Studio exists, including the free Community Edition, as well as several higher tiers that offer additional features at a cost.
- You do not need to spend money to make C# programs.
- This chapter walks you through the various versions of Visual Studio to help you decide which one to use, but as you are getting started, you should consider the free Visual Studio 2017 Community Edition.

To make your own programs, people usually use a program called an *Integrated Development Environment* (IDE). An IDE combines all of the tools you will commonly need to make software, including a special text editor designed for editing source code files, a compiler, and other various tools to help you manage the files in your project.

With C#, nearly everyone chooses to use some variation of Visual Studio, made by Microsoft. There are a few different levels of Visual Studio, ranging from the free Community Edition, to the high-end Enterprise Edition. In this chapter, I'll guide you through the process of determining which one to choose.

As of the time of publication of this book, the latest version is the 2017 family. There will inevitably be future releases, but the bulk of what's described in this book should still largely apply in future versions. While new features have been added over time, the fundamentals of Visual Studio have remained the same for a very long time now.

There are three main flavors of Visual Studio 2017. Our first stop will be to look at the differences among these, and I'll point out one that is most likely your best choice, getting started. (It's free, don't worry!) I'll then tell you how to download Visual Studio and a little about the installation process. By the end of this chapter, you'll be up and running, ready to start doing some C# programming!

## Versions of Visual Studio

Visual Studio 2017 comes in three editions: Community, Professional, and Enterprise. While I'm ultimately going to recommend the Community Edition (it's free, and it still allows you to make and sell commercial applications with it) it is worth briefly considering the differences between the three.

From a raw feature standpoint, Community and Professional are essentially the same thing. Enterprise comes with some nice added bonuses, but at a significantly higher cost. These extra features generally are non-code-related, but instead deal with the surrounding issues, like team collaboration, testing, performance analysis, etc. While nice, these extra features are probably not a great use of your money as you're learning, unless you work for a company or attend school at a place that will buy it for you.

Now that I've pushed you away from Enterprise as a beginner, the only remaining question is what to actually use. And to answer that, we need to compare the Community Edition to the Professional Edition.

Community and Professional are essentially the same product with a different license. Microsoft wants to make Visual Studio available to everybody, but they still want to be able to bring in money for their efforts. With the current licensing model, they've managed to do that pretty well.

While Professional costs roughly $500, Community is free. But the license prevents certain people (the ones with tons of money) from using it. While the following interpretation is not legally binding, the general interpretation of the Community license is essentially this:

You can use it to make software, both commercially and non-commercially, as long as you don't fit in one of the following categories:

- You have 5+ Visual Studio developers in your company. (If you're getting it for personal home use or moonlighting projects, you don't count the place you work for. You have 1 developer.)
- You have 250+ computers or users in your company.
- You have a gross income of $1,000,000.

If any of the above apply, you don't qualify for the Community license, and you must buy Professional. But then again, if any of those apply to you, you probably have the money to pay for Professional anyway.

There are a couple of exceptions to that:

- You're a student or educator, using it solely for educational uses.
- You're working solely on open source projects.

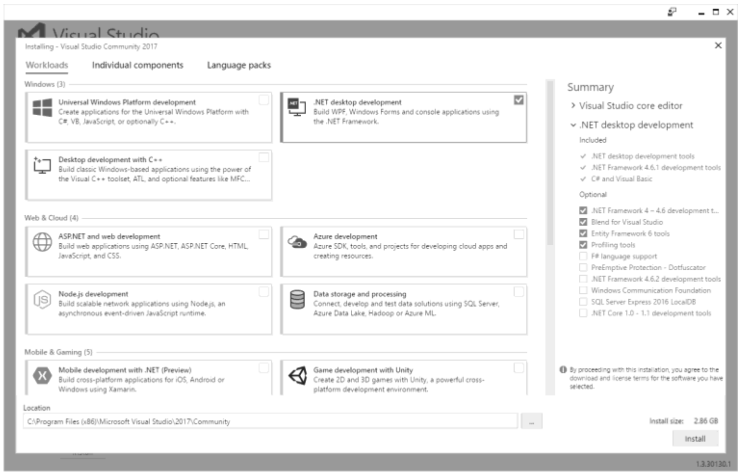
In short, for essentially everybody reading this book, you should be able to either use Community, or you're working somewhere that can afford to buy Professional or Enterprise for you.

This makes the decision simple: you will almost certainly want Visual Studio Community for now.

## The Installation Process

Visual Studio can be downloaded from **https://www.visualstudio.com/downloads**. This will actually install the Visual Studio Installer, which is a separate program from Visual Studio itself. (It sounds complicated, but the Visual Studio Installer is a powerful and useful product in its own right.)

Once you get the installer downloaded and running, you will see a screen that looks similar to this:



Installing - Visual Studio Community 2017

Workloads | Individual components | Language packs

Windows (3)

- Universal Windows Platform development — Create applications for the Universal Windows Platform with C#, VB, JavaScript, or optionally C++.
- [✓] .NET desktop development — Build WPF, Windows Forms and console applications using the .NET Framework.
- Desktop development with C++ — Build classic Windows-based applications using the power of the Visual C++ toolset, ATL, and optional features like MFC...

Web & Cloud (4)

- ASP.NET and web development — Build web applications using ASP.NET, ASP.NET Core, HTML, JavaScript, and CSS.
- Azure development — Azure SDK, tools, and projects for developing cloud apps and creating resources.
- Node.js development — Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.
- Data storage and processing — Connect, develop and test data solutions using SQL Server, Azure Data Lake, Hadoop or Azure ML.

Mobile & Gaming (5)

- Mobile development with .NET (Preview) — Build cross-platform applications for iOS, Android or Windows using Xamarin.
- Game development with Unity — Create 2D and 3D games with Unity, a powerful cross-platform development environment.

Location: C:\Program Files (x86)\Microsoft Visual Studio\2017\Community

Summary

- Visual Studio core editor
- .NET desktop development
  - Included
    - ✓ .NET desktop development tools
    - ✓ .NET Framework 4.6.1 development tools
    - ✓ C# and Visual Basic
  - Optional
    - [✓] .NET Framework 4 – 4.6 development t...
    - [✓] Blend for Visual Studio
    - [✓] Entity Framework 6 tools
    - [✓] Profiling tools
    - [ ] F# language support
    - [ ] PreEmptive Protection - Dotfuscator
    - [ ] .NET Framework 4.6.2 development tools
    - [ ] Windows Communication Foundation
    - [ ] SQL Server Express 2016 LocalDB
    - [ ] .NET Core 1.0 - 1.1 development tools

By proceeding with this installation, you agree to the download and license terms for the software you have selected.

Install size: 2.86 GB

If instead of the above, you see a screen that lists Visual Studio Community, Professional, and Enterprise, choose to install Visual Studio Community (or the option that is the one you need) and you will arrive at this screen.

Visual Studio, with every single bit of functionality, is a lumbering behemoth of a product. Starting with Visual Studio 2017, the product and the installer got a massive rewrite to allow you to install only the components you actually care about. The screen that you see in the previous image is the part of the installer that allows you to choose what components you want.

By default, nothing is checked, which would give you a very barebones Visual Studio. That's probably *not* what you want. Instead, we need to check the items that we want to include.

**For this book, you will want to check the box for *.NET desktop development* on the Workloads tab.** Feel free to look through the rest of the things and check anything else you might want to play around with at some point.

The installer contains three tabs at the top. The **Individual components** tab lets you pick and choose individual items that you might want a la carte. The **Workloads** tab will pre-select groups of items on the **Individual components** tab, to give you groups of items that are well suited for making specific types of applications. The **Language packs** tab is for choosing languages for Visual Studio. English is included by default, but check the box on other languages if you want to be able to use a different language like French or Russian.

When you have the components you want (at a minimum, **.NET desktop development**) hit the Install button and your selected components will be installed for you.

You may get an icon for Visual Studio on your desktop, but you'll also always be able to find Visual Studio in your Start Menu under "Visual Studio 2017."

Also, if you ever want to modify the components that you've installed (either to remove unused ones or add new ones) you can find "Visual Studio Installer" on your start menu as well, and simply re-run it to modify your Visual Studio settings and add or remove components.

Visual Studio will ask you to sign in with a Microsoft account at some point. If you don't have one, you can follow their instructions to make one.

Starting in the next chapter and throughout the rest of this book, we'll cover all sorts of useful tips and tricks on using Visual Studio. Towards the end of this book, in Part 5, we'll get into Visual Studio in a little more depth. Once you get through the next couple of chapters, you can jump ahead to that part whenever you're ready for it. You don't have to read through the book in order.

This book will use screenshots from Visual Studio 2017 Community. You may see some slight differences depending on which version of Visual Studio you're using and what add-ons you have active. But all of the things we talk about in this book will be available in all editions.

### Try It Out!

**Install Visual Studio.** Take the time now to choose a version of Visual Studio and install it, so that you're ready to begin making awesome programs in the next chapter.

## C# Programming on Mac and Linux

If you are interested in doing C# programming on a Mac or on Linux, you're still in luck.

Your first option is Visual Studio Code, which can be grabbed from **https://code.visualstudio.com**. Visual Studio Code is a lightweight version of Visual Studio that runs on Windows, Mac, and Linux. Visual Studio Code is missing a number of significant features that this book talks about, but it does support the basics of editing and compiling your code.

Your second option is Xamarin Studio (**http://xamarin.com/studio**), which works on macOS. Xamarin Studio is a powerful, full IDE similar to Visual Studio. In fact, Microsoft is releasing Visual Studio for Mac, but it is essentially just Xamarin Studio rebranded. (Xamarin is owned by Microsoft, so they're on the same team.)

# 3

# Hello World: Your First C# Program

**In a Nutshell**

- Start a new C# Console Application by going to **File > New > Project...**, choosing the Console Application template, and giving your project a name.
- Inside of the **Main** method, you can add code to write out stuff using a statement like **Console.WriteLine("Hello World!");**
- Compile and run your program with **F5** or **Ctrl + F5**.
- The template includes code that does the following:
    - **using** directives make it easy to access chunks of previously written code in the current program.
    - The **namespace** block puts all of the contained code into a single collection.
    - The code we actually write goes into the **Program** class in a method called **Main**, which the C# compiler recognizes as the starting point for a program.

In this chapter we'll make our very first C# program. Our first program needs to be one that simply prints out some variation of "Hello World!" or we'll make the programming gods mad. It's tradition to make your first program print out a simple message like this whenever you learn a new language. It's simple, yet still gives us enough to see the basics of how the programming language works. Also, it gives us a chance to compile and run a program, with very little chance for introducing bugs.

So that's where we'll start. We'll create a new project and add in a single line to display "Hello World!" Once we've got that, we'll compile and run it, and you'll have your very first program!

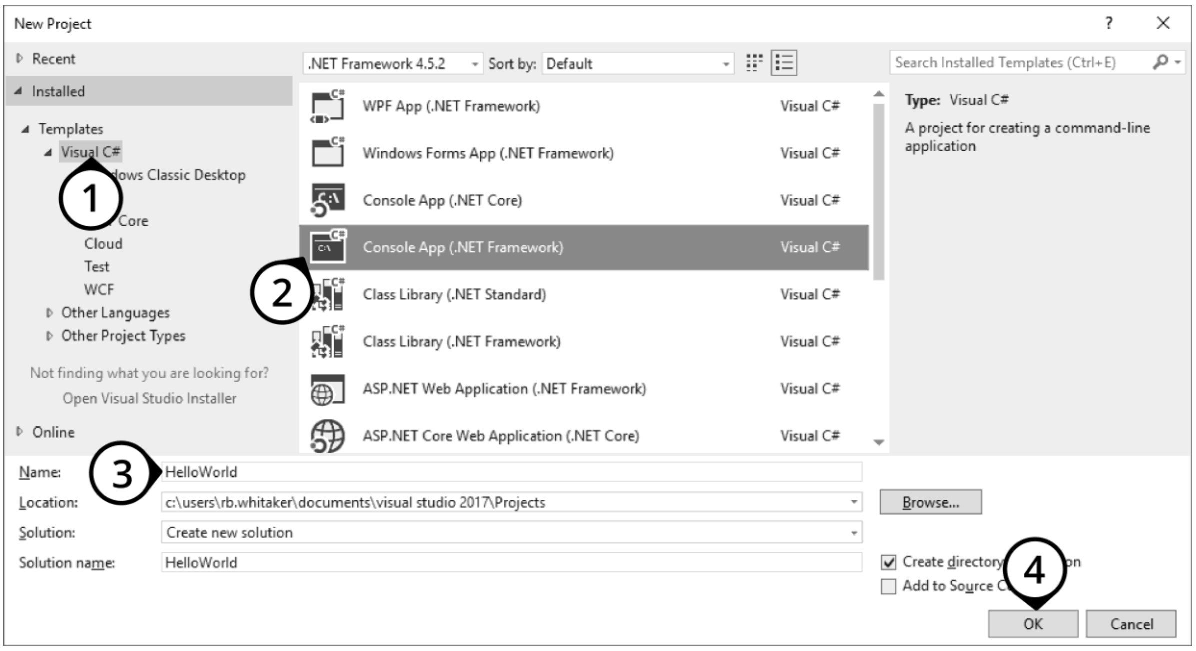
After that, we'll take a minute and look at the code that you have written in more detail before moving on to more difficult, but infinitely more awesome stuff in the future!

## Creating a New Project

Let's get started with our first C# program! Open up Visual Studio, which we installed in Chapter 2.

When the program first opens, you will see the Start Page come up. To create a new project, select **File > New > Project…** from the menu bar. (Note you can also search for the Console Application template on the Start Page directly.)

Once you have done this, a dialog will appear asking you to specify a project type and a name for the project. This dialog is shown below:


New Project
Recent
Installed
Templates
Visual C#
Windows Classic Desktop
.NET Core
Cloud
Test
WCF
Other Languages
Other Project Types
Not finding what you are looking for?
Open Visual Studio Installer
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
WPF App (.NET Framework) Visual C#
Windows Forms App (.NET Framework) Visual C#
Console App (.NET Core) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Standard) Visual C#
Class Library (.NET Framework) Visual C#
ASP.NET Web Application (.NET Framework) Visual C#
ASP.NET Core Web Application (.NET Core) Visual C#
Type: Visual C#
A project for creating a command-line application
Browse...
Create directory for solution
Add to Source Control
OK
Cancel
1
2
3
4

Name: HelloWorld
Location: c:\users\rb.whitaker\documents\visual studio 2017\Projects
Solution: Create new solution
Solution name: HelloWorld

On the left side, you will see a few categories of templates to choose from. Depending on what version of Visual Studio you have installed and what plugins and extensions you have, you may see different categories here, but the one you'll want to select is the Visual C# category, which will list all C#-related templates that are installed.

Once that is selected, in the list in the top-center, find and select the **Console Application (.NET Framework)** template. The Console Application template is the simplest template and it is exactly where we want to start. For all of the stuff we will be doing in this book, this is the template to use.

As you finish up this book, if you want to start doing things like making programs with a graphical user interface (GUI), game development, smart phone app development, or web-based development, you will be able to put these other templates to good use.

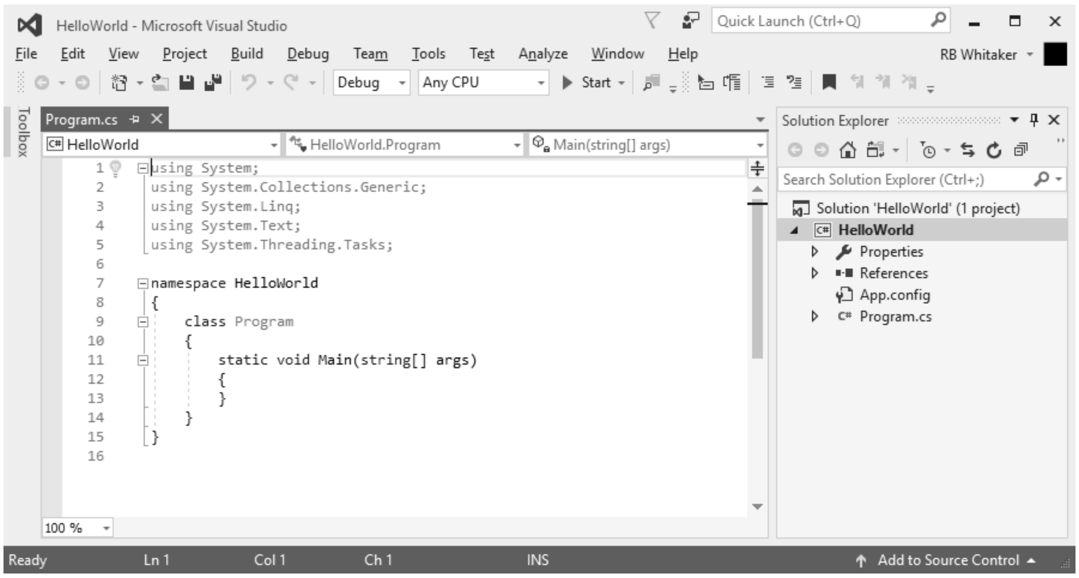
At the bottom of the dialog, type in a name for your project. I've called mine "HelloWorld." Your project will be saved in a directory with this name. It doesn't matter what you call a project, but a good name will help you find it later. By default, Visual Studio tries to call your programs "ConsoleApplication1" or "ConsoleApplication2." If you don't choose a good name, you won't know what each of these do. By default, projects are saved under your Documents directory (Documents/Visual Studio 2017/Projects/).

Finally, press the **OK** button to create your project!

## A Brief Tour of Visual Studio

Once your project has loaded, it is worth a brief discussion of what you see before you. We'll look in depth at how Visual Studio works later on (Chapter 45) but it is worth a brief discussion right now.

By this point, you should be looking at a screen that looks something like this:



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

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Depending on which version of Visual Studio you installed, you may see some slight differences, but it should look pretty similar to this.

In the center should be some text that starts out with **using System;**. This is your program's source code! It is what you'll be working on. We'll discuss what it means, and how to modify it in a second. We'll spend most of our time in this window.

On the right side is the Solution Explorer. This shows you a big outline of all of the files contained in your project, including the main one that we'll be working with, called "Program.cs". The *.cs file extension means it is a text file that contains C# code. If you double-click on any item in the Solution Explorer, it will open in the main editor window. The Solution Explorer is quite important, and we'll use it frequently.

As you work on your project, other windows may pop up as they are needed. Each of these can be closed by clicking on the 'X' in the upper right corner of the window.

If, by chance, you are missing a window that you feel you want, you can always open it by finding it on either the **View** menu or **View > Other Windows**. For right now, if you have the main editor window open with your Program.cs file in it, and the Solution Explorer, you should be good to go.

## Building Blocks: Projects, Solutions, and Assemblies

As we get started, it is worth defining a few important terms that you'll be seeing throughout this book. In the world of C#, you'll commonly see the words *solution*, *project*, and *assembly*, and it is worth taking the time upfront to explain what they are, so that you aren't lost.

These three words describe the code that you're building in different ways. We'll start with a project. A *project* is simply a collection of source code and resource files that will all eventually get built into the same executable program. A project also has additional information telling the compiler how to build it.

When compiled, a project becomes an *assembly*. In nearly all cases, a single project will become a single assembly. An assembly shows up in the form of an EXE file or a DLL file. These two different extensions represent two different types of assemblies, and are built from two different types of projects (chosen in the project's settings).

A *process assembly* appears as an EXE file. It is a complete program, and has a starting point defined, which the computer knows to run when you start up the EXE file. A *library assembly* appears as a DLL file. A DLL file does not have a specific starting point defined. Instead, it contains code that other programs can access and reuse on the fly.

Throughout this book, we'll be primarily creating and working with projects that are set up to be process assemblies that compile to EXE files, but you can configure any project to be built as a library assembly (DLL) instead.

Finally, a *solution* will combine multiple projects together to accomplish a complete task or form a complete program. Solutions will also contain information about how the different projects should be connected to each other.  While solutions can contain many projects, most simple programs (including nearly everything we do in this book) will only need one. Even many large programs can get away with only a single project.

Looking back at what we learned in the last section about the Solution Explorer, you'll see that the Solution Explorer is showing our entire solution as the very top item, which it labels "Solution 'HelloWorld' (1 project)." Immediately underneath that, we see the one project that our solution contains: "HelloWorld." Inside of the project are all of the settings and files that our project has, including the Program.cs file that contains source code that we'll soon start editing.

It's important to keep the solution and project separated in your head. They both have the same name and it can be a little confusing. Just remember the top node is the solution, and the one inside it is the project.

## Modifying Your Project

We're now ready to make our program actually do something. In the center of your Visual Studio window, you should see the main text editor, containing text that should look identical to this:

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

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

In a minute we'll discuss what all of that does, but for now let's go ahead and make our first change—adding something that will print out the message "Hello World!"

Right in the middle of that code, you'll see three lines that say **static void Main(string[] args)** then a starting curly brace ('{') and a closing curly brace ('}'). We want to add our new code right between the two curly braces.

Here's the line we want to add:

```
Console.WriteLine("Hello World!");
```

So now our program's full code should look like this:

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

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

We've completed our first C# program! Easy, huh?

**Try It Out!**

**Hello World!** It's impossible to understate how important it is to actually *do* the stuff outlined in this chapter. Simply reading text just doesn't cut it. In future chapters, most of these *Try It Out!* sections will contain extra things to do, beyond the things described in the actual body of the chapter. But for right now, it is very important that you simply go through the process explained in this chapter. The chapter itself is a *Try It Out!* So follow through this chapter, one step at a time, and make sure you're understanding the concepts that come up, at least at a basic level.

## Compiling and Running Your Project

Your computer doesn't magically understand what you've written. Instead, it understands special instructions that are composed of 1's and 0's called *binary*. Fortunately for us, Visual Studio includes a thing called a *compiler*. A compiler will take the C# code that we've written and turn it into binary that the computer understands.

So our next step is to compile our code and run it. Visual Studio will make this easy for us.

To start this process, press **F5** or choose **Debug > Start Debugging** from the menu.

There! Did you see it? Your program flashed on the screen for a split second! (Hang on… we'll fix that in a second. Stick with me for a moment.)

We just ran our program in debug mode, which means that if something bad happens while your program is running, it won't simply crash. Instead, Visual Studio will notice the problem, stop in the middle of what's going on, and show you the problem that you are having, allowing you to debug it. We'll talk more about how to actually debug your code in Chapter 48.

So there you have it! You've made a program, compiled it, and executed it!

If it doesn't compile and execute, double check to make sure your code looks like the code above.

### Help! My program is running, but disappearing before I can see it!

You likely just ran into this problem when you executed your program. You push **F5** and the program runs, a little black console window pops up for a split second before disappearing again, and you have no clue what happened.

There's a good reason for that. Your program ran out of things to do, so it finished and closed on its own. (It thinks it's so smart, closing on its own like that.)

But we're really going to want a way to make it so that *doesn't* happen. After all, we're left wondering if it even did what we told it to. There are two solutions to this, each of which has its own strengths and weaknesses.

**Approach #1:** When you run it *without* debugging, console programs like this will always pause before closing. So one option is to run it without debugging. This option is called Release Mode. We'll cover this in a little more depth later on, but the bottom line is that your program runs in a streamlined mode which is faster, but if something bad happens, your program will just die, without giving you a chance to debug it.

You can run in release mode by simply pressing **Ctrl + F5** (instead of just **F5**). Do this now, and you'll see that it prints out your "Hello World!" message, plus another message that says "Press any key to continue…" which does exactly what it says and waits for you before closing the program. You can also find this under **Debug > Start Without Debugging** on the menu.

But there's a distinct disadvantage to running in release mode. We're no longer running in debug mode, and so if something happens with your program while it is running, your application will crash and die. (Hey, just like all of the other "cool" programs out there!) Which brings us to an alternative approach:

**Approach #2:** Put another line of code in that makes the program wait before closing the program. You can do this by simply adding in the following line of code, right below where you put the **Console.WriteLine("Hello World!");** statement:

```
Console.ReadKey();
```

So your full code, if you use this approach, would look like this:

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

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadKey();
        }
    }
}
```

Using this approach, there is one more line of code that you have to add to your program (in fact, every console application you make), which can be a little annoying. But at least with this approach, you can still run your program in debug mode, which you will soon discover is a really nice feature.

Fortunately, this is only going to be a problem with console apps. That's what we'll be doing in this book, but before long, you'll probably be making windows apps, games, or awesome C#-based websites, and this problem will go away on its own. They work in a different way, and this won't be an issue there.

## Try It Out!

**See Your Program Twice.** I've described two approaches for actually seeing your program execute. Take a moment and try out each approach. This will give you an idea of how these two different approaches work. Also, try combining the two and see what you get. Can you figure out why you need to push a key twice to end the program?

## A Closer Look at Your Program

Now that we've got our program running, let's take a minute and look at each line of code in the program we've made. I'll explain what each one does so that you'll have a basic understanding of everything in your simple Hello World program.

### Using Directives

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

The first few lines of your program all start with the keyword **using**. A *keyword* is simply a reserved word, or a magic word that is a built-in part of the C# programming language. It has special meaning to the C# compiler, which it uses to do something special. The **using** keyword tells the compiler that there is a whole other pile of code that someone made that we want to be able to access. (This is actually a bit of a simplification, and we'll sort out the details in Chapter 27.)

So when you see a statement like **using System;** you know that there is a whole pile of code out there named *System* that our code wants to use. Without this line, the C# compiler won't know where to find things and it won't be able to run your program. You can see that there are five **using** directives in your little program that are added by default. We can leave these exactly the way they are for the near future.

### Namespaces, Classes, and Methods

Below the **using** directives, you'll see a collection of curly braces ('{' and '}') and you'll see the keywords **namespace**, **class**, and in the middle, the word **Main**. Namespaces, classes, and methods (which **Main** is an example of) are ways of grouping related code together at various levels. Namespaces are the largest grouping, classes are smaller, and methods are the smallest. We'll discuss each of these in great depth as we go through this book, but it is worth a brief introduction now. We'll start at the smallest and work our way up.

*Methods* are a way of consolidating a single task together in a reusable block of code. In other programming languages, methods are sometimes called functions, procedures, or subroutines. We'll get into a lot of detail about how to make and use methods as we go, but the bulk of our discussion about methods will be in Chapter 15, with some extra details in Chapter 28.

Right in the middle of the generated code, you'll see the following:

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

This is a method, which happens to have the name **Main**. I won't get into the details about what everything else on that line does yet, but I want to point out that this particular setup for a method makes it so that C# knows it can be used as the starting point for your program. Since this is where our program starts, the computer will run any code we put in here. For the next few chapters, everything we do will be right in here.

You'll also notice that there are quite a few curly braces in our code. Curly braces mark the start and end of code blocks. Every starting curly brace ('{') will have a matching ending curly brace ('}') later on. In this particular part, the curly braces mark the start and end of the **Main** method. As we discuss classes and namespaces, you'll see that they also use curly braces to mark the points where they begin and end. From looking at the code, you can probably already see that these code blocks can contain other code blocks to form a hierarchy.

When one thing is contained in another, it is said to be a *member* of it. So the **Program** class is a member of the namespace, and the **Main** method is a member of the **Program** class.

*Classes* are a way of grouping together a set of data and methods that operate on that data into a single reusable package. Classes are the fundamental building block of object-oriented programming. We'll get into this in great detail in Part 3, especially Chapters 17 and 18.

In the generated code, you can see the beginning of the class, marked with:

```
class Program
{
```

And later on, after the **Main** method it contains, you'll see a matching closing curly brace:

```
}
```

**Program** is simply a name for the class. It could have been just about anything else. The fact that the **Main** method is contained in the **Program** class indicates that it belongs to the **Program** class.

Namespaces are the highest level grouping of code. Many smaller programs may only have a single namespace, while larger ones often divide the code into several namespaces based on the feature or component that the code is used in. We'll spend a little extra time detailing namespaces and **using** directives in Chapter 27.

Looking at the generated code, you'll see that our **Program** class is contained in a namespace called "HelloWorld":

```
namespace HelloWorld
{
    ...
}
```

Once again, the fact that the **Program** class appears within the **HelloWorld** namespace means that it belongs to that namespace, or is a member of it.

## Whitespace Doesn't Matter

In C#, whitespace such as spaces, new lines, and tabs don't matter to the C# compiler. This means that technically, you could write any program on a single line! But don't do that. That would be a bad idea.

Instead, you should use whitespace to help make your code more readable, both for other people who may look at your code, or even yourself a few weeks from now, when you've forgotten what exactly your code was supposed to do.

I'll leave the decision about where to put whitespace up to you, but as an example, compare the following pieces of code that do the same thing:

```
static void Main(string
[] args) { Console
.WriteLine                                                     (
                "Hello World!"                 );}
```

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
}
```

For the sake of clarity, I'll use a style like the bottom version throughout this book.

## Semicolons

You may have noticed that the lines of code we added all ended with semicolons (‘;’).

This is how C# knows it has reached the end of a statement. A *statement* is a single step or instruction that does something. We’ll be using semicolons all over the place as we write C# code.

### Try It Out!

**Evil Computers.** In the influential movie *2001: A Space Odyssey*, an evil computer named HAL 9000 takes over a Jupiter-bound spaceship, locking Dave, the movie’s hero, out in space. As Dave tries to get back in, to the ship, he tells HAL to open the pod bay doors. HAL’s response is "I’m sorry, Dave. I’m afraid I can’t do that." Since we know not all computers are friendly and happy to help people, modify your Hello World program to say HAL 9000’s famous words, instead of "Hello World!"

This chapter may have seemed long, and we haven’t even accomplished very much. That’s OK, though. We have to start somewhere, and this is where everyone starts. We have now made our first C# program, compiled it, and executed it! And just as important, we now have a basic understanding of the starter code that was generated for us. This really gets us off on the right foot. We’re off to a great start, but there’s so much more to learn!

# 4 Comments

**Quick Start**

- Comments are a way for you to add text for other people (and yourself) to read. Computers ignore comments entirely.
- Comments are made by putting two slashes (**//**) in front of the text.
- Multi-line comments can also be made by surrounding it with asterisks and slashes, like this: **/* this is a comment */**

In this short chapter we'll cover the basics of comments. We'll look at what they are, why you should use them, and how to do them. Many programmers (even many C# books) de-emphasize comments, or completely ignore them. I've decided to put them front and center, right at the beginning of the book—they really are that important.

## What is a Comment?

At its core, a *comment* is text that is put somewhere for a human to read. Comments are ignored entirely by the computer.

## Why Should I Use Comments?

I mentioned in the last chapter that whitespace should be used to help make your code more readable. Writing readable and understandable code is a running theme you'll see in this book. Writing code is actually far easier than reading it, or trying understanding what it does. And believe it or not, you'll actually spend far more time reading code than writing it. You will want to do whatever you can to make your code easier to read. Comments will go a very long way towards making your code more readable and understandable.

You should use comments to describe what you are doing so that when you come back to a piece of code that you wrote after several months (or even just days) you'll know what you were doing.

Writing comments—wait, let me clarify—writing *good* comments is a key part of writing good code. Comments can be used to explain tricky sections of code, or explain what things are supposed to do. They are a primary way for a programmer to communicate with another programmer who is looking at their code. The other programmer may even be on the other side of the world and working for a different company five years later!

Comments can explain what you are doing, as well as why you are doing it. This helps other programmers, including yourself, know what was going on in your mind at the time.

In fact, even if you know you're the only person who will ever see your code, you should still put comments in it. Do you remember what you ate for lunch a week ago today? Neither do I. Do you really think that you'll remember what your code was supposed to do a week after you write it?

Writing comments makes it so that you can quickly understand and remember what the code does, how it does it, why it does it, and you can even document why you did it one way and not another.

## How to Make Comments in C#

There are three basic ways to make comments in C#. For now, we'll only really consider two of them, because the third applies only to things that we haven't looked at yet. We'll look at the third form of making comments in Chapter 15.

The first way to create a comment is to start a line with two slashes: **//**. Anything on the line following the two slashes will be ignored by the computer. In Visual Studio the comments change color—green, by default—to indicate that the rest of the line is a comment.

Below is an example of a comment:

```
// This is a comment, where I can describe what happens next...
Console.WriteLine("Hello World!");
```

Using this same thing, you can also start a comment at the end of a line of code, which will make it so the text after the slashes are ignored:

```
Console.WriteLine("Hello World!"); // This is also a comment.
```

A second method for creating comments is to use the slash and asterisk combined, surrounding the comment, like this:

```
Console.WriteLine("Hi!"); /* This is a comment that ends here... */
```

This can be used to make multi-line comments like this:

```
/* This is a multi-line comment.
    It spans multiple lines.
    Isn't it neat? */
```

Of course, you can do multi-line comments with the two slashes as well, it just has to be done like this:

```
//  This is a multi-line comment.
//  It spans multiple lines.
//  Isn't it neat?
```

In fact, most C# programmers will probably encourage you to use the single line comment version instead of the **/* */** version, though it is up to you.

The third method for creating comments is called XML Documentation Comments, which we'll discuss later, because they're used for things that we haven't discussed yet. For more information about XML Documentation Comments, see Chapter 15.

## How to Make Good Comments

Commenting your code is easy; making *good* comments is a little trickier. I want to take some time and describe some basic principles to help you make comments that will be more effective.

My first rule for making good comments is to write the comments for a particular chunk of code as soon as you've got the piece more or less complete. A few days or a weekend away from the code and you may no longer really remember what you were doing with it. (Trust me, it happens!)

Second, write comments that add value to the code. Here's an example of a bad comment:

```
// Uses Console.WriteLine to print "Hello World!"
Console.WriteLine("Hello World!");
```

The code itself already says all of that. You might as well not even add it. Here's a better version:

```
// Printing "Hello World!" is a very common first program to make.
Console.WriteLine("Hello World!");
```

This helps to explain *why* we did this instead of something else.

Third, you don't need a comment for every single line of code, but it is helpful to have one for every section of related code. It's possible to have too many comments, but the dangers of over-commenting code matter a whole lot less than the dangers of under-commented (or *completely* uncommented code).

When you write comments, take the time put in anything that you or another programmer may want to know if they come back and look at the code later. This may include a human-readable description of what is happening, it may include describing the general method (or *algorithm*) you're using to accomplish a particular task, or it may explain why you're doing something. You may also find times where it will be useful to include why you aren't using a different approach, or to warn another programmer (or yourself!) that a particular chunk of code is tricky, and you shouldn't mess with it unless you really know what you're doing.

Having said all of this, don't take it to an extreme. Good comments don't make up for sloppy, ugly, or hard to read code. Meanwhile nice, clean, understandable code reduces the times that you need comments at all. (The code is the authority on what's happening, not the comments, after all.) Make the code as readable as possible first, then add just enough comments to fill in the gaps and paint the bigger picture.

When used appropriately, comments can be a programmer's best friend.

### Try It Out!

**Comment *ALL* the things!** While it's overkill, in the name of putting together everything we've learned so far, go back to your Hello World program from the last chapter and add in comments for each part of the code, describing what each piece is for. This will be a good review of what the pieces of that simple program do, as well as give you a chance to play around with some comments. Try out both ways of making comments (**//** and **/* */**) to see what you like.

# Part 2

# The Basics

With a basic understanding of how to get started behind us, we're ready to dig in and look at the fundamentals of programming in C#.

It is in this part that our adventure really gets underway. We'll start learning about the world of C# programming, and learn about the key tools that we'll use to get things done.

In this section, we cover aspects of C# programming that are called "procedural programming." This means we'll be learning how to tell the computer, step-by-step, how to get things done.

We'll look at how to:

- Store data in variables (Chapter 5).
- Understand the type system (Chapter 6).
- Do basic math (Chapters 7 and 9).
- Get input from the user (Chapter 8).
- Make decisions (Chapter 10).
- Repeat things multiple times (Chapter 12 and 13).
- Create enumerations (Chapter 14).
- Package related code together in a way that allows you to reuse it (Chapter 15).

5

# Variables

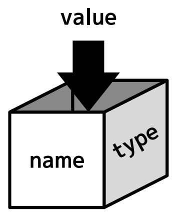
**In a Nutshell**

- You can think of variables as boxes that store information.
- A variable has a name, a type, and a value that is contained in it.
- You declare (create) a variable by stating the type and name: **int number;**
- You can assign values to a variable with the assignment operator ('='): **number = 14;**
- When you declare a variable, you can initialize it as well: **int number = 14;**
- You can retrieve the value of a variable simply by using the variable's name in your code: **Console.WriteLine(number);**
- This chapter also gives guidelines for good variable names.

In this chapter, we're going to dig straight into one of the most important parts of programming in C#. We're going to discuss variables, which are how we keep track of information in our programs. We'll look at how you create them, place different values in them, and use the value that is currently in a variable.

## What is a Variable?

A key part of any program you make, regardless of the programming language, is the ability to store information in memory while the program is running. For example, you might want to store a player's score or a person's name, so that you can refer back to it later or modify it.



You may remember discussing variables in math classes, but these are a little different. In math, we talk about variables being an "unknown quantity" that you are supposed to solve for. Variables in math are a specific value that you just need to figure out.

In programming, a *variable* is a place in memory where you can store information. It's like a little box or bucket to put stuff in. At any point in time, you can look up the contents of the variable or rewrite the contents of the variable with new stuff. When you do this, the variable itself doesn't need to change, just the contents in the box.

Each variable has a name and a type. The name is what you'll use in your program when you want to read its contents or put new stuff in it.

The variable's *type* indicates what kind of information you can put in it. C# has a large assortment of types that you can use, including a variety of integer types, floating point (real valued) types, characters, strings (text), Boolean (true/false), and a whole lot more.

In C#, types are a really big deal. Throughout this book, we'll spend a lot of time learning how to work with different types, converting from one type to another, and ultimately building our own types from scratch.

In the next chapter, we'll get into types in great detail. For now though, let's look at the basics of creating a variable.

## Creating Variables

Let's make our first variable. The process of creating a variable is called *declaring* a variable.

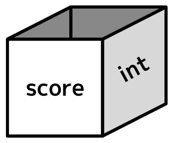
Let's start by going to Visual Studio and creating a brand new console project, just like we did with the Hello World project, back in Chapter 3. Inside of the **Main** method, add the following single line of code:

```
int score;
```

So your code should look something like this:

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

namespace Variables
{
    class Program
    {
        static void Main(string[] args)
        {
            int score;
        }
    }
}
```

Congratulations! You've made your first variable! When you declare a variable, the computer knows that it will need to reserve a place in memory for this variable.



As you can see, when you declare a variable, you need to indicate the variable's name and type. This one line has both of those parts on it. The first part you see here is **int**. This is the variable's type. We'll look at the different types that are available in a minute, but for now all we need to know is that the **int** type is for storing integers. (In case you don't remember from math class, integers are whole numbers and their negatives, so 0, 1, 2, 3, 4, ..., and -1, -2, -3, -4, ....) Because we've made a variable that stores integers, we know we could put the number 100 in it, or -75946. But we could *not* store the number 1.3483 (it's not an integer), and we also could not store a word like "hamburger" (it's not an integer either). The variable's type determines what kind of stuff we can put in it.

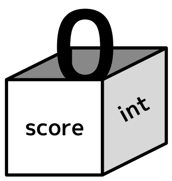
The second part of declaring a variable is giving it a name. It is important to remember that a variable's name is meant for humans. The computer doesn't care what it is called. (In fact, once you hand it off to the computer, it changes the name to a memory location anyway.) So you want to choose a name that makes sense to humans, and accurately describes what you're putting in it. In math, we often call variables by a single letter (like x), but in programming we can be more precise and call it something like **score** instead.

As always, C# statements end with a ';', telling the computer that it has reached the end of the statement. After this line, we have made a new variable with the name **score** and a type of **int** which we can now use!

## Assigning Values to Variables

The next thing we want to do is put a value in the variable. This is called *assigning* a value to the variable, and it is done using the assignment operator: "**=**". The line of code below assigns the value 0 to the score variable we just created:

```
score = 0;
```



You can add this line of code right below the previous line we added.

This use of the equals sign is different than how it is used in math. In math, "=" indicates that two things are the same, even though they may be written in different formats. In C# and many other programming languages, it means we're going to take the stuff on the right side of the equals sign and place it in the variable that is named on the left.

You can assign any integer value to **score**, and you can assign different values over time:

```
score = 4;
score = 11;
score = -1564;
```

You can assign a value to a variable whenever you want, as long as it is after the variable has been declared. Of course, we haven't learned very powerful tools for programming yet, so "whenever you want" doesn't mean much yet. (We'll get there soon, don't worry!)

When we create a variable, we often want to give it a value right away. (The C# compiler is not a big fan of you trying to see what's inside an empty variable box.) While you can declare a variable and assign it a value in separate steps, it is also possible to do both of them at the same time:

```
int theMeaningOfLife = 42;
```

This line creates a variable called **theMeaningOfLife** with the type **int**, and gives it a starting value of 42.

## Retrieving the Contents of a Variable

As we just saw, we can use the assignment operator ('=') to put values into a variable. You can also see and use the contents of a variable, simply by using the variable's name. When the computer is running your code and it encounters a variable name, it will go to the variable, look up the contents inside, and use that value in its place.

For example, int the code below, the computer will pull the number out of the **number** variable and write 3 to the console window:

```
int number = 3;
Console.WriteLine(number); // Console.WriteLine prints lots of things, not just text.
```

When you access a variable, here's what the computer does:

1. Locates the variable that you asked for in memory.
2. Looks in the contents of the variable to see what value it contains.
3. Makes a *copy* of that value to use where it is needed.

The fact that it grabs a copy of the variable is important. For one, it means the variable keeps the value it had. Reading from a variable doesn't change the value of the variable. Two, whatever you do with the copy won't affect the original. (We'll learn more about how this works in Chapter 16, when we learn about value and reference types.)

For example, here is some code that creates two variables and assigns the value of one to the other:

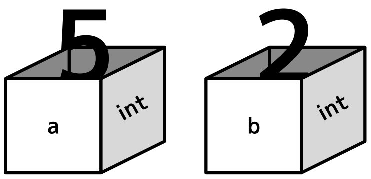
```
int a = 5;
int b = 2;

b = a;
a = -3;
```

With what you've learned so far about variables, what value will **a** and **b** have after this code runs?

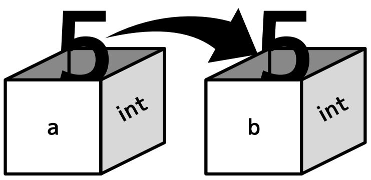
**Try It Out!**

**Playing with Variables.** Take the little piece of code above and make a program out of it. Follow the same steps you did in Chapter 3 when we made the Hello World program, but instead of adding code to print out "Hello World!", add the lines above. Use **Console.WriteLine**, like we did before and print out the contents of the two variables. Before you run the code, think about what you expect to be printed out for the **a** and **b** variables. Go ahead and run the program. Is it what you expected?

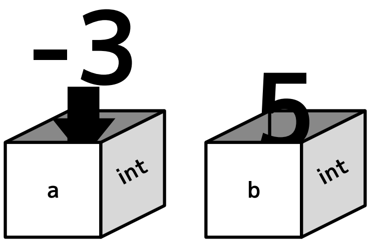
Right at the beginning of those four lines, we create two variables, one named **a**, and one named **b**. Both can store integers, because we're using the **int** type. We also assign the value 5 to **a**, and 2 to **b**. After the first two lines, this is what we're looking at:



We then use the assignment operator to take the value inside of **a** and copy it to **b**:



Finally, on the last line we assign a completely new value to **a**:



If we printed out **a** and **b**, we would see that **a** is -3 and **b** is 5 by the time this code is finished.

## How Data is Stored

Before we move into a discussion about the C# type system, we need to understand a little about how information is stored on a computer. This is a key part of what drives the need for types in the first place.

It is important to remember that computers only work with 1's and 0's. (Technically, they're tiny electric pulses or magnetic fields that can be in one of two states which we label 1 and 0.)

A single 1 or 0 is called a *bit*, and a grouping of eight of them is called a *byte*. If we do the math, this means that a single byte can store up to a total of 256 different states.

To get more states than this, we need to put multiple bytes together. For instance, two bytes can store 65,536 possible states. Four bytes can store over 4 billion states, and eight bytes combined can store over 18 quintillion possible states.

But we need a way to take all of these states and make sense out of them. This is what the type system is for. It defines how many bytes we need to store different things, and how those bits and bytes will be interpreted by the computer, and ultimately, the user.

For example, let's say we want to store letters. Modern systems tend to use two bytes to store letters. Programmers have assigned each letter a specific pattern of bits. For instance, we assign the capital letter 'A' to the bit pattern 00000000 01000001. 'B' is one up from that: 00000000 01000010. Because we're using two bytes, we have 65,536 different possibilities. That's enough to store every symbol in every language that is currently spoken on Earth, including many ancient languages, and still have room to spare.

For each different type of data, we interpret the underlying bits and bytes in a different way. The **int** type that we were using earlier works like this. The **int** type uses four bytes. For brevity, in this discussion, I'm leaving off the first three bytes, which contain all zeros for the small sample numbers we're using here. The value 0 is represented with the bit pattern 00000000. The value 1 is represented with the bit pattern 00000001. 2 is represented with 00000010. 3 is 00000011. This is basically counting in a base two numbering system, or a binary numbering system.

Other types will use their bits and bytes in other ways. We won't get into the specifics about how they all work, as that's really beyond the scope of this book. I'll just point out that the way C# interprets bits and bytes uses the same standard representations as nearly every other language and computer.

## Multiple Declarations and Assignments

Our earlier code for creating a variable and for assigning a value to a variable just did one per line. But you can declare multiple variables at the same time using code like this:

```
int a, b, c;
```

If you do this, all variables must be of the same type (**int** in this case). We'll talk about types in more depth in Chapter 6.

You can also assign the same value to multiple different variables all at the same time:

```
a = b = c = 10;
```

Most cases will probably lead you to make and assign values individually, rather than simultaneously, but it is worth knowing that this is an option.

## Good Variable Names

Before we go on, let's talk about how to choose good names for your variables. Not everybody agrees on what makes a variable name good. But I'm going to give you the rules I follow, which you'll discover are pretty typical, and not too far off from what most experienced programmers do.

The purpose of a variable name is to give a human-readable label for the variable. Anyone who stumbles across the variable name should be able to instantly know what information the variable contains.

It's easy to write code. It's hard to write code that you can actually go back and read and understand. Like comments, good variable names are an absolutely critical part of writing readable code, and it's not something that can be ignored. Here are my rules:

**Rule #1: Meet C#'s Requirements.** C# has a few requirements for variable names. All variable names have to start with a letter (a-z or A-Z) or the underscore ('_') character, and can then contain any number of other letters, numbers, or the underscore character. You also cannot name a variable the same thing as one of the reserved keywords that C# defines. These keywords are highlighted in blue in Visual Studio, but includes things like **namespace**, **int**, and **public**. Your code won't compile if you don't follow this rule.

**Rule #2: Variable names should describe the stuff you intend on putting in it.** If you are putting a player's score in it, call it **score**, or **playerScore**, or even **plrscr** if you have to, but don't call it **jambalaya**, **p**, or **monkey**. But speaking of **plrscr**…

**Rule #3: Don't abbreviate or remove letters.** Looking at the example of **plrscr**, you can tell that it resembles "player score." But if you didn't already know, you'd have to sit there and try to fill in the missing letters. Is it "plural scar," or "plastic scrabble"? Nope, it is "player score." You just have to sit there and study it. The one exception to this rule is common abbreviations or acronyms. HTML is fine.

**Rule #4: A good name will usually be kind of long.** In math, we usually use single letters for variable names. In programming, you usually need more than that to accurately describe what you're trying to do. In most cases, you'll probably have at least three letters. Often, it is 8 to 16. Don't be afraid if it gets longer than that. It's better to be descriptive than to "save letters."

**Rule #5: If your variables end with a number, you probably need a better name.** If you've got **count1** and **count2**, there's probably a better name for them. (Or perhaps an array, which we'll talk about later.)

**Rule #6: "data", "text", "number", and "item" are usually not descriptive enough.** For some reason, people seem to fall back to these all the time. They're OK, but they're just not very descriptive. It's best to come up with something more precise in any situation where you can.

**Rule #7: Make the words of the variable name stand out from each other**. This is so it is easier to read a variable name that is composed of multiple words. **playerScore** (with a capital 'S') and **player_score** are both more readable than **playerscore**. My personal preference is the first, but both work.

## Try It Out!

**Variables Quiz.** Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. Name the three things all variables have.
2. **True/False.** You can use a variable before it is declared.
3. How many times must a variable be declared?
4. Out of the following, which are legal variable names? answer, 1stValue, value1, $message, delete-me, delete_me, PI.

**Answers: (1)** name, type, value. **(2)** False. **(3)** 1. **(4)** answer, value1, delete_me, PI.

# 21 Structs

**In a Nutshell**

- A *struct* or *structure* is similar to a class in terms of the way it is organized, but a struct is a value type, not a reference type.
- Structs should be used to store compound data (composed of more than one part) that does not involve a lot of complicated methods and behaviors.
- All of the simple types are structs.
- The primitive types are all aliases for certain pre-defined structs and classes.

A few chapters ago we introduced classes. These are complex reference types that you can define and build from the ground up. C# has a feature call *structs* or *structures* which look very similar to classes organizationally, but they are value types instead of reference types.

In this chapter, we'll take a look at how to create a struct, as well as discuss how to decide if you need a struct or a class. We'll also discuss something that may throw you for a loop: all of the built-in types like **bool**, **int**, and **double**, are actually all aliases for structures (or a class in the case of the **string** type).

## Creating a Struct

Creating a struct is very similar to creating a class. The following code defines a simple struct, and an identical class that does the same thing:

```
struct TimeStruct
{
    private int seconds;

    public int Seconds
    {
        get { return seconds; }
        set { seconds = value; }
    }

    public int CalculateMinutes()
    {
        return seconds / 60;
    }
}

class TimeClass
{
    private int seconds;

    public int Seconds
    {
        get { return seconds; }
        set { seconds = value; }
    }

    public int CalculateMinutes()
    {
        return seconds / 60;
    }
}
```

You can see that the two are very similar—in fact the same code is used in both, with the single solitary difference being the **struct** keyword instead of the **class** keyword.

## Structs vs. Classes

Since the two are so similar in appearance, you're probably wondering how the two are different.

The answer to this question is simple: structs are value types, while classes are reference types. If you didn't fully grasp that concept back when we discussed it in Chapter 16, it is probably worth going back and taking a second look.

While this is a single difference in theory, this one change makes a world of difference. For example, a struct uses value semantics instead of reference semantics. When you assign the value of a struct from one variable to another, the entire struct is copied. The same thing applies for passing one to a method as a parameter, and returning one from a method.

Let's say we're using the struct version of the **TimeStruct** we just saw, and did this:

```
public static void Main(string[] args)
{
    TimeStruct time = new TimeStruct();
    time.Seconds = 10;

    UpdateTime(time);
}

public static void UpdateTime(TimeStruct time)
{
    time.Seconds++;
}
```

In the **UpdateTime** method, we've received a copy of the **TimeStruct**. We can modify it if we want, but this hasn't changed the original version, back in the **Main** method. We've modified a copy, and the original still has a value of 10 for **seconds**.

Had we used **TimeClass** instead, handing it off to a method copies the reference, but that copied reference still points the same actual object. The change in the **UpdateTime** method would have affected the time variable back in the **Main** method.

Like I said back when we were looking at reference types, this can be a good thing or a bad thing, depending on what you're trying to do, but the important thing is that you are aware of it.

Interestingly, while we get a copy of a value type as we move it around, it doesn't necessarily mean we've completely duplicated everything it is keeping track of. Let's say you had a struct that contained within it a reference type, like an array, as shown below:

```
struct Wrapper
{
    public int[] numbers;
}
```

And then we used it like this:

```
public static void Main(string[] args)
{
    Wrapper wrapper = new Wrapper();
    wrapper.numbers = new int[3] { 10, 20, 30 };
    UpdateArray(wrapper);
}

public void UpdateArray(Wrapper wrapper)
{
    wrapper.numbers[1] = 200;
}
```

We get a copy of the **Wrapper** type, but for our **numbers** instance variable, that's a copy of the reference. The two are still pointing to the same actual array on the heap.

Tricky little things like this are why if you don't understand value and reference types, you're going to get bit by them. If you're still fuzzy on the differences, it's worth a second reading of Chapter 16.

There are other differences that arise because of the value/reference type difference:

- Structs can't be assigned a value of **null**, since **null** indicates a reference to nothing.
- Because structs are value types, they'll be placed on the stack when they can. This could mean faster performance because they're easier to get to, but if you're passing them around or reassigning them a lot, the time it takes to copy them could slow things down.

Another big difference between structs and classes is that in a struct, you can't define your own parameterless constructor. For both classes and structs, if you don't define any constructors at all, one still exists: a default parameterless constructor. This constructor has no parameters, and is the simplest way to create new objects of a given type, assuming there's no special setup logic required.

With classes, you can create your own parameterless constructor, which then allows you to replace the default one with your own custom logic. This cannot be done with structs. The default parameterless constructor creates new objects where everything is zeroed out. All numbers within the struct start at 0, all **bool**s start at false, all references start at **null**, etc. While you can create other constructors in your struct, you cannot create a parameterless one to replace this default one.

## Deciding Between a Struct and a Class

Despite the similarities in appearance, structs and classes are made for entirely different purposes. When you create a new type, which one do you choose? Here are some things to think about as you decide.

For starters, do you have a particular need to have reference or value semantics? Since this is the primary difference between the two, if you've got a good reason to want one over the other, your decision is basically already made.

If your type is not much more than a compound collection of a small handful of primitives, a struct might be the way to go. For instance, if you want something to keep track of a person's blood pressure, which consists of two integers (systolic and diastolic pressures) a struct might be a good choice. On the other hand, if you think you're going to have a lot of methods (or events or delegates, which we'll talk about in Chapters 32 and 33) then you probably just want a class.

Also, structs don't support inheritance which is something we'll talk about in Chapter 22, so if that is something you may need, then go with classes.

In practice, classes are far more common, and probably rightly so, but it is important to remember that if you choose one way or the other, and then decide to change it later, it will have a huge ripple effect throughout any code that uses it. Methods will depend on reference or value semantics, and to change from one to the other means a lot of other potential changes. It's a decision you want to make consciously, rather than just always defaulting to one or the other.

## Prefer Immutable Value Types

In programming, we often talk about types that are *immutable*, which means that once you've set them up, you can no longer modify them. (As opposed to mutable types, which you can modify parts of its data on the fly.) Instead, you would create a new copy that is similar, but with the changes you want to make. All of the built-in types (including the **string** type, which is a reference type) are immutable.

There are definite benefits to making both value and reference types immutable, but especially so with structs. This is because we think of value types like structs as a cohesive specific value. Because it has value semantics (copies of the whole thing are made, rather than just making a second reference to the same actual bytes in memory) we end up duplicating value types all over the place.

If we aren't careful, with a mutable, changeable value type, we might think we're modifying the original, but are instead modifying the original.

For example, what will the following code output?

```
struct S
{
    public int Value { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        S[] values = new S[10];             // New array of structs with default values is created here.
        S item = values[0];                 // Danger! Copy is made here.
        item.Value++;                       // Copy is modified here.
        Console.WriteLine(values[0].Value); // Original, unmodified value is printed here.
    }
}
```

It actually prints out 0. This might come as a surprise. This is because the line that says **S item = values[0];** produces a copy for the assignment. So when you do **item.Value++**, you are modifying the copy, not the original. (This would not be true if **S** were a class instead of a struct.)

If we make **S** immutable so you can't modify its **Value** property at all, then the only way to produce a new version with the correctly incremented value would be to create a new **S** object, populated with the correct value at construction time. (You would want to define a constructor that allows you to specify value at creation time if you do this.)

At this point, that **item.Value++** line would have to become **item = new S(item.Value + 1);**, and the error becomes much more obvious to spot.

Making things in general immutable has many benefits, but for structs, you should definitely have a preference for making them immutable. (Sometimes the overhead performance cost associated with creating lots of objects will supersede the usefulness of immutable types, for example.)

## The Built-In Types are Aliases

Back in Chapter 6, we took a look at all of the primitive or built-in types that C# has. This includes things like **int**, **float**, **bool**, and **string**. In Chapter 16, we looked at value types vs. reference types, and we discovered that these primitive types are value types, except for **string**, which is a reference type.

In fact, more than just being value types, they are actually structs! This means that everything we've been learning about structs also applies to these built-in types.

Even more, all of the primitive types are *aliases* for other structs (or class, in the case of the **string** type).

We've been working with things like the **int** type. But behind the scenes the C# compiler is simply changing this over to a struct that is defined in the same kind of way that we've seen here. In this case, it is the **Int32** struct (**System.Int32**).

So while we've been writing code that looks like this:

```
int x = 0;
```

We could have also used this:

```
Int32 x = new Int32();
Int32 y = 0;             // Or combined.
int z = new Int32();     // Or combined another way. It's all the same thing.
int w = new int();       // Yet another way...
```

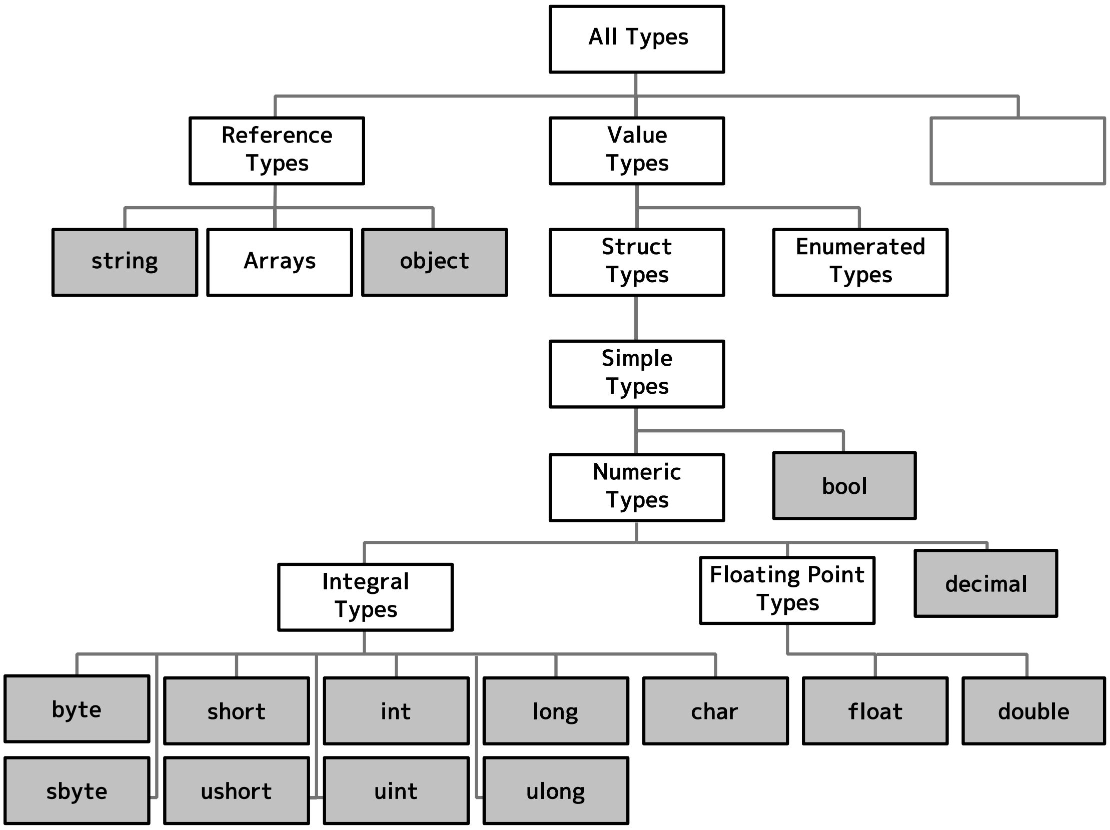
The following table identifies the aliases for each of the built-in types:

| Primitive Type | Alias For: |
|---|---|
| bool | System.Boolean |
| byte | System.Byte |
| sbyte | System.SByte |
| char | System.Char |
| decimal | System.Decimal |
| double | System.Double |
| float | System.Single |
| int | System.Int32 |
| uint | System.UInt32 |
| long | System.Int64 |
| ulong | System.UInt64 |
| object | System.Object |
| short | System.Int16 |
| ushort | System.UInt16 |
| string | System.String |

With only a few of exceptions, the “real” struct name is the same as the keyword version, just with different capitalization. Keywords in C# are all lowercase by convention, but nearly everybody will capitalize type names, which explains that difference.

You'll also see that instead of **short**, **int**, and **long**, the structs use **Int** followed by the number of bits they use. It explicitly states exactly how many bits are in each type, which gives some clarity.

Additionally, **float** becomes **Single** rather than **Float**. Technically, **float**, **double**, and **decimal** are all floating point types. But **double** has twice the bits as **float**, so the term "single" is a more specific (and therefore technically more accurate) name for it. The C# language designers stuck with **float** because that's the keyword that is used for this data type in many other languages.



**Try It Out!**

**Structs Quiz.** Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. Are structs value types or reference types?
2. **True/False.** It is easy to change classes to structs, or structs to classes.
3. **True/False.** Structs are always immutable.
4. **True/False.** Classes are never immutable.
5. **True/False.** All primitive/built-in types are structs.

**Answers: (1)** Value types. **(2)** False. **(3)** False. **(4)** False. **(5)** False. string and object are reference types.

37

# Lambda Expressions

**In a Nutshell**

- Lambda expressions are methods that appear “in line” and do not have a name.
- Lambda expressions have different syntax than normal methods, which for simple lambda expressions makes it very readable. The expression: **x => x < 5** is the equivalent of the method **bool AMethod(int x) { return x < 5; }**.
- Multiple parameters can be used: **(x, y) => x * x + y * y**
- As can zero parameters: **() => Console.WriteLine("Hello World!")**
- The C# compiler can typically infer the types of the variables in use, but if not, you can explicitly provide those types: **(int x) => x < 5**.
- If you want more than one expression, you can use a statement lambda instead, which has syntax that looks more like a method: **x => { bool lessThan5 = x < 5; return lessThan5; }**
- Lambda expressions can use variables that are in scope at the place where they are defined.
- Expression syntax can be used to define normal, named methods, properties, indexers, and operators as well: **bool AMethod(int x) => x < 5;**

## The Motivation for Lambda Expressions

Lambda expressions are a relatively simple concept. The trick to understanding lambda expressions is in understanding what they’re actually good for. So that’s where we’re going to start our discussion.

For this discussion, let’s say you had the following list of numbers:

```
// Collection initializer syntax (see Chapter 25).
List<int> numbers = new List<int>(){ 1, 7, 4, 2, 5, 3, 9, 8, 6 };
```

Let’s also say that somewhere in your code, you want to filter out some of them. Perhaps you want only even numbers. How do you do that?

### The Basic Approach

Knowing what we learned way back in some of the early chapters about methods and looping, perhaps we could create something like this:

```
public static List<int> FindEvenNumbers(List<int> numbers)
{
    List<int> onlyEvens = new List<int>();

    foreach(int number in numbers)
    {
        if(number % 2 == 0) // checks if it is even using mod operator
            onlyEvens.Add(number);
    }

    return onlyEvens;
}
```

We could then call that method and get back our list of even numbers. But that's a lot of work for a single method that may only ever be used once.

## The Delegate Approach

Fast forward to Chapter 32, where we learned about delegates. For this particular task, delegates will actually be able to go a long way towards helping us.

As it so happens, there's a method called **Where** that is a part of the **List** class (actually, it is an extension method) that uses a delegate. Using the **Where** method looks like this:

```
IEnumerable<int> evenNumbers = numbers.Where(MethodMatchingTheFuncDelegate);
```

The **Func** delegate that the **Where** method uses is generic, but in this specific case, must return the type **bool**, and have a single parameter that is the same type that the **List** contains (**int**, in this example). The **Where** method goes through each element in the array and calls the delegate for each item. If the delegate returns true for the item, it is included in the results, otherwise it isn't.

Let me show you what I mean with an example. Instead of our first approach, we could write a simple method that determines if a number is even or not:

```
public static bool IsEven(int number)
{
    return (number % 2 == 0);
}
```

This method matches the requirements of the delegate the **Where** method uses in this case (returns **bool**, with exactly one parameter of type **int**).

```
IEnumerable<int> evenNumbers = numbers.Where(IsEven);
```

That's pretty readable and fairly easy to understand, as long as you know how delegates work. But let's take another look at this.

## Anonymous Methods

While what we've done with the delegate approach is a big improvement over crafting our own method to do all of the work, it has two small problems. First, a lot of times that we do something like this, the method is only ever used once. It seems like overkill to go to all of the trouble of creating a whole method to do this, especially since it starts to clutter the namespace. We can no longer use the name **IsEven** for anything else within the class. That may not be a problem, but it might.

Second, and perhaps more important, that method is located somewhere else in the source code. It may be elsewhere in the file, or even in a completely different file. This separation makes it a bit harder to truly understand what's going on when you look at the source code. It our current case, this is mostly solved by calling the method something intelligent (**IsEven**) but you don't always get so lucky.

This issue is common enough that back in C# 2.0, they added a feature called *anonymous methods* to deal with it. Anonymous methods allow you to define a method "in line," without a name.

I'm not going to go into a whole lot of detail about anonymous methods here, because lambda expressions mostly replaced them.

To accomplish what we were trying to do with an anonymous method, instead of creating a whole method named **IsEven**, we could do the following:

```
numbers.Where(delegate(int number) { return (number % 2 == 0); });
```

If you take a look at that, you can see that we're basically taking the old **IsEven** method and sticking it in here, "in line."

This solves our two problems. We no longer have a named method floating around filling up our namespace, and the code that does the work is now at the same place as the code that needs the work.

I know, I know. You're probably saying, "But that code is not very readable! Everything's just smashed together!" And you're right. Anonymous methods solved some problems, while introducing others. You would have to decide which set of problems works best for you, depending on your specific case.

But this finally brings us to lambda expressions.

## Lambda Expressions

Basically, a *lambda expression* is simply a method. More specifically, it is an anonymous method that is written in a different form that (theoretically) makes it a lot more readable. Lambda expressions were new in C# 3.0.

### In Depth

**The Name "Lambda."** The name "lambda" comes from lambda calculus, which is the mathematical basis for programming languages. It is basically the programming language people used before there were computers at all. (Which is kind of strange to think about.) "Lambda" would really be spelled with the Greek letter lambda (λ) but the keyboard doesn't have it, so we just use "lambda."

Creating a lambda expression is quite simple. Returning to the **IsEven** problem from earlier, if we want to create a lambda expression to determine if a variable was even or odd, we would write the following:

```
x => x % 2 == 0
```

The lambda operator (**=>**) is read as "goes to" or "arrow." (So, to read this line out loud, you would say "x goes to x mod 2 equals 0" or "x arrow x mod 2 equals 0.") The lambda expression is basically saying to take the input value, **x**, and mod it with 2 and check the result against 0.

You may also notice with a lambda expression, we didn't use **return**. The code on the right side of the **=>** operator must be an expression, which evaluates to a single value. That value is returned, and its type becomes the return type of the lambda expression.

This version is the equivalent of all of the other versions of **IsEven** that we wrote earlier in this chapter. Speaking of that earlier code, this is how we might use this along with everything else:

```
IEnumerable<int> evens = numbers.Where(x => x % 2 == 0);
```

It may take a little getting used to, but generally speaking it is much easier to read and understand than the other techniques that we used earlier.

## Multiple and Zero Parameters

Lambda expressions can have more than one parameter. To use more than one parameter, you simply list them in parentheses, separated by commas:

```
(x, y) => x * x + y * y
```

The parentheses are optional with one parameter, so in the earlier example, I've left them off.

This example above could have been written instead as a method like the following:

```
public int HypoteneuseSquared(int x, int y)
{
    return x * x + y * y;
}
```

Along the same lines, you can also have a lambda expression that has no parameters:

```
() => Console.WriteLine("Hello World!")
```

## Type Inference Failures and Explicit Types

The C# compiler's type inference is smart enough to look at most lambda expressions and figure out what variable types and return type you are working with, but in some cases, the type inference fails, and you have to fall back to explicitly stating the types in use, or the code won't compile.

If this happens, you'll need to explicitly put in the type of the variable, like this:

```
(int x) => x % 2 == 0;
```

Using explicit types in your lambda expressions is always an option, not just when the compiler can't infer the type. Most C# programmers will generally take advantage of type inference when possible in a lambda, but if you like the syntax better or if it makes some specific situation clearer, feel free to use a named type instead of just using type inference, even if it isn't required.

## Statement Lambdas

As you've seen by now, most methods are more than one line long. While lambda expressions are particularly well suited for very short, single line methods, there will be times that you'll want a lambda expression that is more than one line long. This complicates things a little bit, because now you'll need to add in semicolons, curly braces, and a **return** statement, but it can still be done:

```
(int x) => { bool isEven = x % 2 == 0; return isEven; }
```

The form we were using earlier is called an *expression lambda*, because it had only one expression in it. This new form is called a *statement lambda*. As a statement lambda gets longer, you should probably consider pulling it out into its own method.

## Scope in Lambda Expressions

From what we've seen so far, lambda expressions have basically behaved like a normal method, only embedded in the code and with a different, cleaner syntax. But now I'm going to show you something that will throw you for a loop.

Inside of a lambda expression, you can access the variables that were in scope at the location of the lambda expression. Take the following code, for example:

```
int cutoffPoint = 5;
List<int> numbers = new List<int>(){ 1, 7, 4, 2, 5, 3, 9, 8, 6 };

IEnumerable<int> numbersLessThanCutoff = numbers.Where(x => x < cutoffPoint);
```

If our lambda expression had been turned into a method, we wouldn't have access to that **cutoffPoint** variable. (Unless we supplied it as a parameter.) This actually adds a ton of power to the way lambda expressions can work, so it is good to know about.

(For what it's worth, anonymous methods have the same feature.)

## Expression-Bodied Members

Lambda expressions were introduced to C# in version 3.0, and as I mentioned earlier, one of the big draws to it is that the syntax is much more concise. That's great for short methods that would otherwise require a lot of overhead to define.

C# 6.0 extends this a little, allowing you to use the same expression syntax to define normal non-lambda methods within a class. For example, consider the method below:

```
public int ComputeSquare(int value)
{
    return value * value;
}
```

Now that we know about lambda expressions and the syntax that goes with them, it makes sense to point out that this method could also be implemented with the same expression syntax:

```
public int ComputeSquare(int value) => value * value;
```

This only works if the method can be turned into a single expression. In other words, we can use the expression lambda syntax, but not the statement lambda syntax. If we need a statement lambda, we would just write a normal method.

This syntax is not just limited to methods. Any method-like member of a type can use the same syntax. So that includes indexers, operator overloads, and properties (though this only applies to read-only properties where your expression defines the getter and the property has no setter). The following simple class shows all four of these in operation:

```
public class SomeSortOfClass
{
    // These two private instance variables are used by the methods below.
    private int x;
    private int[] internalNumbers = new int[] { 1, 2, 3 };

    // Property (read-only, no setter allowed)
    public int X => x;

    // Operator overload
    public static int operator +(SomeSortOfClass a, SomeSortOfClass b) => a.X + b.X;

    // Indexer
    public int this[int index] => internalNumbers[index];

    // Normal method
    public int ComputeSquare(int value) => value * value;
}
```

.

## Lambdas vs. Local Functions

In all cases where you might use a lambda, you could also use a local function, which was introduced in Chapter 28. The scenarios in which you might use a lambda are also good fits for local functions, and the two can even be combined together, using an expression-bodied local function. To illustrate, consider the following three methods which are all equivalent in terms of functionality:

```
public static IEnumerable<int> FindEvenNumbers1(List<int> numbers)
{
    return numbers.Where(x => x % 2 == 0); // Plain lambda expression.
}

public static IEnumerable<int> FindEvenNumbers2(List<int> numbers)
{
    bool IsEven(int number) // Local function.
    {
        return number % 2 == 0;
    }

    return numbers.Where(IsEven);
}

public static IEnumerable<int> FindEvenNumbers3(List<int> numbers)
{
    bool IsEven(int number) => number % 2 == 0; // Expression-bodied local function.

    return numbers.Where(IsEven);
}
```

Each of the three above options are functionally equivalent, but with rather different syntax. The first is a plain lambda expression. This is probably the most concise of the three, and for somebody comfortable with lambda expressions, is quite readable.

The second is a local function. It isn't nearly as concise, but has the advantage of giving a name to the functionality.

The third is a local function with an expression body. This is something of a compromise of the two.

Each of the above can be the best option in different scenarios. All have their place. Pick the one that produces the most readable code for any given situation.

### Try It Out!

**Lambda Expressions Quiz.** Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** Lambda expressions are a special type of method.
2. **True/False.** A lambda expression can be given a name.
3. What operator is used in lambda expressions?
4. Convert the following to a lambda expression: **bool IsNegative(int x) { return x < 0; }**
5. **True/False.** Lambda expressions can only have one parameter.
6. **True/False.** Lambda expressions have access to the local variables in the method they appear in.

**Answers: (1)** True. **(2)** False. **(3)** Lambda operator (=>). **(4)** x => x < 0. **(5)** False. **(6)** True.

# Glossary

## .NET Core
A newer.NET Platform stack that is designed to be more cross-platform friendly, and primarily targets Linux and macOS. (Chapter 44.)

## .NET Framework
The oldest (original) most popular, and most complete stack within the .NET Platform. Aimed primarily at Windows computers. This term is frequently used to refer to the entire .NET ecosystem, though this book makes a distinction between these two, and calls the entire system the .NET Platform. (Chapters 1 and 44.)

## .NET Platform
The platform C# is built for and utilizes. The term used in this book to describe the entire .NET ecosystem, including all stacks (the .NET Framework, .NET Core, Xamarin, etc.), all app models, the entire .NET Standard Library, the compilers, CLR runtime, CIL language, and other tools. This is also sometimes called simply ".NET" and also frequently called the .NET Framework, though this book makes a distinction between the two. (Chapters 1 and 44.)

## .NET Standard
A specification that defines a vast collection of reusable types (classes, interfaces, structs, enums, etc.) that exist across multiple stacks within the .NET Platform. The .NET Standard allows you to reuse code and produce code that can be migrated from stack to stack. It allows you to write code that runs on the original .NET Framework, as well as .NET Core, Xamarin, and other stacks. The .NET standard has many different levels or version numbers. Higher version numbers include more reusable material. Lower version numbers allow you to target more diverse stacks. (Chapters 1 and 44.)

## .NET Standard Library
See *.NET Standard*.

## Abstract Class
A class that you cannot create instances of. Instead, you can only create instances of derived classes. The abstract class is allowed to define any number of members, both concrete (implemented) and abstract (unimplemented). Derived classes must provide an implementation for any abstract members defined by the abstract base class before you can create instances of the type. (Chapter 23.)

## Abstract Method
A method declaration that does not provide an implementation or body. Abstract methods can only be defined in abstract classes. Derived classes that are not abstract must provide an implementation of the method. (Chapter 23.)

## Accessibility Level
Types and members are given different levels that they can be accessed from, ranging from being available to anyone who has access to the code, down to only being accessible from within the type they are defined in. More restrictive accessibility levels make something less vulnerable to tampering, while less restrictive levels allow more people to utilize the code to get things done. It is important to point out that this is a mechanism provided by the C# language to make programmer's lives easier, but it is not a way to prevent hacking, as there are still ways to get access

to the code. Types and type members can be given an access modifier, which specifies what accessibility level it has. The **private** accessibility level is the most restrictive, and means the code can only be used within the type defining it, **protected** can be used within the type defining it and any derived types, **internal** indicates it can be used anywhere within the assembly that defines it, and **public** indicates it can be used by anyone who has access to the code. Additionally, the combination of **protected internal** can be used to indicate that it can be used within the defining type, a derived type, or within the same assembly. (Chapters 18 and 22.)

### Accessibility Modifier
See *Accessibility Level*.

### Anonymous Method
A special type of method where no name is ever supplied for it. Instead, a delegate is used, and the method body is supplied inline. Because of their nature, anonymous methods cannot be reused in multiple locations. Lambda expressions largely supersede anonymous methods and should usually be used instead. (Chapter 37.)

### Anonymous Type
A type (specifically a class) that does not have a formal type name and is created by using the new keyword with a list of properties. E.g., **new { A = 1, B = 2 }**. The properties of an anonymous type are read-only. (Chapter 19.)

### App Model
A component of the .NET Platform that allows you to easily create a specific type of application. This primarily consists of a library of reusable code for creating applications of that type, but also contains additional infrastructure such as a deployment model or a security model. (Chapter 44.)

### Argument
See *parameter*.

### Array
A collection of multiple values of the same type, placed together in a list-like structure. (Chapter 13.)

### ASP.NET
An app model for building web-based applications using the .NET Framework or .NET Core stacks. This book does not cover ASP.NET in depth. (Chapter 44.)

### Assembly
Represents a single block of redistributable code, used for deployment, security, and versioning. An assembly comes in two forms: a process assembly, in the form of an EXE file, and a library assembly, in the form of a DLL file. An EXE file contains a starting point for an application, while a DLL contains reusable code without a specified starting point. See also *project* and *solution*. (Chapter 44.)

### Assembly Language
A very low level programming language where each instruction corresponds directly to an equivalent instruction in machine or binary code. Assembly languages can be thought of as a human readable form of binary. (Chapter 44.)

### Assignment
The process of placing a value in a specific variable. (Chapter 5.)

### Associativity
See *Operator Associativity*.

### Asynchronous Programming
The process of taking a potentially long running task and pulling it out of the main flow of execution, having it run on a separate thread at its own pace. This relies heavily on threading. (Chapters 39 and 40.)

### Attribute
A feature of C# that allows you to give additional meta information about a type or member. This information can be used by the compiler, other tools that analyze or process the code, or at run-time. You can create custom attributes by creating a new type derived from the **Attribute** class. Attributes are applied to a type or member by using the name and optional parameters for the attribute in square brackets immediately above the type or member's declaration. (Chapter 43.)

### Base Class
In inheritance, a base class is the one that is being derived from. The members of the base class are included in the derived type. A base class is also frequently called a superclass or a parent class. A class can be a base class, and a derived class simultaneously. See also *inheritance*, *derived class*, and *sealed class*. (Chapter 22.)

### Base Class Library
The .NET Standard Library implementation that is a part of the .NET Framework. It is the most expansive and most widely use implementation of the Standard Library (Chapter 44.)

### BCL
See *Base Class Library*.

# Index

# The C# Player's Guide

3
THIRD EDITION

The C# Player's Guide is the ultimate guide for people starting out with C#, whether you are new to programming, or an experienced vet. This guide takes you from your journey's beginning, through the most challenging parts of programming in C#, and does so in a way that is casual, informative, and fun.

- Get off the ground quickly, with a gentle introduction to C#, Visual Studio, and a step-by-step walkthrough of how to make your first C# program.
- Learn the fundamentals of procedural programming, including variables, math operations, decision making, looping, methods, and an in-depth look at the C# type system.
- Delve into object-oriented programming from start to finish, including inheritance, polymorphism, interfaces, and generics.
- Explore some of the most useful advanced features of C#, and take on some of the most common tasks that a programmer will tackle.
- Learn to control the tools and tricks of programming in C#, including the .NET framework, Visual Studio, dealing with compiler errors, and hunting down bugs in your program.
- Master the needed skills by taking on a large collection of Try It Out! challenges and quizzes to ensure that you've learned the things you need to.

With this guide, you'll soon be off to save the world (or take over it) with your own awesome C# programs!

Check out starboundsoftware.com/books/c-sharp for solutions to challenge problems, additional content, and support.

Starbound

**Shelve in Programming/C#**
**User level: Beginner**