The C# Player's Guide

Using C# 5.0 and .NET 4.5

The must have guide for anyone venturing into the world of programming with C#

RB Whitaker

The C# Player's Guide

RB Whitaker

Starbound Software

Starbound Software
879 W Mallard Loop
Logan, UT 84321

ISBN: 0-985-58010-0
ISBN-13: 978-0-9855801-0-0

Contents at a Glance

Table of Contents

Acknowledgements

The task of writing a book is like writing software. When you start, you know 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 to the Mass Effect trilogy. (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 XNA 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 often get stuck, provide you with hands-on examples to explore, give you quizzes to take 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 a little over 300 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, in 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 games for the Xbox 360 with C#, we won't be discussing WPF, ASP.NET, XNA, 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 one 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 18, in my opinion, is the critical point of the book. It is where we get into the details of making your own classes, which is the most powerful way C# provides for building your own data types. Everything before this point is giving us the building blocks that we need to understand and make classes. Everything that we do after is simply providing us with more ways to use these custom-made types or showing how to use other classes that have already been made for us.

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 18, 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 programs in C#, additional information about the .NET Framework, 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 present you with a simple challenge problems and quizzes that gives you a chance to play around with some of the new concepts in the chapter, and test your understanding. If you were in a class, you'd get things like this as 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 at least spend a few minutes trying out each of these challenges to help you understand what you're reading and to help you be sure that you've learned what you needed to.

If you've got something else you want to try out with the things you're learning 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 46, 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 my solutions 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 understand, as far as C# is concerned, 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 Framework, 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, throughout the book.

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 kind of 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 over 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 come up with. This is especially true if you already know a similar programming language.

At a minimum, you should start from the beginning and read through Chapter 18. 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 at a pretty good pace.)

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 18 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 18 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 18 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 that you're creating. After a few weeks or a few months, when you've had a chance to make some real 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 opportunities for inadvertently leaving 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 can be a little like software, and there's always the opportunity for a version 2.0 to make the book better in the future. 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 all of the work you have done 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. This is helpful, because 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/.

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 Framework is.
- Outlines how your C# source code is turned into IL code by the compiler, then JIT compiled by the CLR as the program is running.

I'm going to start off this book with a real 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 Framework is, then this is the place for you.

I should point out that we'll get into a whole lot of detail about how the .NET Framework functions, and what it provides for you as a programmer in Chapter 40. This chapter will just provide a quick overview of the basics.

What Exactly 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 things they do boil down to instructions, written with 1's and 0's.

But humans are notoriously bad at doing anything with a random pile of billions 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 made up of 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 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.

C# is one of the most popular of all of the programming languages available. There are literally thousands, maybe even tens of thousands of these languages, and each one is designed for their own purposes. 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 Framework?

C# relies heavily on something called the .NET Framework. The .NET Framework is a large and powerful platform, which we'll discuss in detail in Chapter 40. (You can go read it as soon as you're done with this chapter, if you want.)

The .NET Framework primarily consists of two parts. The first part is the *Common Language Runtime*, often abbreviated as the *CLR*. The CLR is a virtual machine—a special type of software program that functions as though it is an actual computer. C# code is actually executed by the CLR, instead of being run by the actual physical computer. There are, of course, benefits and drawbacks to this kind of a setup, which we'll discuss in Chapter 40, but it turns out to be a good idea for everything except low level stuff, like an operating system or a device driver. C# is a bad choice for things like that. (Try C or C++ instead.)

I should also point out that though running your code through a virtual machine may make it a bit slower, in most cases, this isn't enough to matter. In some situations, it could actually end up being *faster*. In other cases, you can call outside code that not running on a virtual machine. The bottom line is, don't stress too much about if the CLR will be fast enough.

Because C# code runs on the .NET Framework, the process your code goes through before executing is a little more complex than what I described earlier. Rather than just a single step to compile your code to binary executable code, it goes through two steps. Here's what happens:

- The C# compiler turns your source code into *Common Intermediate Language* (*CIL* or *IL* for short).
- This IL code is packaged into a .exe file or .dll file, which can be shared with others.
- When it is time to run your program, the IL code in your .exe or .dll will be handed off to the CLR to run it.
- As your program is running, the CLR will look at the IL code and compile it into binary executable code that the computer it is running on understands. For each block of code you have, this will only happen once each time you run your program—the first time it needs it. This is called *Just-in-Time compiling*, or *JIT compiling*.

The other piece of the .NET Framework is an absolutely massive library of code that you can reuse within your own programs, to accelerate the development of whatever you are working on. This library is much bigger than what you typically find packaged with a programming language (though Java also has a library of comparable size). It is impossible to cover all of this code in this or any book, though we'll cover the most important stuff. Just keep this in mind, and before trying to build your own code for what seems like a common task, search the Internet to see if this code already exists as a part of the core C# library.

C# and .NET Versions

The latest version of the C# programming language is version 5, while the newest version of the .NET Framework is version 4.5. The two were released at the same time, in August 2012. In most cases, these latest versions are what you will want to use for new programs.

There are very few actual changes to the C# language in this latest release. There are more changes and additions to the .NET Framework's vast code library (the Base Class Library). This book is focused on the C# language, and as such, covers all of the features of C# itself, including the new features in version 5. While this book doesn't cover the entire Base Class Library, parts that are particularly important or useful, including things that are new in .NET 4.5, are discussed throughout the book.

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 exist, including both free versions that specialize in specific types of programs (desktop, web, metro-style apps, and phone) and full, paid-for versions that combine them all in one, and add some extra nice features.
- 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 in short, as you're getting started, you should consider either Visual Studio 2012, if you have the money for it, or the free Visual Studio Express 2012 for Desktop, at least while you're getting started. Once you know the basics of C#, you can then move on to the other more specialized express editions.

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

With C# in particular, nearly everyone chooses to use some variation of Visual Studio, which is made by Microsoft. There are actually lots of different flavors of Visual Studio (including a variety of free ones), tailored for different tasks and programmers, and in this chapter, I'll help guide you through the process of picking the right one for whatever you're trying to do.

Versions of Visual Studio

The 2012 family of Visual Studio were released in August of 2012. There are lots of options to choose from, and so I want to take a little bit of time to outline the various versions and what they're good for, to help you decide which one to start with.

Microsoft has gone out of their way to make sure that the versions all work in a similar way (as similar as possible) so that to a large degree, once you know how one works, you know how they all work. More importantly, because of this, while you're going through this book, it largely won't matter which one you select. Everything we cover in this book will work in all of the versions with a couple of exceptions and variations, which I'll point out as we go.

Along those lines, remember that throughout this book, when I use the name "Visual Studio", I'm referring to any of the versions described here, including the Visual Studio Express versions.

Visual Studio 2012

This is the latest and greatest, but it costs money. (I'll tell you about the free variations in a second.) It allows you to use everything including the new .NET 4.5 stuff, and it combines nearly everything that you'd want to do with C#. The full Visual Studio 2012 comes in various levels, including Professional (the "low" end), Premium, and Ultimate. Professional costs about $500, while Ultimate costs over $6000. I won't get into too many details on the differences between these versions. You're reading this book because you're just getting started, and so probably anything above Professional would be overkill for now.

This is the version to get if you have the money for it, but nothing we cover in this book will ever need the full, paid-for version. In fact, between the various free express editions of Visual Studio, you can make pretty much any program that you can with the full "pro" versions. This pro version just makes it easier.

You can find this version here: http://www.microsoft.com/visualstudio/eng/downloads#professional

Visual Studio Express 2012 for Windows Desktop

If you're not willing to spend the money (you're not alone) the express editions offer a nice free alternative. There are a variety of express editions available. While the full Visual Studio allows you to make any application of any type, all from the same IDE, the express editions split things by project type.

For instance, there's an express edition that is designed specifically for web-based application development and one for Windows phone development. The web version makes it easy to do web-based development, but does not include anything for Windows phone development, and so on. We'll get into the details on these other express editions in a second.

The one that you should start with is Visual Studio Express 2012 for Windows Desktop. This is because this version allows you to make console applications, which is the simplest type of program, and the place we'll start in this book. You can get Visual Studio Express 2012 for Windows Desktop from here: http://www.microsoft.com/visualstudio/eng/downloads#d-2012-express

Because there are other version of Visual Studio Express 2012, be sure to get the one that says Windows Desktop instead of Web, Windows 8, or the Team Foundation Server Express.

Other Members of the Visual Studio Family

There are other express versions of the Visual Studio family, and while I still recommend starting with Visual Studio Express 2012 for Windows Desktop to start, you will want to consider these other express versions as you progress.

This includes Visual Studio Express 2012 for Web, which is designed for making C# based websites using ASP.NET, Visual Studio Express for Windows Phone, which is built for making apps for the Windows 7 Phone, and Visual Studio Express 2012 for Windows 8, which is designed for making the new "Metro"-style applications that run on Windows 8. You can download these from the same link as I provided above for Visual Studio Express 2012 for Windows Desktop.

For comparison purposes, Visual Studio 2012 (the full version) is the equivalent of all of these express versions combined together, plus a variety of additional features, plus the ability to install (and create) add-ons to extend Visual Studio.

The Installation Process

Regardless of which version of Visual Studio you select, you should expect similar installation processes, with only a few minor differences between them. Go to the web page that I pointed out earlier and download the version of your choice.

Once downloaded, start the installer and follow the steps presented to you. The installation process takes several minutes, depending on how fast your Internet connection is. (There is an optional download that lets you get it all at once, if you won't have Internet access while you install.)

Once installed, it may place an icon on your desktop that you can use, but it will also put it in your Start Menu under All Programs. Look for it under Visual Studio 2012 or Visual Studio Express 2012, depending on what you installed. You can go there to start running your version of Visual Studio.

For all of the versions, you should also expect to have it ask you about a registration key. While all of the express editions are free and don't require payment, Microsoft is trying to keep track of how many people have their products installed, so when you get a chance, follow their instructions to go online and get a registration key. Otherwise, you'll only be able to use the program for a limited time.

In the next chapter we'll take a brief look at how Visual Studio works, and throughout this book, we'll continue to learn some of the tasks that you can do with the various versions of Visual Studio. While you're going through this book, it almost doesn't matter what version you choose. As far as the basics go, they're all very similar.

Throughout this book, I'm going to show you screenshots from Visual Studio 2012. Depending on which version you chose, things might look a little bit different, but for the most part, you should expect to see similar things to what you see in the screenshots. In the places that the versions are significantly different, I'll point out those differences so that you're never lost.

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.

Side Note

Visual Studio Alternatives. Almost everyone who writes programs in C# use some version of Visual Studio. It is generally considered the best and most comprehensive option. If you want an alternative, there are other choices. Just keep in mind that this book will assume you are using Visual Studio. Here are a few alternates:

- SharpDevelop: http://www.icsharpcode.net/OpenSource/SD/
- MonoDevelop: http://monodevelop.com/

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!

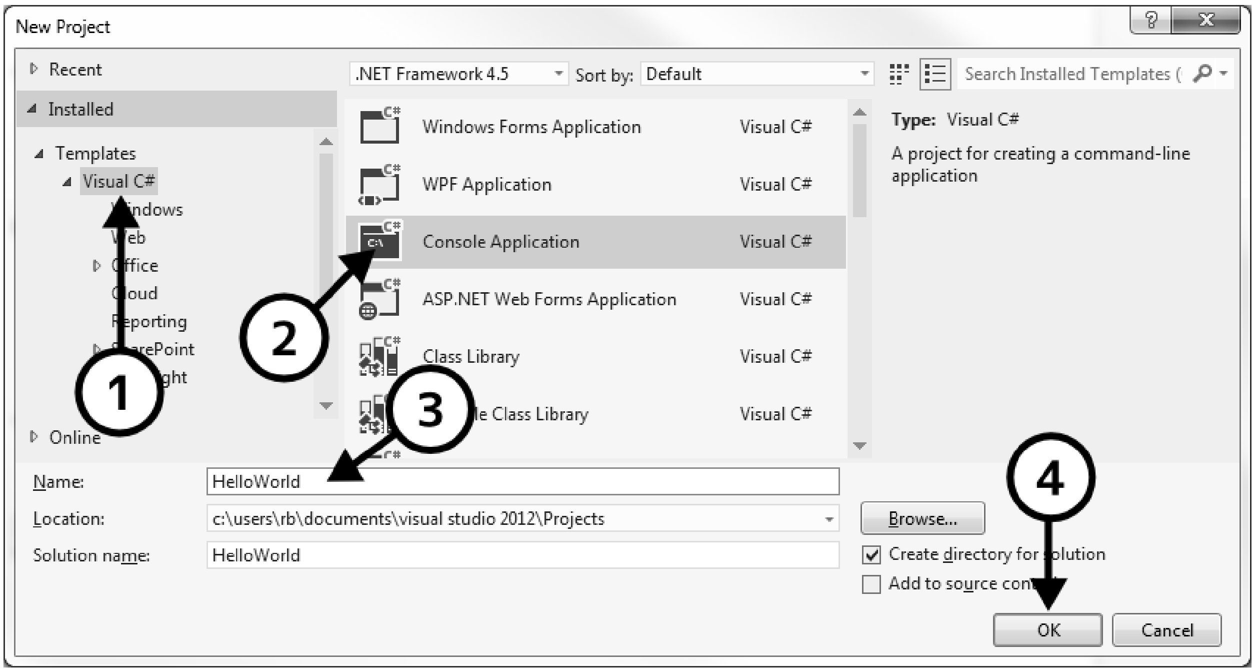
After that, we'll take a minute and look at the code that you have written in more detail before moving on to more difficult, but infinitely more awesome stuff in the future!

Creating a New Project

Let's get started with our first C# program! Open up Visual Studio, which we installed in Chapter 2. (Remember when I refer to Visual Studio, I'm referring to any member of the Visual Studio family that we discussed in Chapter 2, including any express editions.)

When the program first opens, you will see the Start Page come up. To create a new project, you can either select the "New Project..." button on the Start Page, or you can go up to the menu and choose **File > New Project...** in the express editions, or **File > New > Project...** in the full Visual Studio, from the menu bar.

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
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Forms Application Visual C#
Class Library Visual C#
Portable Class Library Visual C#
1
2
3
4
Browse...
Create directory for solution
Add to source control
OK
Cancel

Type: Visual C#

A project for creating a command-line application

Name: HelloWorld

Location: c:\users\rb\documents\visual studio 2012\Projects

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, 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** 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 we'll be using.

If you chose one of the other 2012 express editions, you won't see this template. It's for this reason that I recommend starting with Visual Studio 2012 or Visual Studio Express 2012 for Windows Desktop. The other 2012 express editions technically have the capability of building and running console applications, but a template isn't provided, which definitely makes things more complicated.

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

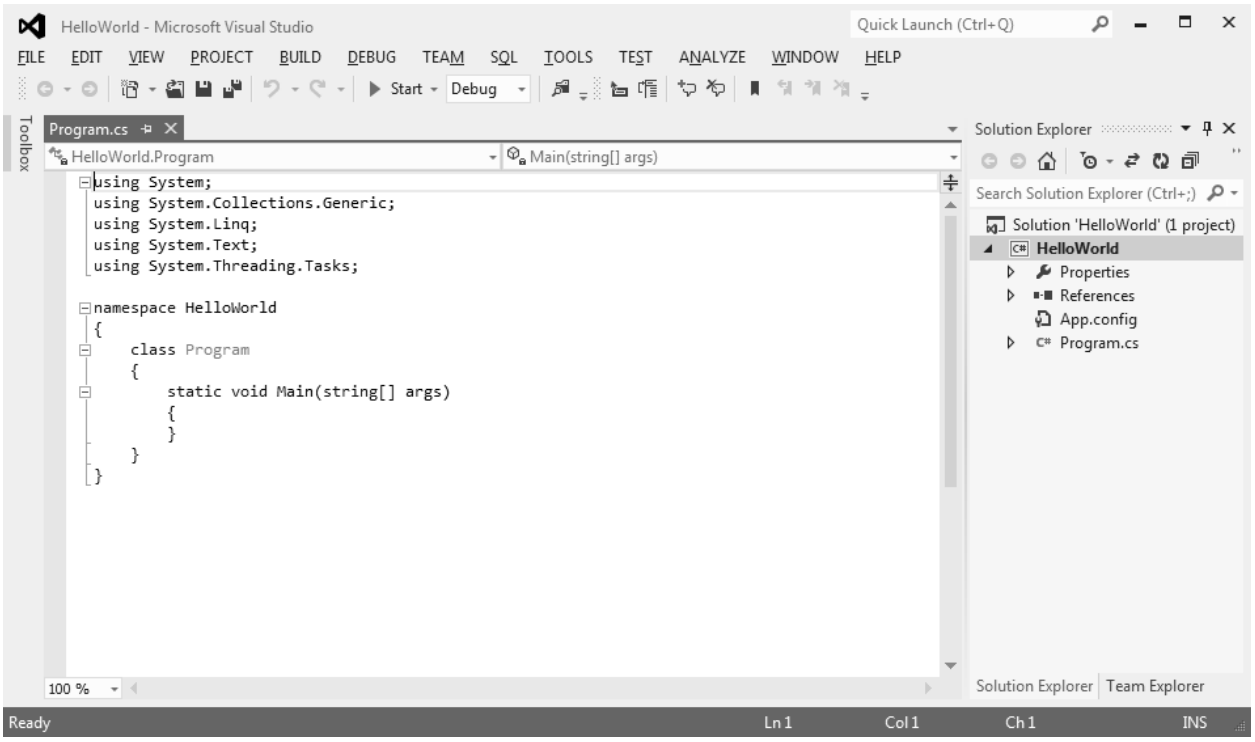
At the bottom of the dialog, type in a name for your project. I've called mine "HelloWorld." Your project will be saved in a directory with this name. It doesn't really matter what you call a project, but you want to name it something intelligent, so you can find it later when you are looking at a list of all of your projects. By default, Visual Studio tries to call your programs "ConsoleApplication1" or "ConsoleApplication2." If you haven't chosen a good name, you won't know what each of these do. By default, projects are saved under your Documents or My Documents directory (Documents/Visual Studio 2012/Projects/).

Finally, press the **OK** button to create your project! After you do this, you may need to wait for a little bit for Visual Studio to get everything set up for you.

A Brief Tour of Visual Studio

Once your project has loaded, it is worth at least a brief discussion of what you see before you. We'll look in depth at how Visual Studio works later on (Chapter 41) 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 going to View > Other Windows and choosing the window you want to see. 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 spread 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 a .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 a .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 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 is labeling "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

Now that our program is saved, we're 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;

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;

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.

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 really 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 later on, in Chapter 44.

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 but only 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 the 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.

The Visual Studio Express editions don't have a menu option to run in release mode by default, so you'll need to use the keyboard shortcut above. If you have Expert Settings turned on (see Chapter 41), it would show up as **Debug > Start Without Debugging.** I'll explain how to enable expert settings later on, when we discuss Visual Studio in depth. (The full Visual Studio automatically runs in a mode like Expert Settings, so if you're using the full Visual Studio, you will see that menu item.)

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;

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 when you write console apps. That's what we'll be doing in this book, but before long, you'll probably be making windows apps, XNA 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.

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 should only take a couple of minutes, and you'll get 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 of the lines of code in the program we've made. I'll try to 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;
```

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 has 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 26.)

So when you see a statement like **using System;** you know that there is a whole pile of code out there, that is labeled "System," which our code will want 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 four **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, 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 27.

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, as we add code in here, the computer is going to run it. 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 sort of 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 work 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 which is contained within the class, 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 26.

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 every single program on only one line! But don't do that. That would be a pretty 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 later, 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 most part, in this book I'll use the style in the last block, because I feel it is the easiest to read.

Semicolons

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

This is often 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 importantly, 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!

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 (in three different ways).

Unlike many other C# books, which ignore comments or only briefly mentions them, I've decided to put them front and center, describing them in detail right near 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. Whenever you get the chance, 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 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 (and others) can 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 like this (or the third form that we'll look at in Chapter 15) 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 a little time and describe some basic principles to help guide you in making 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 a bad example of a comment for a line of code:

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

Everything the comment says, we already knew. You might as well not even add it. Here's a better example.

```
// 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. I would venture a guess that it is possible to over-comment, 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, and 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.

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

Try It Out!

Comment *ALL* the things! While it may be overkill, in the name of putting 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.

20 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 couple of 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 your own 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 that we've been working with since we first learned about types 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 we use the **struct** keyword to create a struct, while we use the **class** keyword to create a class.

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 created in the previous example, and we 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. It 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, and 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 duplicated everything it is keeping track of entirely. 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 going back to Chapter 16 and covering it again.

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 could slow things down.

Deciding Between a Struct and a Class

Despite the similarities in appearance, structs and classes are designed for entirely different purposes. So when it comes time to create a new type, which one do you choose?

Here are a few things to think about as you decide between the two. 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, that should go a long way to helping you decide what to do.

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 numbers (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 30 and 31) then you probably just want a class.

Also, structs don't support inheritance which is something we'll talk about in Chapter 21, 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 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.

For value types like the structs you create, there is a danger to allowing them to be mutable. Because they have value semantics, any time you pass a value from one variable to another, or to a different method as a parameter, you end up with a copy of the original. It is far too easy and common to think that the value you got, which is actually a copy, could be used to modify the original. If your type doesn't allow you to modify the individual instance variables that make up your type, then the only way to make a change is by creating a new one with the changes you need applied to it.

Making a value type immutable will save you a great deal of trouble in the long run.

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, with the exception of **string**, which is actually a reference type.

It turns out that not only are they value types, but they are also struct types. This means that everything that applies to structs that we've been learning about also applies to these primitive types.

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

While we've been working with say, the **int** type, 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 **int** type and the **Int32** struct are identical. There is no difference at all between the two. Which brings us to a slightly updated definition of a *primitive type*, or *built-in type*: a type that the compiler has special knowledge about, and allows for special, simplified syntax to use it.

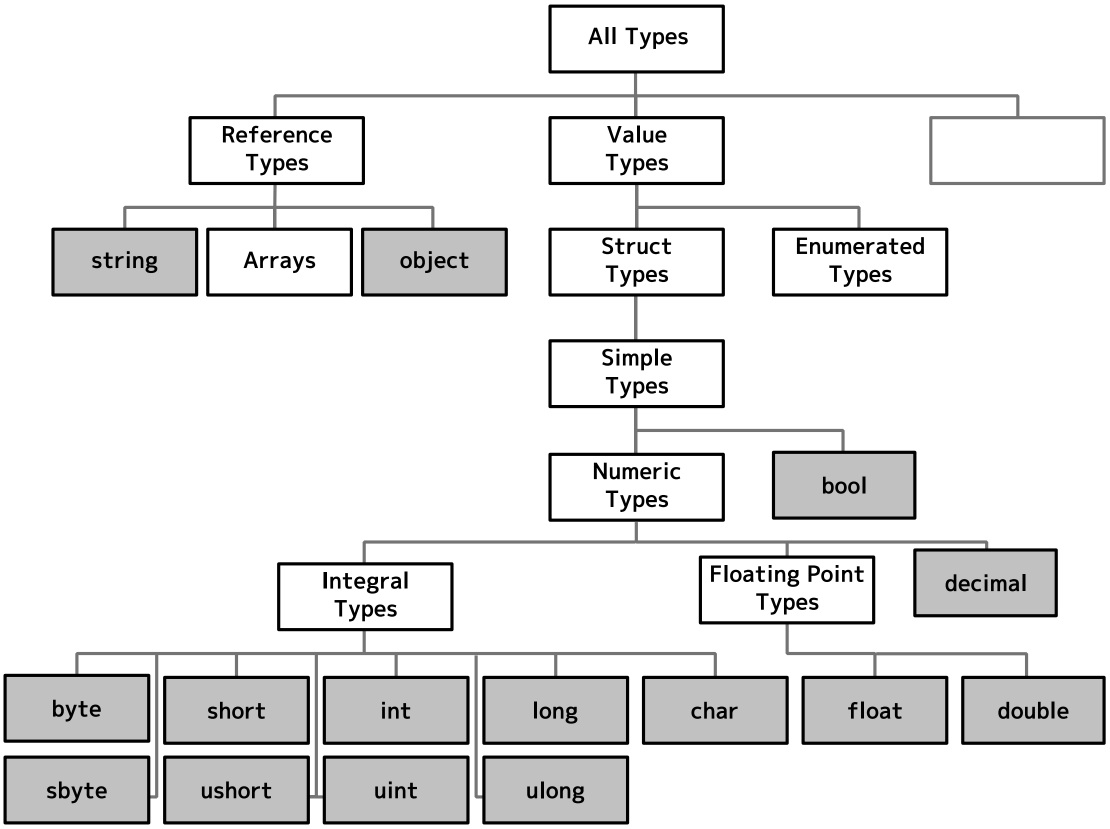
All of the primitive types have aliases, shown in the table below:

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

I want to point out a couple of things about the naming here. Nearly all of these types have the same name, except with a capital letter. 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. This is far more explicit than the keyword versions. There's no confusing how big each type is.

And last, you'll notice that the **float** type is **Single**, instead of **Float**. The word "float" is not very accurate, since both **double** and **float** are both technically floating point types. The term "single" is perhaps more correct, simply because it is more precise. The people who made C#, though, chose to use **float** because it is similar to the languages that it is based on (C/C++/Java) all of which have a **float** type.



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
- Zero parameters can be used: () => Console.WriteLine("Hello World!")
- The C# compiler can typically infer the types of the variables being used, 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.

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:

```
List<int> numbers = new List<int>();
numbers.Add(1);
numbers.Add(7);
numbers.Add(4);
numbers.Add(2);
numbers.Add(5);
numbers.Add(3);
numbers.Add(9);
numbers.Add(8);
numbers.Add(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 earlier 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 simply 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 Chapter30, 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. (Trust me, we're still coming back to lambda expressions here before long.)

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 it's not always perfect, and may not always apply.

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 tend to solve some problems, while bringing up 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.

Side Note

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 a strange thing to think about.) "Lambda" would really be spelled with the Greek letter lambda (λ) but the keyboard doesn't have it, so we just write "lambda."

Creating a lambda expression is quite simple. To continue with the problem we've been looking at, if we wanted 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." (So, to read this line out loud, you would say "x goes to 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.

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 and Explicit Types

The C# compiler is smart enough to look at most lambda expressions and figure out what variable types and return type you are working with. This is called *type inference*. For instance, in our first lambda expression it was smart enough to figure out that **x** was an integer and the whole expression returned a **bool**. With the expression **(x, y) => x * x + y * y** we saw that the C# compiler was smart enough to figure out that **x** and **y** were integer values and the resulting expression returned an **int** as well. And with **() => Console.WriteLine("Hello World!")**, it was smart enough to know that there were no parameters, and the lambda expression didn't return anything (**void** return type).

Type inference is actually a pretty big deal. It's not a trivial accomplishment, and I'd imagine there were a lot of smart people working on it to get it right.

Sometimes though, the compiler can't figure it out. If it can't, you'll get an error message when you compile. If this happens, you'll need to explicitly put in the type of the variable, like this:

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

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>();

// TODO: Add some values to numbers here...

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.)

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 Framework

The framework that C# is built for and utilizes, consisting of a virtual machine called the Common Language Runtime and a massive library of reusable code called the Framework Class Library. (Chapters 1 and 40.)

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 22.)

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 22.)

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 and the .NET Framework 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, 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 21.)

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.)

Argument
See *parameter*.

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

ASP.NET
A framework for building web-based applications using the .NET Framework. This book does not cover ASP.NET in depth. (Chapter 47.)

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 40.)

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 40.)

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

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. (Chapter 33.)

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 39.)

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 21.)

Base Class Library
The central library of code that nearly all C# programs will utilize, including the built-in types, arrays, exceptions, threading, and file I/O. (Chapter 40.)

BCL
See *Base Class Library*.

Binary Code
The executable instructions that computers work with to do things. All programs are built out of binary code. (Chapters 1 and 40.)

Binary Instructions
See *Binary Code*.

Binary Operator
An operator that works on two operands or values. Many of the most common operators are binary, such as addition and subtraction. (Chapter 7.)

Bit Field
The practice of storing a collection of logical (Boolean) values together in another type (such as **int** or **byte**) where each bit represents a single logical value. Enumerations can also be used as a bit field by applying the **Flags** attribute. When working with a bit field, the bitwise operators can be used to modify the individual logical values contained in the bit field. (Chapter 39.)

Bitwise Operator
One of several operators that operate on the individual bits of a value, as opposed to treating the bits as a single value with semantic meaning. Bitwise operators include bitwise logical operators, which perform operations like and, or, not, and xor (exclusive or) on two bits in the same location of two different values. It also includes bitshift operators, which slide the bits of a value to the left or right. In C#, the extra spots are filled with the value 0. (Chapter 39.)

Boolean
Pertaining to truth values. In programming, a Boolean value can only take on the value of true or false. Boolean types are a fundamental part of decision making in programming. (Chapter 6.)

Index

The C# Player's Guide

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 and explanation 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

US $24.99
Shelve in Programming/C#
User level: Beginner