THE C# PLAYER'S GUIDE

FOURTH EDITION

C# 9
.NET 5
Visual Studio 2019

RB WHITAKER

The C# Player's Guide

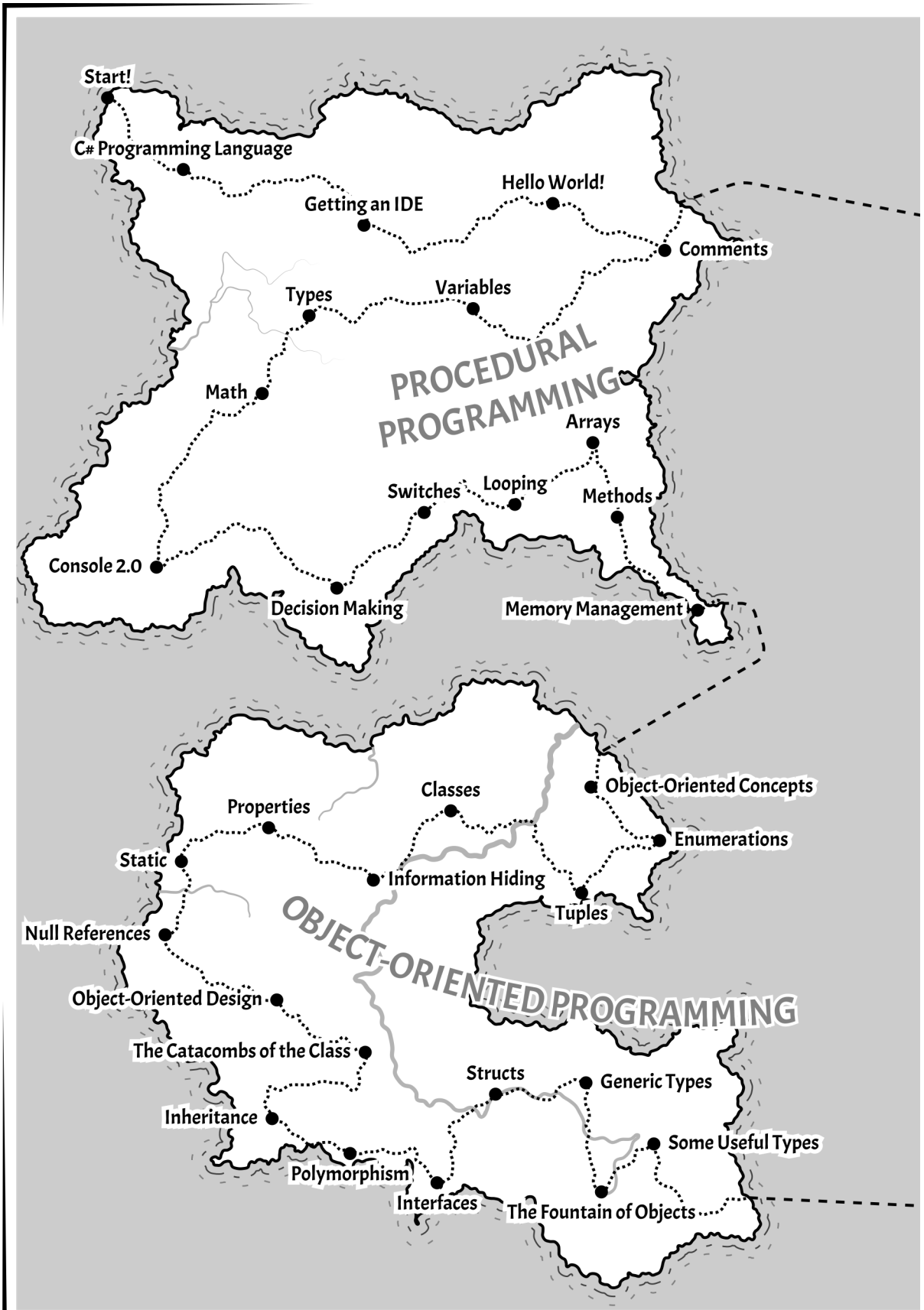
Fourth Edition

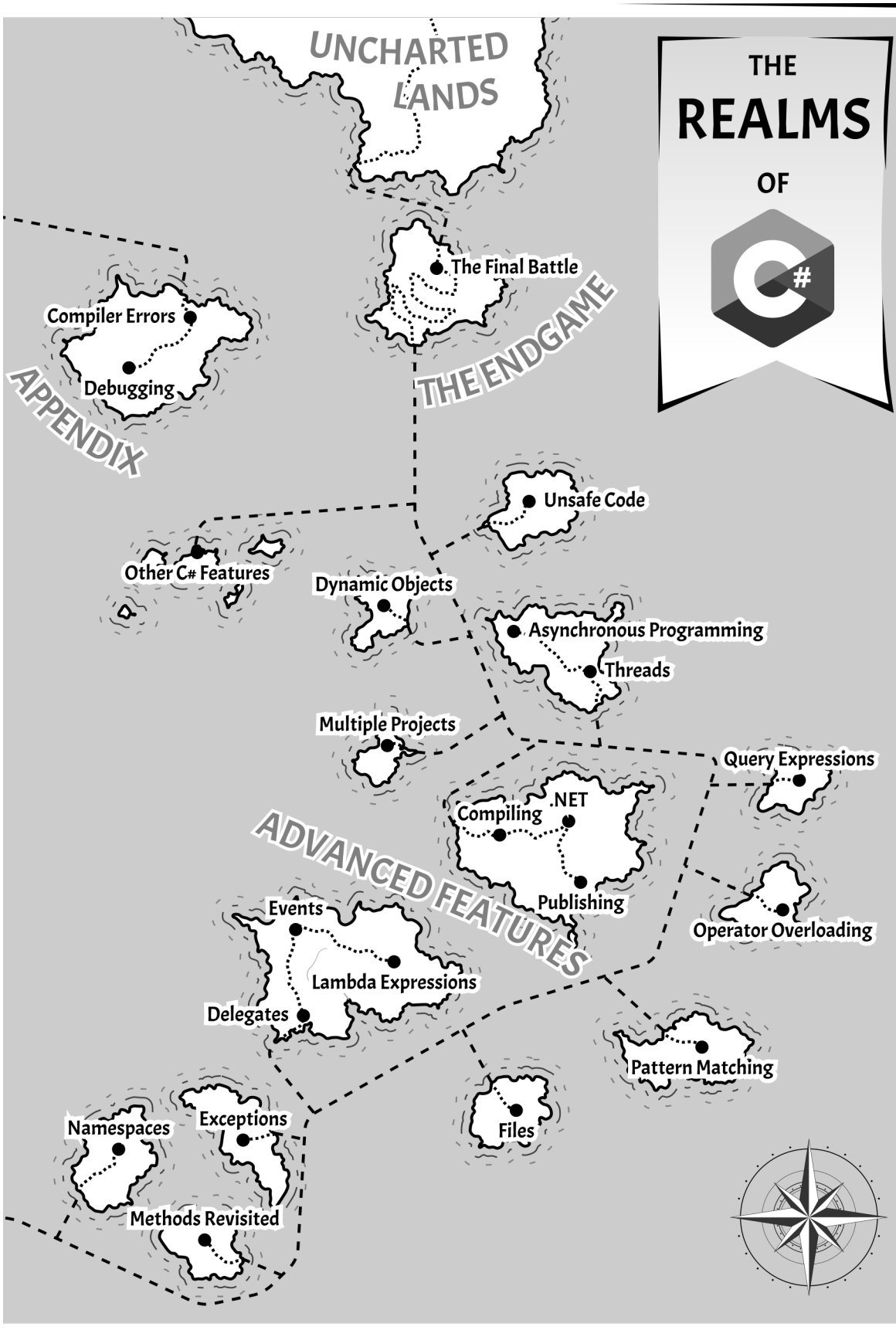
RB Whitaker

Starbound Software

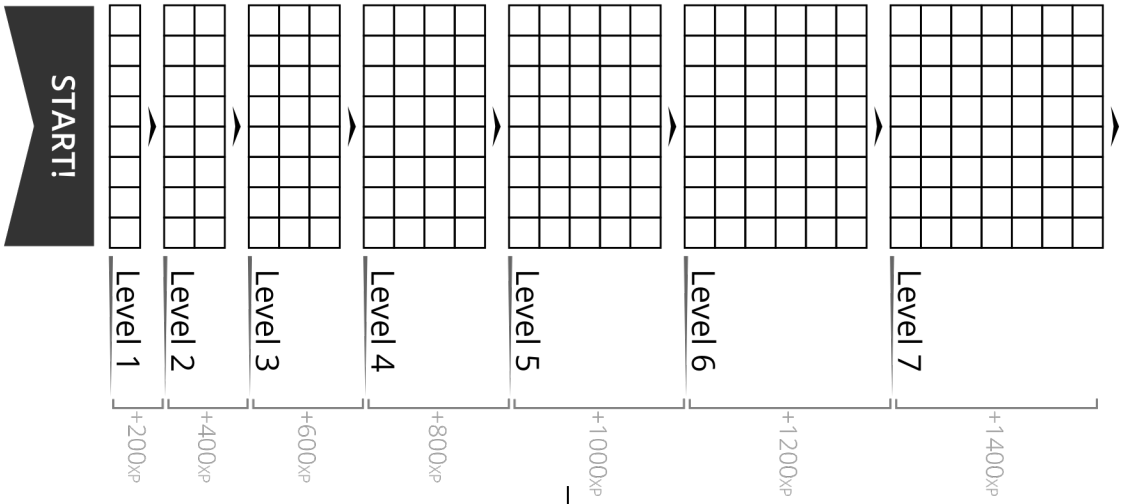
RB Whitaker
rbwhitaker@outlook.com

ISBN-13: 978-0-9855801-4-8


Start!
C# Programming Language
Getting an IDE
Hello World!
Comments
Types
Variables
PROCEDURAL PROGRAMMING
Math
Arrays
Switches
Looping
Methods
Console 2.0
Decision Making
Memory Management
Classes
Object-Oriented Concepts
Properties
Enumerations
Static
Information Hiding
Tuples
Null References
OBJECT-ORIENTED PROGRAMMING
Object-Oriented Design
The Catacombs of the Class
Structs
Generic Types
Inheritance
Some Useful Types
Polymorphism
Interfaces
The Fountain of Objects


THE
REALMS
OF
C#
UNCHARTED
LANDS
The Final Battle
THE ENDGAME
Compiler Errors
Debugging
APPENDIX
Unsafe Code
Other C# Features
Dynamic Objects
Asynchronous Programming
Threads
Multiple Projects
Query Expressions
Compiling
.NET
Publishing
ADVANCED FEATURES
Operator Overloading
Events
Lambda Expressions
Delegates
Pattern Matching
Files
Namespaces
Exceptions
Methods Revisited

Experience Points Progress Bar

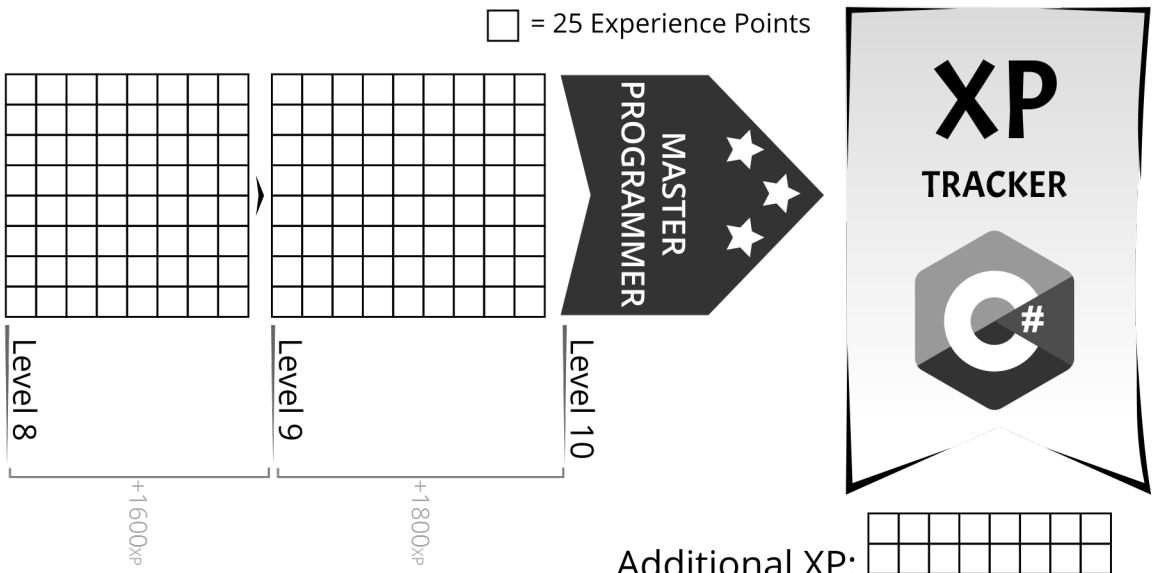


Part 1: The Basics

✓	Page	Name	XP	/	☐
☐	10	Knowledge Check - Level 1	25	/	1
☐	14	Install Visual Studio	75	/	3
☐	19	Hello World!	50	/	2
☐	21	What Comes Next	50	/	2
☐	22	The Makings of a Programmer	50	/	2
☐	24	Consolas and Telim	50	/	2
☐	28	The Thing Namer 3000	100	/	4
☐	34	Knowledge Check - Level 5	25	/	1
☐	42	The Variable Shop	100	/	4
☐	43	The Variable Shop Returns	50	/	2
☐	45	Knowledge Check - Level 6	25	/	1
☐	49	The Triangle Farmer	100	/	4
☐	52	The Four Sisters and the Duckbear	100	/	4
☐	53	The Dominion of Kings	100	/	4
☐	64	The Defense of Consolas	200	/	8
☐	71	Repairing the Clocktower	100	/	4
☐	73	Watchtower	75	/	3
☐	77	Buying Inventory	100	/	4
☐	78	Discounted Inventory	50	/	2
☐	83	The Prototype	100	/	4
☐	84	The Magic Cannon	100	/	4
☐	89	The Replicator of D'To	100	/	4
☐	90	The Laws of Freach	50	/	2
☐	100	Taking a Number	100	/	4
☐	101	Countdown	100	/	4
☐	116	Knowledge Check - Level 14	25	/	1
☐	117	Hunting the Manticore	250	/	10

Part 2: Object-Oriented Programming

✓	Page	Name	XP	/	☐
☐	123	Knowledge Check - Level 15	25	/	1
☐	126	Simula's Test	100	/	4
☐	135	Simula's Soups	100	/	4
☐	145	Vin Fletcher's Arrows	100	/	4
☐	153	Vin's Trouble	50	/	2
☐	160	The Properties of Arrows	100	/	4
☐	164	Arrow Factories	100	/	4
☐	183	The Point	75	/	3
☐	183	The Color	75	/	3
☐	183	The Card	100	/	4
☐	184	The Locked Door	100	/	4
☐	184	The Password Validator	100	/	4
☐	185	Rock-Paper-Scissors	150	/	6
☐	186	15-Puzzle	150	/	6
☐	186	Hangman	150	/	6
☐	187	Tic-Tac-Toe	300	/	12
☐	197	Packing Inventory	150	/	6
☐	201	Labeling Inventory	50	/	2
☐	202	Carrying Water	75	/	3
☐	209	The Old Robot	100	/	4
☐	217	Room Coordinates	50	/	2
☐	221	War Preparations	100	/	4
☐	230	Colored Items	100	/	4
☐	232	The Fountain of Objects	500	/	20
☐	234	Small, Medium, or Large	100	/	4
☐	234	Pits	100	/	4
☐	234	Maelstroms	100	/	4
☐	235	Amaroks	100	/	4
☐	235	Getting Armed	100	/	4
☐	236	Getting Help	100	/	4
☐	239	The Robot Pilot	50	/	2
☐	241	Time in the Cavern	50	/	2
☐	245	Lists of Commands	75	/	3



Additional XP:

Part 3: Advanced Features

✓	Page	Name	XP	/	☐
☐	260	Knowledge Check - Level 33	25	/	1
☐	261	The Feud	75	/	3
☐	261	Dueling Traditions	75	/	3
☐	267	Safer Number Crunching	50	/	2
☐	269	Knowledge Check - Level 34	25	/	1
☐	269	Better Random	100	/	4
☐	281	Exepti's Game	100	/	4
☐	286	The Sieve	100	/	4
☐	292	Knowledge Check - Level 37	25	/	1
☐	293	The Automatic Tree Harvester	75	/	3
☐	298	Knowledge Check - Level 38	25	/	1
☐	298	The Lambda Sieve	50	/	2
☐	306	The Long Game	100	/	4
☐	314	The Potion Masters of Pattren	150	/	6
☐	322	Knowledge Check - Level 41	25	/	1
☐	322	Navigating Operand City	100	/	4
☐	323	Indexing Operand City	75	/	3
☐	323	Converting Directions to Offsets	50	/	2
☐	332	Knowledge Check - Level 42	25	/	1
☐	333	The Three Lenses	100	/	4
☐	340	The Repeating Stream	150	/	6
☐	350	Knowledge Check - Level 44	25	/	1
☐	350	Asynchronous Random Words	150	/	6
☐	351	Many Random Words	50	/	2
☐	356	Uniter of Adds	75	/	3
☐	357	The Robot Factory	100	/	4
☐	363	Knowledge Check - Level 46	25	/	1
☐	384	Knowledge Check - Level 47	25	/	1
☐	389	Colored Console	100	/	4
☐	390	The Great Humanizer	100	/	4
☐	395	Knowledge Check - Level 49	25	/	1
☐	400	Knowledge Check - Level 50	25	/	1
☐	405	Altar of Publication	75	/	3

Part 4: The Endgame

✓	Page	Name	XP	/	☐
☐	411	Core Game: Building Character	300	/	12
☐	411	Core Game: The True Programmer	100	/	4
☐	412	Core Game: Actions and Players	300	/	12
☐	412	Core Game: Attacks	200	/	8
☐	413	Core Game: Damage and HP	150	/	6
☐	414	Core Game: Death	150	/	6
☐	414	Core Game: Battle Series	150	/	6
☐	414	Core Game: The Uncoded One	100	/	4
☐	414	Core Game: The Player Decides	200	/	8
☐	415	Expansion: The Game's Status	100	/	4
☐	416	Expansion: Items	200	/	8
☐	416	Expansion: Gear	300	/	12
☐	417	Expansion: Stolen Inventory	150	/	6
☐	417	Expansion: Vin Fletcher	150	/	6
☐	418	Expansion: Attack Modifiers	150	/	6
☐	418	Expansion: Damage Types	150	/	6
☐	419	Expansion: Making it Your Own	?	/	?
☐	420	Expansion: Restoring Balance	100	/	4

Part 5: Bonus Levels

✓	Page	Name	XP	/	☐
☐	432	Knowledge Check - Bonus Level A	25	/	1
☐	437	Knowledge Check - Bonus Level B	25	/	1
☐	442	Knowledge Check - Bonus Level C	25	/	1

Table of Contents

Acknowledgments

I remember the day I published the first edition of this book. I remember thinking that I was done with the hard part, and I could tweak a paragraph here and add a chapter there to make new editions as the C# language progressed. I seriously underestimated how fast the C# community and the C# language itself could change. And I seriously underestimated how much I would learn myself, especially about how to teach complex programming topics. While this edition is philosophically the same as the first edition, I feel like I have rewritten the entire book from scratch. I don't think there is a single sentence that I didn't rework. In many ways, it feels like an entirely different book. In others, it is still the same book.

An undertaking like this does not happen alone.

I couldn't have ever finished this book without help.

I have had conversations with many readers over the years that have helped me take the right path. I especially need to thank those who participated in this edition's Early Access program and gave constructive criticism and feedback. Your efforts immensely improved the book.

I also need to thank my family. My parents' confidence and encouragement to do my best have caused me to do things I could never 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 was 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. She has been patient with me as I've done four editions of this book over the years. Without her, this book would still be a random scattering of files 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

THE GREAT GAME OF PROGRAMMING

I have a firmly held personal belief, grown from decades of programming: in a very real sense, programming is a game. At least, it can be *like* playing a game with the right mindset.

For me, spending a few hours programming—crafting code that bends these amazing computational devices to your will, to create worlds of living software—is entertaining and rewarding. For me, it competes with delving into the Nether in *Minecraft*, snatching the last Province card in *Dominion*, or taking down a reaper in *Mass Effect*.

I don't mean that programming is mindless entertainment. It is rarely that. Most of your time is spent puzzling out the right next step or figuring out why things aren't working as you expected. But part of what makes games engaging is that they are challenging. They require creativity, patience, and exploration. If you start your journey into programming with that mindset—that creating meaningful and useful software is a challenge and a puzzle and that it takes patience, curiosity, and creativity to work through—you will be far better off than 95% of the people who set out to program. Some days, it will feel like you are playing *Flappy Bird*, *Super Meat Boy,* or *Dark Souls*—all notoriously difficult games—but creating software is challenging in all the *right* ways.

If programming is like a game, then it is a massively-multiplayer, open-world sandbox game with role-playing elements. That is to say:

- **Massively multiplayer:** While you may tackle specific problems as an individual, the Internet and things like this book ensure you are never alone.
- **An open-world sandbox game:** You have few constraints placed on you; you can build what, when, and how you want.
- **Role-playing elements:** With practice, learning, and experience, you get better in the skills and tools you work with, going from a lowly Level 1 beginner to a master, sharpening your skills and abilities as you go.

If programming is to be fun or rewarding, then learning to program must be so as well. Rare is the book that can make learning complex technical topics anything more than tedious. This book attempts to do just that. If a spoonful of sugar can help the medicine go down, then there

must be some blend of eleven herbs and spices that will make even the most complex technical topic have an element of fun, challenge, and reward.

Over the years, strategy guides, player handbooks, and player's guides have been made for popular games. These guides help players learn and understand the game world and challenges they will encounter, provide time-saving tips and tricks, and help prevent players from getting stuck at any one place for too long.

This book seeks to guide people interested in learning to play the Great Game of Programming using C#.

This book skips the typical business-centric examples found in other books in favor of samples with a little more spice. Many are game-related, and many of the hands-on challenges involve building small games or slices of games. After all, these make the journey more entertaining and exciting. While C# is an excellent language for game development, this book is not specifically a C# game programming book. You will undoubtedly come away with ideas to try if that the path you choose, but this book is focused on becoming skilled with the C# language so that you can use it to build *any* type of program, not just games. (Most programmers are paid to make business-centric applications, web apps, and smartphone apps.)

This book focuses on console applications. Console applications are those text-based programs where the computer receives text input from the user and displays text responses in the stereotypical white text on a black background window. Admittedly, console applications are less exciting than a web app, mobile app, or game. (Though we will learn tricks that will add a bit of splash to console applications.)

Why not start with a more exciting type of application? The main reason is that regardless of if you want to build games, smartphone apps, web apps, or desktop apps, the *starting points* in those worlds already expect that you know much about programming and using the language. For example, I just looked over the starter code for a certain C# game development framework. In it, I can see that it demands you already know how to use advanced topics covered in Level 25 (inheritance), Level 26 (polymorphism), and Level 30 (generics) just to get started! While some people successfully dive right in and manage to stay afloat, it is usually wiser to build up your swimming skills in a lap pool before trying to swim across the raging ocean. Starting from the basics gives you the right foundation to build upon. It makes learning the specific things you need to develop specific applications go much faster and more smoothly. Few will be satisfied with just console applications, but spending a few weeks covering the basics here before moving on will make the process go much more smoothly.

BOOK FEATURES

To produce a fun and rewarding (or at least not dull and useless) book means adding some features that most programming books do not have. Let's look at a few of the book's features so that you know what to expect.

Speedruns

At the start of each level (chapter) is a Speedrun section that outlines the key points described in the level. It is not a substitute for going through the whole level in detail but is helpful in a handful of situations:

1. You're reviewing the material and want a reminder of the key points.
2. You are skimming to see if some level has information that you will need soon.
3. You are trying to remember which level covered some particular topic.

Challenges

Scattered throughout the book are hands-on challenges that give you a specific problem to work on. These start small early in the book, but some of the later ones are quite large. Each of these challenges is marked with the following icon:

I strongly recommend that you do these challenges. You don't beat a game by reading the player's guide. You don't learn to program by reading a book. You will only truly learn if you sit down and program.

I also recommend you do these challenges as you encounter them, instead of reading ten chapters and returning to them. The *read a little, program a little* model is far better at helping you learn fast.

I will also suggest that these challenges should not be the only things you program as you learn, especially if you are relatively new to programming in general. Half of your programming time should come from the challenges here, with the rest coming from your own imagination. Working on things of your own invention will be more exciting to you. But also, when you are in that creative mindset, you mentally explore the programming world better. You start to think about how you can apply the tools you have learned in new situations, rather than being told, "Go use this tool over here."

As you do that, keep in mind the size of the challenges you are inventing for yourself. If you are first learning how to draw, you don't go find millennia-old chapel ceilings to paint (or at least you don't expect them to turn out like the Sistine Chapel). Aim for things that push your limits a little but aren't intimidating. Keep in mind that everything is a bit harder than you initially expect. Don't be afraid to end up with a few garbage drawings in your sketchbook to continue the art analogy. It is fine to end up with a few programs where you say, "Well, at least I learned a thing or two from that experiment. Let's go try something else." They won't all be masterpieces, and that's fine.

If these specific challenges are not your style, then skip them. But please substitute them with something else. You will learn little if you don't sit down and write some code.

When a challenge contains a **Hint**, these are suggestions or possibilities, not things you must do. If you find a different path that works, go for it.

Some challenges also include things labeled **Answer this question**. I recommend writing out your answer. (Comments (Level 4) could be a good approach.) Our brains like to tell us it understands something without proving it does. We mentally skip the proof, often to our detriment. Writing it out ensures we know it. (And these questions usually only take a few seconds to answer.)

I have posted my answers to these challenges on the book's website, described later in this introduction. If you want a hint or want to compare answers, you can review what I did. Just because our solutions are different doesn't make yours bad or wrong. I make plenty of my own mistakes, have my own preferences for the various tools in the language, and have also been programming in C# for a long time. As long as you have a solution that works, you're in great shape.

Knowledge Checks

Some levels in this book focus on conceptual topics that are not well-tested by a programming problem. In these cases, instead of a Challenge problem, these levels will have a Knowledge Check, containing a quiz with true/false, multiple-choice, and short answer questions. The answers are immediately below the Knowledge Check, so you can see if you learned the key points right away. These are marked with the knowledge scroll icon below:

Experience Points and Levels

Since this book is a player's guide, I've attempted to turn the learning process into a game. Each Challenge and Knowledge Check section is marked in the top right with experience points (written as *XP*, as most games do) that you earn by completing the challenge. When you complete a challenge successfully, you can claim the XP associated with it and add it to your total. Towards the front of this book, after the title page and the map, is an XP Tracker. You can use this to track your progress, check off challenges as you complete them, and marking off your progress as you go.

You can also get extra copies of the XP Tracker on the book's website (below) if you do not want to write in your book, have a digital copy, or a used copy where somebody else has already marked it.

As you collect XP, you will accumulate enough points to level up from, all the way to Level 10. If you reach Level 10, you will have completed nearly every challenge in this book and should have a solid grasp of C# programming.

The XP assigned to each challenge is not random. Easier challenges have fewer points; harder challenges have more. While measuring difficulty is somewhat subjective, you can generally count on spending more time with challenges that have more points and get a larger reward for your efforts.

Narratives and the Plot

The challenges form a loose storyline that has you, the (soon to be) Master Programmer journeying through a land that has been stripped of the ability to program by the malevolent and amorphous Uncoded One. Using your growing C# programming skills, you will be able to help the land's inhabitants, fend off the Uncoded One's onslaught, and eventually face the Uncoded One in a final battle at the end of the book.

Even if this plot is not attractive to you, the challenges are still worth doing. Feel free to ignore the book-long storytelling if it isn't helpful for you.

While much of the book's plot is revealed in the Challenge descriptions themselves, there were places where doing so felt shoehorned. Narrative sections supplement the descriptions in the Challenge sections but have no purpose other than to advance this book-long plot. These are marked with the icon below:

If you are ignoring the plot, you can skip these sections. They do not contain information that helps you be a better C# programmer.

Side Quests

While everything in this book worth knowing (skilled C# programmers know all of it), some sections are more important than others. The less useful sections can reasonably be skipped in your first pass going through the book. These sections are marked as Side Quests, shown by the following icon:

These often deal with situations that are less common or less impactful. If you're pressed for time, these sections are safer to skip than the rest. However, I recommend coming back to them later if you don't get them the first time around.

Glossary

In addition to a programming language, programmers themselves have a mountain of jargon and terminology unique to the field. Understanding how programmers speak is a vast additional challenge for new programmers. To help you with this undertaking, I have carefully defined new terminology within the book as it arises and collected all of these new words and concepts into a glossary at the back of the book. Only the lucky few will remember all such words from seeing it defined once. Use the glossary to refresh your mind on any term you don't remember well.

The Website

This book has a website associated with it, which has a lot of supporting content. The main page for this book is at **http://csharpplayersguide.com/**. Among the things on the website is the following:

- **http://csharpplayersguide.com/solutions** Contains my solutions to all the Challenge sections in this book. My answer is not necessarily more correct than yours, but it can give you some thoughts on a different way to solve the problem and perhaps some hints on how to progress if you are stuck.
- **http://csharpplayersguide.com/solutions** also contains more thorough explanations for all of the Knowledge Checks in the book.
- **http://csharpplayersguide.com/errata** This page contains errata (errors in the book) that have been reported to provide clarity on what was meant. If you notice something that seems wrong or inconsistent, you may find a correction here.

I Want Your Feedback

I depend on readers like you to help me see how to make the book better. This book is much better because people who were once in your situation reached out to me and let me know what was working (or not) for them.

Naturally, I'd love to hear that you loved the book. But I need constructive criticism too. If there is a challenge that was too hard, a typo you found, a section that wasn't clear, or even that you felt an entire chapter or the whole book was bad, I want to hear it. I have gone to great lengths to make this book as good as possible, but I can make it even better for our fellow

programmers who follow in our footsteps with help from you. Don't hesitate to reach out to me, whether your feedback is good or bad!

I have many ways that you can reach out to me. Go to **http://csharpplayersguide.com/contact** to find a way that works for you.

An Overview

Let's take a peek at what this book covers. This book has five major parts:

- **Part 1—The Basics.** This first part covers a lot of the simplest elements of C# programming. It focuses on what programmers call procedural programming, including storing data, picking and choosing which lines of code to run, and creating reusable chunks of code.
- **Part 2—Object-Oriented Programming.** C# uses an approach called object-oriented programming to help you break down a large program into smaller pieces that are each responsible for a little slice of the whole program. These tools are essential as you begin building bigger programs.
- **Part 3—Advanced Topics.** While Parts 1 and 2 deal with the most critical elements of the C# language, there are various other language features that are worth knowing. This part consists of mostly independent topics. You can jump around and learn the ones that you feel are most important to you (or skip them all entirely, for a while). In some ways, you could consider all of Part 3 to be a big Side Quest, though you will be missing out on some cool C# features if you skip it all.
- **Part 4—The Endgame.** While hands-on challenges are scattered throughout the book, Part 4 consists of a single, extensive, final program that will test the knowledge and skills that you have learned. It will also wrap up the book, pointing you toward Lands Uncharted and where you might go after finishing this book.
- **Part 5—Bonus Levels.** The end of the book contains a few additional chapters that guide you on what to do when you don't know what else to do—dealing with compiler errors and debugging your code. After the bonus levels comes the glossary, some tables and charts that summarize important aspects of C# programming, and the index.

Please do not feel like you must read this book cover to cover to get value from it.

If you are new to programming, I recommend a slow, careful path through Parts 1 and 2, skipping the Side Quest sections and only advancing when you feel comfortable taking the next step.

After Part 2, you might continue your course through the advanced features of Part 3, or you might also choose to skim it to get a flavor for what else C# offers without going into depth on anything. Even if you skim or skip Part 3, you can still attempt the Final Battle in Part 4. The bonus levels in Part 5 will also be valuable to you any time after you finish Level (not Part) 3.

If you're making consistent progress and getting good practice in, it doesn't matter if you are progressing slowly. It isn't a race.

Things will be different if you are an experienced programmer, especially somebody already familiar with object-oriented programming or a language with a similar structure to C# (like Java, C++, and C). In that case, you will likely be able to race through Part 1 quickly, slow down only a bit in Part 2 as you learn how C# deals with object-oriented programming, and then spend most of your time in Part 3, learning the features that make C# stand out in the crowd.

Part 1

The Basics

The world of C# programming lies in front of you, waiting to be explored. In Part 1, we begin our adventure and learn the basics of programming in C#:

- Learn the main features of C# and .NET (Level 1).
- Install tools to allow us to begin programming in C# (Level 2).
- Write our first few programs and learn the basic ingredients of a C# program (Level 3).
- Annotate your code with comments (Level 4).
- Store data in variables (Level 5).
- Understand the type system (Levels 6).
- Do basic math (Level 7).
- Get input from the user (Level 8).
- Make decisions (Levels 9 and 10).
- Run code more than once in loops (Level 11).
- Make arrays, which contain multiple pieces of data (Level 12).
- Make methods, which are named, packaged, reusable bits of code (Level 13).
- Understand how memory is used in C# (Level 14).

Level 1

The C# Programming Language

Speedrun

- C# is a general-purpose programming language. You can make almost anything with it.
- C# runs on .NET, which is many things: a runtime that supports your program, a library of code to build upon, and a set of tools to aid in constructing programs.

Computers are amazing machines, capable of running billions of instructions every second. Yet computers have no innate intelligence and do not know what instructions will solve a problem independently. The people who can harness these powerful machines to solve meaningful problems are the wizards of the computing world we call programmers.

Humans and computers do not speak the same language. Human language is imprecise and open to interpretation. The binary instructions computers use, formed from 1's and 0's, are precise but very difficult for humans to use. Programming languages are the bridge between the two—precise enough for a computer to run and clear enough for a human to understand.

What is C#?

There are many programming languages out there, but C# is one of the few that is both widely used and very loved. Let's talk about some of its key features.

C# is a general-purpose programming language. Some languages solve only a specific type of problem. C# is designed to solve virtually any problem equally well. You can use it to make games, desktop programs, web applications, and smartphone apps, and more. However, C# is at its best when building applications (of any sort) with it. You probably wouldn't write a new operating system or device driver with it (though both have been done).

C# strikes a balance between power and ease of use. Some languages give the programmer more control than C#, but with more ways to go wrong. Other languages do more to ensure bad things can't happen by removing some of your power. C# tries to give you both power and ease of use and often manages to do both but always strikes a balance between the two when needed.

C# is a living language. It changes over time to adapt to a changing programming world. Programming as a practice has changed significantly in the 20 years since it was created. C# has evolved and adapted over time. At the time of publishing, C# is on version 9.0, with new major updates every year or two.

C# is in the same family of languages as C, C++, and Java, meaning that C# will be easier to pick up if you know any of those. After learning C#, learning any of those will also be easier. This book sometimes points out the differences between C# and these other languages for readers who may know them.

C# is a cross-platform language. It can run on every major operating system, including Windows, Linux, macOS, iOS, and Android.

This next paragraph is for veteran programmers; don't worry if none of this makes sense. (Most will make sense after this book.) C# is a statically typed, garbage collected, object-oriented programming language with imperative, functional, and event-driven aspects. When needed, it also allows for dynamic typing and unmanaged code in small doses.

WHAT IS .NET?

C# is built upon a thing called *.NET* (pronounced "dot net"). .NET is often called a framework or platform, but .NET is the entire ecosystem surrounding C# programs and the programmers that use it. For example, .NET includes a *runtime*, which is the environment your C# program runs within. Figuratively speaking, it is like the air your program breathes and the ground it stands on as it runs. Every programming language has a runtime of one kind or another, but the .NET runtime is extraordinarily capable, taking a lot of burden off of you as a programmer.

.NET also includes a pile of code that you can use in your program directly. This collection is called the *Base Class Library* (*BCL*). You can think of this like mission control supporting a rocket launch: a thousand people who each know their specific job well, ready to jump in and support the primary mission (your code) the moment they are needed. For example, you won't have to write your own code to open files or compute a square root because the Base Class Library can do this for you.

.NET includes a broad set of tools called a *Software Development Kit* (*SDK*) that makes programming life easier.

.NET also includes things to help you build specific kinds of programs like web, mobile, and desktop applications.

.NET is an ecosystem shared by other programming languages. Aside from C#, the three other most popular languages are Visual Basic, F#, and PowerShell. You could write code in C# and then use it in a Visual Basic program. Because of their shared ecosystem, there are plenty of similarities, and in some cases, I'll point these out.

Knowledge Check **Level 1** **25 XP**

Check your knowledge with the following questions:

1. **True/False.** C# is a special-purpose language optimized for making web applications.
2. What is the name of the framework that C# runs on?

Answers: (1) False. **(2)** .NET

Level 2

Getting an IDE

Speedrun

- Programming is complex; you want an IDE to make programming life easier.
- Visual Studio is the most used IDE for C# programming. Visual Studio Community is free, feature-rich, and recommended for beginners.
- Other C# IDEs exist, including Visual Studio Code and Rider.

Modern-day programming is complex and challenging, but a programmer does not have to go alone. Programmers work with an extensive collection of tools to help them get their job done. An *integrated development environment* (*IDE*) is a program that combines these tools into a single application, designed to streamline the programming process. An IDE does for programming what Microsoft Word does for word processing or Adobe Photoshop for image editing. Most programmers will use an IDE as they work.

There are several C# IDEs to choose from. (In fact, you can do without one and use the raw tools directly; I don't recommend that for new programmers.)

In this level, we will look at the most popular C# IDEs and discuss their strengths and weaknesses.

As we begin programming, we will use our IDE for many tasks. Unfortunately, every IDE does things differently, and this book cannot cover how to do every job in all possible IDEs. While this book's focus is on the C# language and not a specific IDE, this book will illustrate how to do a task in Visual Studio Community Edition when necessary. You can still feel free to use a different IDE. The C# language itself is the same regardless of which IDE you pick, but you may find some small differences when performing a task in the IDE. Usually, the process is intuitive, and if tinkering fails, a Google search usually finds the answer quickly.

A Comparison of IDEs

There are several notable IDEs that you can choose from.

Visual Studio

Microsoft Visual Studio is the stalwart, tried-and-true IDE that most C# developers use. Visual Studio versions go back even before C# existed, though it has grown up a lot since those days.

Of the IDEs we discuss here, this is the most feature-rich and capable, though it has one significant drawback: it works on Windows but not Mac or Linux.

Visual Studio comes in three different "editions" or levels: Community, Professional, and Enterprise. The Community and Professional editions have the same feature set, while Enterprise has an expanded set of features with some nice bells and whistles at extra cost.

The difference between the Community Edition and the Professional Edition is only in the cost and the license. Visual Studio Community Edition is free but is meant for students, hobbyists, open-source projects, and individuals, even for commercial use. Large companies do not fit into this category and must buy Professional. (If you have more than 250 computers, make more than $1 million annually, or have more than five Visual Studio users, you'll need to pay for Professional.)

Visual Studio Community edition is my recommended choice for new C# programmers running on Windows and is what this book uses throughout. (Though Professional and Enterprise would work in the same way.)

Visual Studio Code

Microsoft Visual Studio Code is a lightweight editor (a step down from a fully-featured IDE) that works on Windows, Mac, and Linux. Visual Studio Code is also free for everybody and has a vibrant and growing community. It does not have nearly the same expansive feature set that Visual Studio has, and in some places, the limited feature set is harsh; you sometimes have to fall back to running commands on the command line. If you are used to command-line interfaces, this cost is low. But if you're new to programming, it may feel like an alien world.

I recommend Visual Studio Code as a secondary choice to the full Visual Studio. Use it if you are on Linux or Mac and are comfortable with the command line. In these situations, the lightweight nature of Visual Studio Code is pleasant.

Visual Studio for Mac

Visual Studio for Mac, once called Xamarin Studio, is a separate IDE for C# programming that works on Mac. While it shares its name with Visual Studio, it is a different product with many notable differences. Like Visual Studio (for Windows), this has Community, Professional, and Enterprise editions. If you are on a Mac, this IDE is worth considering.

JetBrains Rider

The only non-Microsoft IDE on this list is the Rider IDE from JetBrains. Rider is a comparative newcomer to the C# IDE world, though JetBrains is a veteran of the IDE world, having built them for many other languages. JetBrains does not have a free tier as this book goes to publication; the cheapest option is about $140 per year. But it is both feature-rich and cross-platform. If you have the money to spend, this is a good choice on any operating system.

Other IDEs

There are other IDEs out there, but most C# programmers use one of the above. Other IDEs tend to be missing lots of features, aren't well supported, and have less online help and

documentation. But if you find another IDE that you enjoy, go for it. The C# language will work the same either way.

No IDE

You do not need an IDE to program in C#. If you are a veteran programmer, skilled at using the command line, and accustomed to patching together different editors and scripts to program in your own way, you can entirely skip the IDE. I do not recommend this approach for new programmers. It is a bit like needing to build your car from parts before you can drive it. For the seasoned mechanic, it may even be part of the enjoyment. Everybody else wants something that they can hop in and drive. The IDEs above are in that category.

In short, at a high level, to work without an IDE in C# requires using the **`dotnet`** command-line tool to create, compile, test, and package your programs. Even if you are using an IDE, you may still find this a useful technique from time to time. (If you use Visual Studio Code, you will *need* to use it from time to time.)

But if you are new to programming, start with an IDE and learn the basics first.

INSTALLING VISUAL STUDIO

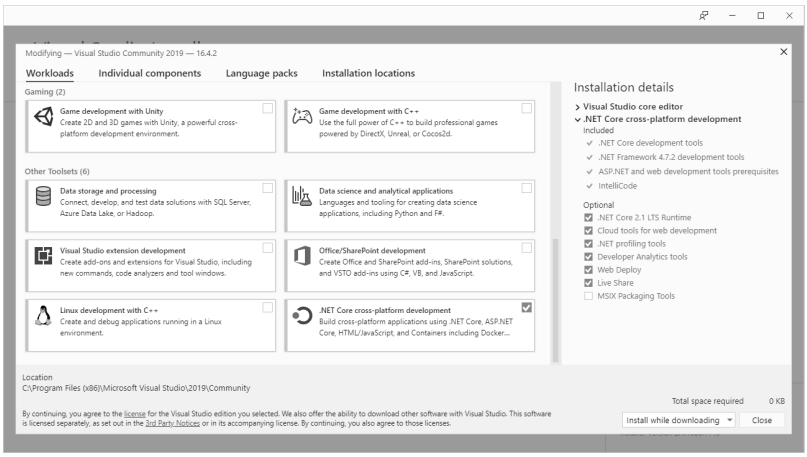
While this book's focus is the C# language itself, when I need to illustrate some tasks requiring an IDE, this book uses Visual Studio Community Edition. The Professional and Enterprise Editions should be identical. Other IDEs are usually similar, but you will find differences.

Visual Studio Code is popular enough that I posted an article on the book's website illustrating how to get started with Visual Studio Code: **http://csharpplayersguide.com/articles/visual-studio-code**.

You can download Visual Studio Community Edition from **https://www.visualstudio.com/downloads**. The installation process is not substantially different than any other program, so I'll spare you the details of walking through the installer here, with one notable point: the link above will install the Visual Studio *Installer* rather than Visual Studio itself. Once that is installed, you will need to use it to install and configure Visual Studio.

You can use the Visual Studio Installer to install Community, Professional, or Enterprise. I recommend just starting with Community, but this program manages all three of them.

As you begin installing Visual Studio, it will ask you which components to include:



With everything installed, Visual Studio is a lumbering, all-powerful behemoth. You do not need every possible feature of Visual Studio. In fact, for what we will do in this book, we will only need a small slice of what Visual Studio has to offer.

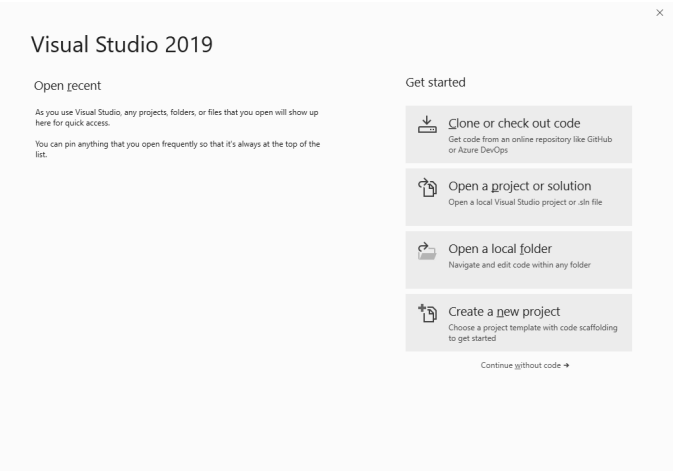
You can install whatever you find interesting (and have storage space for), but there is only one item you *must* install for the code in this book. On the **Workloads** tab, find the one towards the bottom named **.NET Core cross-platform development** and click on it to enable it. (If you forget to do this, you can always rerun the Visual Studio Installer and change what components you have installed.)

Once Visual Studio is installed, open it. (You may end up with a desktop icon, but you can always find it in the Windows Start Menu under Visual Studio 2019.)

Visual Studio will ask you to sign in with a Microsoft account, even for the free Community Edition. You can avoid it for 30 days, but it will require it eventually. If you don't have one, follow the instructions to make one. (It's free.) Creating an account lets you sync your settings across multiple devices, among other things.

If you are installing Visual Studio for your first time, you will also get a chance to pick development settings—keyboard shortcuts and a color theme. In this book, I have used the light theme because it looks clearer in print. Many developers are partial to the dark theme. Whatever you pick can be changed later.

You know you are done when you make it to the launch screen shown below:



Challenge **Install Visual Studio** **75 XP**

As your journey begins, you must start by getting the tools ready to start programming in C#. Install Visual Studio Community edition (or another IDE) and get it ready to start programming.

LEVEL 3

HELLO WORLD: YOUR FIRST PROGRAM

Speedrun

- New projects usually begin life by being generated from a template.
- A C# program starts running in the program's entry point or main method.
- A full Hello World program looks like this: `System.Console.WriteLine("Hello World!");`
- Statements are single commands for the computer to perform. They run one after the next.
- Expressions allow you to define a value that is computed as the program runs from other elements.
- Variables let you store data for use later.
- Console.ReadLine() retrieves a full line of text that a user types from the console window.

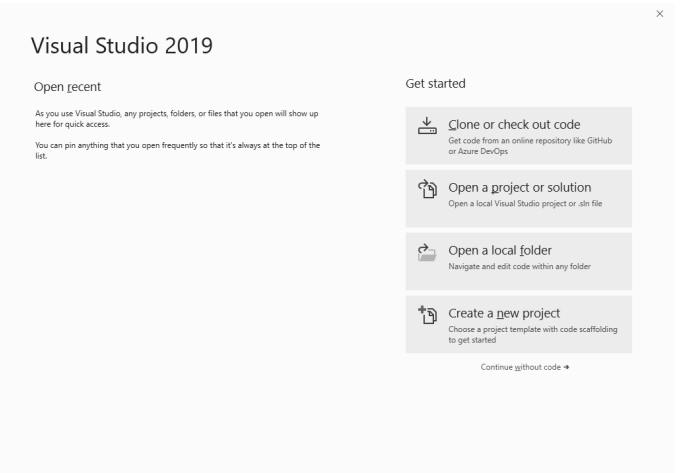
Our adventure begins in earnest in this level, as we make our first real programs in C# and learn the basics of the language. We'll start with a simple program called *Hello World*, the classic first program to write in any new language. It is about the smallest possible program we could make. It gives us a glimpse into what the language looks like and verifies that we got everything installed. Anything else would make the programming gods mad, and we don't want that!

CREATING A NEW PROJECT

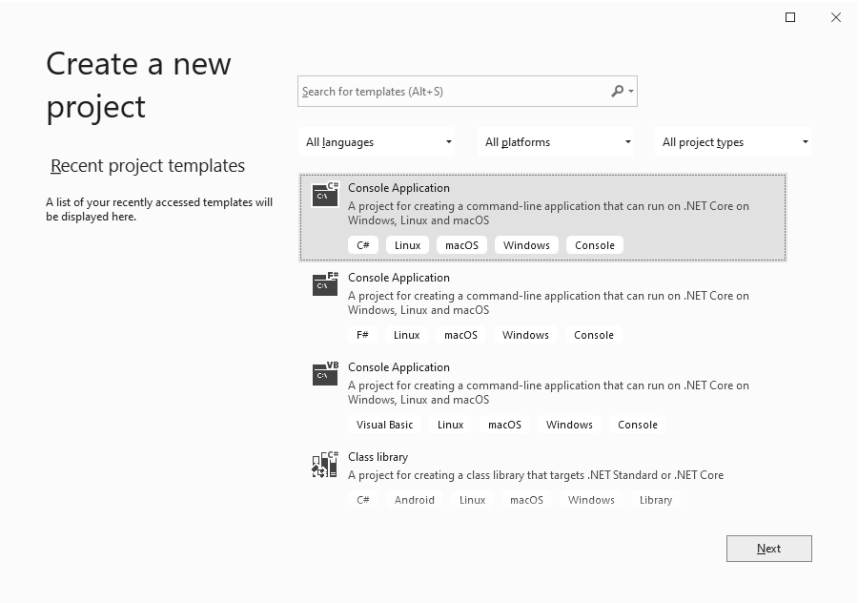
A C# project is a combination of two things. The first is your C# *source code*—instructions you write in C# for the computer to run. The second is configuration—instructions you give to the computer to help it know how to compile or translate C# code into the binary instructions the computer can run. Both of these live in simple text files on your computer. C# source code files use the *.cs* extension. A project's configuration uses the *.csproj* extension. Because these are both simple text files, we could handcraft them ourselves if we needed to.

But most C# programs begin their life by being generated from one of several *templates*. Templates are standard starting points; they help you get the configuration right for specific project types and give you some starting code. We will use a template to create our projects.

Start Visual Studio so that you can see the launch screen below:



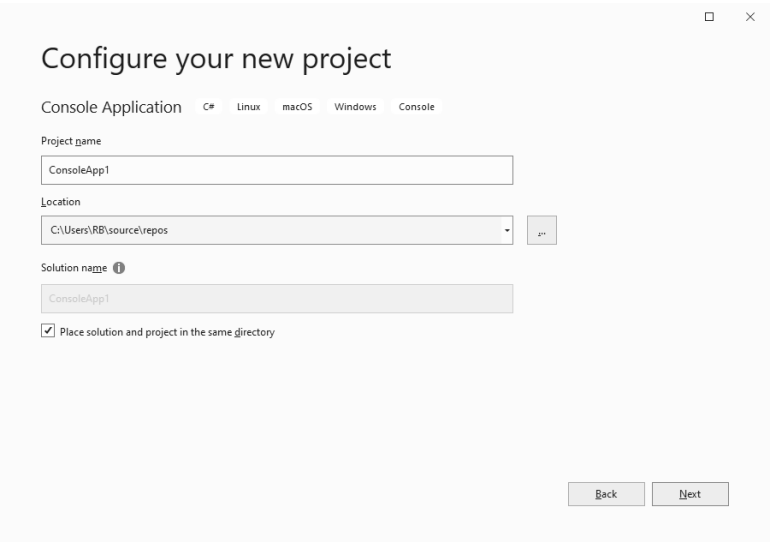
Click on the **Create a new project** button on the bottom right. Doing this advances you to the **Create a new project** page:



There are many templates to choose from, and yours might not be a perfect match for what you see above. For this book, we will always select the **Console Application** template. Be careful! You want to make sure that the one you chose is the C# version (look at the tags) and also not the one labeled Console Application (.NET Framework), which is an older template.

As you make progress in the C# world, you will use other templates.

After choosing the C# **Console Application** template, press the **Next** button to advance to a page that lets you enter your new program's details:



Always give your projects a good name. You won't remember what ConsoleApp12 did in two weeks.

For the location, pick a spot that you can find later on. (The default location is fine, but it isn't a prominent spot, so take note of where it is.)

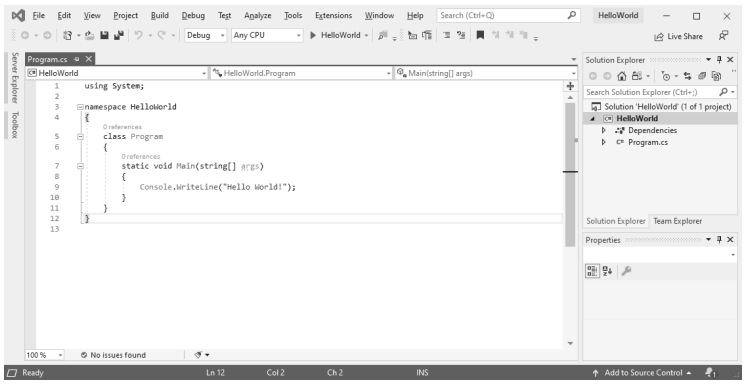
There is also a checkbox for **Place solution and project in the same directory**. For small projects, I recommend checking this box. Larger programs (solutions) may be formed from many projects. In that case, putting projects in their own directory (folder) under a solution directory makes sense. But for small programs with a single project, it is simpler just to put everything in a single folder.

Press the **Next** button to choose your target framework on the final page. You will want to pick **.NET 5** (or newer, if one is available) for this book.

Once you have chosen the framework, push the **Create** button to create the project.

A Brief Tour of Visual Studio

With a new project created, we get our first glimpse at the Visual Studio window:



Visual Studio is extremely capable, so there is much to explore. This book focuses on programming in C#, not on becoming a Visual Studio expert; we won't get into every detail of

Visual Studio here. However, we will cover some of the essential elements that help get the job done as we go.

Right now, there are three things you need to know to get going. First, the big open area on the left side with text is the Code Window or the Code Editor. You will spend most of your time working here. Second, on the right side is the Solution Explorer. That shows you the high-level view of your code and the configuration needed to turn it into working code. You will spend only a little time here initially, but you will use this more as you begin to make larger programs. Third, we will run our programs using the part of the Standard Toolbar shown below:

Bonus Level A covers Visual Studio in a bit of depth. You can read that level (and the other bonus levels) whenever you are ready for it. Even though they are at the end of the book, they don't require knowing everything else before them. If you're new to Visual Studio, I recommend reading Bonus Level A before getting through too many more levels. It will give you a better feel for Visual Studio.

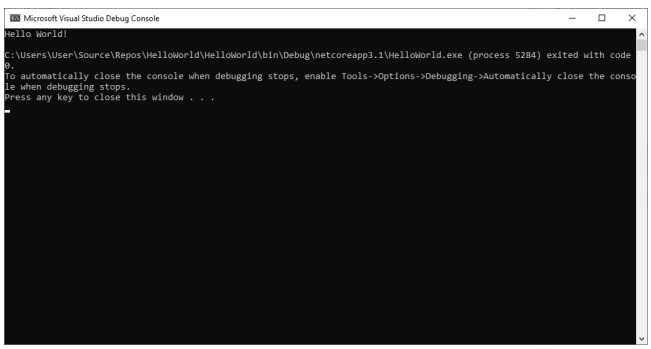
Compiling and Running Your Program

Generating a new project from the template has produced a complete program. Before we start dissecting it, let's run it.

The computer's circuitry cannot run C# code itself. It only runs low-level binary instructions formed out of 1's and 0's. Before the computer can run our program, we must transform it into something it can run. This transformation is called *compiling*, done by a special program called a *compiler*. The compiler takes your C# code and your project's configuration and produces the final binary instructions that the computer can run directly. (This is an oversimplification, but it's accurate enough for now.) The compiler creates an *.exe* or *.dll* file as a result, which the computer can run.

Visual Studio makes it easy to compile and then immediately run your program with any of the following: (a) choose **Debug > Start Debugging** from the main menu, (b) press **F5**, or (c) push the green start button on the toolbar, shown below:

When you run your program, you will see a black and white console window appear:



Look at the first line:

```
Hello World!
```

That's what our program was supposed to do! (The rest of the text just tells you that the program has ended and gives you instructions on how not to show it in the future. You can ignore that text for now.)

Challenge — Hello World! — 50 XP

You open your eyes and find yourself face down on the beach of a large island, the waves crashing on the shore not far off. A voice nearby calls out, "Hey, you! You're finally awake!" You sit up and look around. Somehow, opening your IDE has pulled you into the Realms of C#, a strange and mysterious land where it appears that you can use C# programming to solve problems. The man comes closer, examining you. "Are you okay? Can you speak?" Creating and running a "Hello World!" program seems like a good way to respond.

Objectives:

- Create a new Hello World program from the C# **Console Application** template, targeting **.NET 5**.
- Run your program using any of the three methods described above.

THE ADVENTURE BEGINS

Let's look at how to begin building C# programs. Every programming language has its own distinct structure—its own set of rules that describe how to make a working program in that language. This set of rules is called the language's *syntax*. Learning the structure of C# programs is covered on every page of this book, but this section gets us going by looking at the most foundational structural elements.

Every C# program has a starting point—the place where the program begins running. This location is called the program's *entry point* or *main method*. C# has two ways to define an entry point. One is the traditional way, which is what the template used. If you look at the code, you'll see a bunch of stuff that surrounds the central **`"Hello World!"`**, including that whole **`public static void Main(string[] args)`** thing. There is a lot of code out there that uses this traditional approach because it was the original way. There is a newer way that is streamlined and removes the clutter.

We will learn how to use the traditional way in Level 33, but the new way is simpler and easier to understand. It is what we will use as we begin.

Take a glance at the code generated from the template so that you can spot it when you see it in the wild, but then select all of the text and delete it, replacing it with just this single line:

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

This one line does *exactly* what all 12 of the previous lines did, just cleaner.

Let's pick this code apart. The **`"Hello World!"`** part is self-explanatory. That is the text that is displayed. But what about the rest?

`System`, **`Console`**, and **`WriteLine`** are all formally known as *identifiers*, or more casually as *names*. Identifiers are used to name new things you create and to refer to things that were previously created. **`System`**, **`Console`**, and **`WriteLine`** are each previously created things. We didn't make any of these; they are each part of the supporting cast of stuff you can build upon found in .NET's standard library, the Base Class Library.

Some things can contain other things. The thing inside something else is referred to as a member of the container. The period symbol (`.`) is called the *member access operator* or the *dot operator*. You use this to access a member contained in something else. So the thing called `System` has a member called `Console`, which has a member called `WriteLine`, and our code uses the dot operator to dig down from the top. We will use the dot operator a lot.

This shows C#'s organizational structure, but `System`, `Console`, and `WriteLine` are in very different categories.

`System` is a *namespace*. It contains other things under a shared name and is primarily an organizational tool, almost like a directory or folder in real life or a computer's file system. We'll learn more about namespaces as we go, and Level 33 covers them in detail.

`Console` is a *class*. We will soon see how vital classes are to building C# programs (all of Part 2 focuses on this). You can think of a class as a single supporting entity that focuses on solving one problem well. It combines the data it needs to perform its job and provides tasks other parts of the system can ask it to do. The `Console` class's focus is on interaction with the console window.

`WriteLine` is one such task—one that takes text and displays it in the console window on its own line. Tasks like `WriteLine` are called *methods*. Each method contains other code that will run when something asks for the task to be performed. Methods are a powerful tool because we can define what code is needed to complete the job once and then ask the task to run whenever we need it. Parentheses are used (among other things) to tell a method to run. This is called *method invocation* or *calling a method*. The parentheses of our `WriteLine("Hello World!")` code do this.

Some methods allow you to supply some information for it to use when running the task. If a method expects this, that information will be placed inside the parentheses. `WriteLine` happens to expect this in the form of the text you want to be displayed. (Some methods let you supply multiple pieces of information. We'll see those soon.)

Thus the line `System.Console.WriteLine("Hello World!");` finds the namespace `System`, finds its member class called `Console`, finds its member method called `WriteLine`, and asks it to run with the text `"Hello World!"`.

This line constitutes what is called a *statement*. A statement is a single command to perform. Most kinds of statements in C# need to end with a semicolon (`;`), as we see above.

C# executes statements top to bottom and left to right (though C# programmers typically put one statement per line), one at a time. Statements are an essential building block.

One thing that may surprise new programmers is how specific you need to be when giving the computer statements to run. Most humans can be given vague directives and then make judgment calls to fill in the gaps. Computers have no such capacity. They do exactly what they are told without variation. If it does something unexpected, then what you thought you commanded and what you thought you commanded were not the same. Making the mental shift from "The computer did something dumb" to "That was unexpected; why did it do that instead of what I thought it would do?" is one of the most important abilities for a new programmer to gain.

Fortunately for us, somebody already figured out how to instruct the computer to display text in the console window and packaged it up in the pre-made bundle that is `Console`'s `WriteLine` method. Leveraging other people's code can save us a lot of time.

C# ignores whitespace (spaces, tabs, newlines) as long as it can tell where one thing ends and the next begins. We could have written the above line like this, and the compiler wouldn't care:

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

But which is easier for *you* to read? This is a critical point about writing code: **You will spend more time reading code than writing it. Do yourself a favor and go out of your way to make code easy to understand, regardless of what the compiler is willing to allow.**

`using` Directives

Let's start making modifications to our simple Hello World program.

If we want to refer to a class to access its methods, we would generally need to refer to it through its namespace. We can't just refer to **`Console`**, but we must refer to it as **`System.Console`**. If we use **`Console`** a bunch in a program, this will get annoying. It is like referring to everybody you know by their full name all the time.

C# has a special type of statement that we can put at the top of a file to sidestep this. This statement is called a **`using`** directive because it uses the word using. Here is an example:

```
using System;
```

For any namespace we repeatedly use in a C# source code file, we can add a **`using`** directive for it at the top of the file and then skip the namespace later on in the file.

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

When the compiler encounters the identifier **`Console`**, it knows it should search in the **`System`** namespace to see if it can find something by that name. We can skip the long **`System.Console`** name—called a *fully qualified name*—and use a class's simple name. We will start using other things in the **`System`** namespace as well, and a single using **`System`** can work for all of them at once.

We will use the **`System`** namespace a lot. For the rest of this book, you can assume that all code samples are contained in a file that starts with **`using System;`**.

You may notice that when you type **`using`** into a C# code file, it changes colors (blue in most color themes). That is because **`using`** is a *keyword*, which is a special word that the language and compiler use to understand your intent. C# has over 100 keywords.

Challenge **What Comes Next** **50 XP**

The man seems surprised that you've produced a working "Hello World!" program. "Been a while since I saw somebody program like that around here. Do you know what you're doing with that? Can you make it do something besides just say 'hello'?"

Build on your original Hello World program with the following:

Objectives:

- Replace the file's contents with a **`using System;`** and **`Console.WriteLine("Hello World!");`**

- Change your program to say something (anything!) besides "Hello World!"

Multiple Statements

A C# program runs one statement at a time in the order they appear in the file. Putting multiple statements into your program makes it do multiple things. The following code displays three lines of text:

```
using System;

Console.WriteLine("Hi there!");
Console.WriteLine("My name is Dug.");
Console.WriteLine("I have just met you and I love you.");
```

Each line asks the **`Console`** class to perform its **`WriteLine`** method, just with different data on each line. Once all statements in the program have been completed, the program ends.

Challenge The Makings of a Programmer 50 XP

The man, who tells you his name is Ritlin, asks you to follow him over to a few of his friends, fishing on the dock. "This one here has the makings of a Programmer!" Ritlin says. The group looks at you with eyes widening and mouths agape. Ritlin turns back to you and continues, "I haven't seen nor heard tell of anybody who can wield that power in a million clock cycles of the CPU. Nobody has been able to do that since the Uncoded One showed up in these lands." He describes the shadowy and mysterious Uncoded One, an evil power that rots programs and perhaps even the world itself. The Uncoded One's presence has prevented anybody from wielding the power of programming, the only thing that might be able to stop it. Yet somehow, you have been able to grab hold of this power anyway. Ritlin's companions suddenly seem doubtful. "Can you show them what you showed me? Use some of that Programming of yours to make a program? Maybe something with more than one statement in it?"

Objectives:

- Make a program with 5 **`Console.WriteLine`** statements in it.
- **Answer this question:** How many statements do you think a program can contain?

Expressions

Our next building block is an *expression*. Expressions are bits of code that your program must process or *evaluate* to determine what their result is. We use the same word in the math world to refer to something like 3 + 4 or -2 × 4.5. Essentially, expressions let you define a value from parts assembled into the final value as the program is running.

C# programs use expressions heavily. Anywhere that a value is needed, an expression can be put in its place. Your program will first evaluate the expression and then use the result of that evaluation in its place. While we do could this:

```
Console.WriteLine("Hi User");
```

We can also use an expression instead:

```
Console.WriteLine("Hi " + "User");
```

The code **`"Hi " + "User"`** is an expression rather than a single value. As your program runs, it will evaluate the expression to determine its value. This code shows that you can use **`+`** between two text pieces to produce the combined text (**`"Hi User"`**).

The **`+`** symbol is one of many operators that can be used to build expressions. We will learn more as we go.

One thing that makes expressions powerful is how they can be built out of other expressions. Most expressions are formed from parts that are expressions themselves. You can think of a specific value like **`"Hi User"`** as the simplest type of expression. But if we wanted, we could split **`"User"`** into **`"Us" + "er"`** or further still into **`"U" + "s" + "e" + "r"`**. That isn't very practical, but it does illustrate how you can build expressions out of smaller expressions. Simpler expressions are better than complicated ones that do the same job, but you have lots of flexibility when you need it.

Variables

Variables are another essential building block of C# programs. Variables are containers for data. They are called variables because their contents can change or vary as the program runs. Before we can use a variable, we must indicate that we need one. This is called *declaring* the variable. In doing so, we must provide a name for the variable and indicate its type. Once a variable exists, we can place values in the variable to use later in the program. Doing so is called *assignment*, or assigning a value to the variable. Once we have done that, we can use the variable in expressions later on. All of this is shown below:

```
string name;
name = "User";
Console.WriteLine("Hi " + name);
```

The first line declares the variable with a type and a name. Its type is **`string`**, and its name is **`name`**. We'll get into more detail on variables in Level 5, but a *string* is just a fancy programmer word for text. So this **`name`** variable can contain strings (text). The second line assigns it a value of **`"User"`**. The third line uses the variable in an expression. As your program runs, it will evaluate the expression **`"Hi " + name`** by retrieving the current value out of the name variable, then combining it with the constant value of **`"Hi "`**. We'll see plenty more examples of expressions and variables soon.

Expressions are fragments of code. Expressions are not statements, though most types of statements can use expressions within them. Some statements are also expressions. Expressions are an important building block in a C# program.

Reading Text from the Console

Some methods produce a result as a part of the job they were designed to do. This result can be stored in a variable or used in an expression. For example, **`Console`** has a **`ReadLine`** method that retrieves text that somebody types until they hit the Enter key. It is used like so:

```
Console.ReadLine()
```

`ReadLine` does not require any information to do its job, so the parentheses are empty. But the result it produces can be used in other things. For example, this code stores the produced value in the **`name`** variable:

```
string name;
name = Console.ReadLine();
Console.WriteLine("Hi " + name);
```

This code no longer displays the same text every time. It waits for the user to type in their name and then displays a greeting to the user by name. (If you run this, you will see a blank screen. Don't be alarmed; it is merely waiting for you to enter your name. It would be better if we had used **`WriteLine`** to display something like, "Enter your name:" first, so the user knew what we were waiting for.)

When a method like this produces a value, programmers often say that it *returns* the value. So you might say that **`Console.ReadLine()`** returns the text the user typed.

Challenge **Consolas and Telim** **50 XP**

These lands have not seen Programming in a long time due to the blight of the Uncoded One. Even old programs once made have crumbled to bits. Your skills with Programming are only fledgling now, but you can still make a difference in these people's lives. Maybe someday soon, your skills will have grown strong enough to take on the Uncoded One directly. But for now, you decide to do what you can.

In the nearby city of Consolas, food is running short. Telim has a magic oven that can produce bread from thin air. He is willing to share, but Telim is an Excelian, and Excelians love paperwork; they demand it for all transactions—no exceptions. Telim will share his bread with the city if you can build a program that will let him enter the names of those receiving it. A sample run of this program looks like this:

```
Bread is ready.
Who is the bread for?
RB
Noted: RB got bread.
```

Objectives:

- Make a program that runs as shown above, including taking a name from the user.

Compiler Errors, Debuggers, and Configurations

There are a few loose ends that we should tie up before we move on and dive deeper into the C# language. These are more about how programmers construct C# programs rather than the language itself. These are compiler errors, debugging, and build configurations.

Compiler Errors

As you write C# programs, you will accidentally write some code that the compiler cannot figure out. When this happens, the compiler can not transform your code into something the computer can understand.

When this happens, you will see two things occur. When you try to build and run your program, you will see the Error List window appear, listing problems that the compiler sees. Double-clicking on an error takes you to the problematic line. You will also see broken code underlined with a red squiggly line. You may even see this appear as you type.

You may often see the problem quickly, but other times, it may not be so obvious. Bonus Level B provides suggestions for what to do when you cannot get your program to compile. As with all of the bonus levels, feel free to jump over and do it whenever you have an interest or need. You do not need to wait until you have completed all the levels before it.

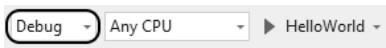
Debugging

Writing code that the compiler can understand is only the first step. It also needs to do what you expected it to do. Trying to figure out why a program does not do what you expected and then adjusting it is called *debugging*. It is a skill that takes practice, but Bonus Level C will show you the tools you can use in Visual Studio to make this task less intimidating. Like the other bonus levels, you can jump over and read them whenever you have an interest or a need without reading everything before that.

Build Configurations

The compiler uses your source code and configuration data to produce software the computer can run. In the C# world, configuration data is organized into different build configurations. Each configuration provides different information to the compiler about how to build things. By default, there are two configurations defined, and you don't often need more. Those configurations are the Debug configuration and the Release configuration. The two are mostly the same. The main difference is that the Release configuration has optimizations turned on, which allow the compiler to make certain adjustments so that your code can run faster without changing what it does. (As an example, if you declare a variable and never use it, optimized code will strip it out. Unoptimized code will leave it in.) The Debug configuration has this turned off. When you are debugging your code, these optimizations can make it harder to hunt down problems. As you are building your program, it is usually better to run with the Debug configuration. When you're ready to share your program with others, you compile it with the Release configuration instead.

You can choose which configuration you're using by picking it from the toolbar's dropdown list, near where the green arrow button is to start your program.



Level 4

Comments

Speedrun

- Comments let you put text in a program that the computer ignores, but that helps programmers understand or remember what the code does.
- Anything after two slashes (`//`) on a line is a comment, as is anything between `/*` and `*/`.

Comments are little bits of text placed into your program, meant to be annotations on the code for yourself or other programmers, which the compiler ignores.

Comments have a variety of uses:

- You can add a description about how some tricky piece of code works, so you don't have to try to reverse engineer it from your code again.
- You can leave reminders in your code of things you still need to do. These are sometimes called TODO comments.
- You can add documentation about how some specific thing should be used or works. Documentation comments like this can be handy because somebody (even yourself) can look at a piece of code that is meant to be reused in your program and know how it is supposed to work without looking through every line of code.
- They are sometimes used to remove code from the compiler's view temporarily. For example, suppose some code is not working, and you want to remove the offending code temporarily. In that case, you can turn the code into a comment until you are ready to bring it back in. (This should only be temporary! Don't leave large chunks of commented-out code hanging around!)

There are three ways to put in a comment, though we will only discuss two of them here and save the third for later.

You can start a comment anywhere within your code by placing two forward slashes (`//`). Everything on the line after these two slashes will become a comment, which the compiler will pretend doesn't even exist. For example:

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

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

Some programmers have strong preferences for each of the two placements. My general rule is to put more important comments above the code itself and only use comments that share their line with code for side notes about that line of code.

As the second style of comments, placing something between the slash and star combination of **/*** and ***/** will also make it a comment:

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

You can use this to make both multi-line comments and embedded comments:

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

Console.WriteLine("Hi " /* Here comes the good part! */ + userName);
```

That second example is awkward but does have its uses (especially when you're commenting out code that you want to ignore temporarily).

Of course, you can make multi-line comments with double-slash comments; you just have to put the slashes on every line. Many C# programmers prefer double-slash comments over multi-line **/*** and ***/** comments, but both are common.

HOW TO MAKE GOOD COMMENTS

The mechanics of adding comments are simple enough. The real challenge is in making meaningful comments.

My first suggestion is not to let TODO or reminder comments (often in the form of **`// TODO: Some message here`**) or commented out code last long. Both are meant to be short-lived. They have no long-term benefit and only clutter the code.

Second, don't say things that can be quickly gleaned from the code itself. The first comment below adds no value, while the second one does:

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

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

The second comment explained *why* this was done, which isn't apparent from looking at the code itself.

Third, write comments roughly at the same time as you write the code. You will never remember what the code did three weeks from now, so don't wait to describe what it does.

Fourth, find the balance in how much you comment. It is entirely possible to add both too few and too many comments. If you can't make sense of your code when you revisit it after a couple of weeks, you probably aren't commenting enough. If you keep discovering that

comments have gotten out of date, it is sometimes an indication that you are using too many comments or putting the wrong information in comments. (Some corrections are expected as code evolves.) As a new programmer, the consequences of too few comments are usually worse than too many comments.

Don't use comments to excuse hard-to-read code. Make the code as easy to understand as possible, and then add just enough comments to fill in the missing details.

Challenge The Thing Namer 3000 100 XP

As you walk through the city of Commenton, admiring its forward-slash-based architectural buildings, a young man approaches you in a panic. "I dropped my *Thing Namer 3000* and broke it. I think it's mostly working, but all my variable names got reset! I don't understand what they do!" He shows you the following program:

```
using System;

Console.WriteLine("What kind of thing are we talking about?");
string a = Console.ReadLine();
Console.WriteLine("How would you describe it? Big? Azure? Tattered?");
string b = Console.ReadLine();
string c = "of Doom";
string d = "3000";
Console.WriteLine("The " + b + " " + a + " of " + c + " " + d + "!");
```

"You gotta help me figure it out!"

Objectives:

- Rebuild the program above on your computer.
- Add comments near each of the four variables that describe what they store. You must use at least one of each comment type (`//` and `/* */`).
- Find the bug in the text displayed and fix it.
- **Answer this question:** Aside from comments, what is one other thing you could do to make this code more understandable.

Level 5

Variables

Speedrun

- A variable is a named location in memory for storing data.
- Variables have a type, a name, and a value (contents).
- Variables are declared (created) like this: `int number;`.
- Assigning values to variables is done with the assignment operator: `number = 3;`
- Using a variable name in an expression will copy the value out of the variable.
- Give your variables good names. You will be glad you did.

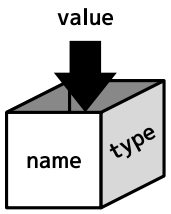
In this level, we will look at variables, first seen in Level 3, in more depth, including how to declare them, put data into them, and pull data out of them. We will also look at some rules around good variable names.

What is a Variable?

A crucial part of building software is storing data in temporary memory to use later. For example, we might store a player's current score or remember a menu choice long enough to respond to it. When we talk about memory and variables, we are talking about "volatile" memory (or RAM) that sticks around while your program runs but is wiped out when your program closes or the computer is rebooted. (To let data survive longer than the program, we must save it to persistent storage in a file, which is the topic of Level 39.)

A computer's total memory is gigantic. Even my old smartphone has 3 gigabytes of memory—large enough to store 750 million different numbers. Each memory location has a unique numeric *memory address*, which can be used to view any specific location's contents. But remembering what spot #45387 is used for is not practical. Data comes and goes in a program. We might need one thing for a split second and another for the whole time the program is running. Plus, not all pieces of data are the same size. The text "Hello World!" takes up more space than a single number does. We need something smarter than raw memory addresses.

A *variable* solves this problem for us. Variables are named locations where data is stored in memory. Each variable has three parts: its name, its type, and its contents or data. A variable's type is important because it lets us know how many bytes to reserve for it in memory, and it also allows the compiler to ensure that we are using its contents correctly.



The first step in using a variable is to *declare* it. Declaring a variable allows the computer to reserve a spot in memory of the appropriate size for it.

After declaring a variable, you can *assign* values or contents to the variable. The first time you assign a value to a variable is called *initializing* it. Before a variable is initialized, it is impossible to know what bits and bytes might be in that memory location, so initialization ensures we are always working with legitimate data.

While you can only declare a variable once, you can assign it different values over time as the program runs. A variable for the player's score can update as they collect points. The underlying memory location remains the same, but the contents change with new values over time.

The third thing you can do with a variable is to retrieve its current value. If we save off data for later, we inevitably will want to come back to it. As long as a variable has been initialized, we can retrieve its current contents whenever we need it.

CREATING AND USING VARIABLES IN C#

The following code shows all three primary variable-related activities:

```
string username;                        // Declaring a variable
username = Console.ReadLine();          // Assigning a value to a variable
Console.WriteLine("Hi " + username);    // Retrieving its current value
```

A variable is declared by listing its type and its name together (**`string username;`**).

A variable is assigned a value by placing the variable name on the left side of an equal sign and the new value on the right side. This new value may be an expression that the computer will evaluate to determine the value (**`username = Console.ReadLine();`**).

Reading the variable's current value is done by simply using the variable's name in an expression (**`"Hi " + username`**). In this case, your program will start by retrieving the current value in **`username`**. It then uses that value to produce the complete **`"Hi [name]"`** message. The combined message is what is supplied to the **`WriteLine`** method.

You can declare a variable anywhere within your code. Still, because variables must be declared before they are used, it is common for programmers to tend to put most or all variable declarations at the top of their code.

Each variable can only be declared once, though your programs can create many variables. You can assign new values to variables or retrieve the current value in a variable as often as you want:

```
string username

username = Console.ReadLine();
Console.WriteLine("Hi " + username);
```

```
username = Console.ReadLine();
Console.WriteLine("Hi " + username);
```

Given that **username** above is used to store two different usernames over time, it is reasonable to reuse the variable. On the other hand, if the second value is supposed to represent something else—say a favorite color—then it is better to make a second variable:

```
string username;
username = Console.ReadLine();
Console.WriteLine("Hi " + username);

string favoriteColor;
favoriteColor = Console.ReadLine();
Console.WriteLine("Hi " + favoriteColor);
```

Remember that variable names are meant for humans to use, not the computer. Pick names that will help human programmers understand their intent. The computer does not care.

Declaring a second variable technically takes up more space in memory, but spending a few extra bytes (when you have billions) to make the code more understandable is a clear win.

INTEGERS

Every variable and every value created in your C# programs has a type associated with it. Before now, the only type we have seen has been **string**s—text. But many other types exist, and we can even define our own types. Let's look at a second type: **int**, representing an integer.

An integer is a whole number (no fractions or decimals) but either positive, negative, or zero. Given the computer's capacity for doing math, it should be no surprise that storing numbers is a common thing to do, and many variables use the **int** type. For example, all of these would be well represented as an **int**: a player's score, pixel locations on a screen, a file's size, and a country's population.

Declaring an **int**-typed variable is as simple as using the **int** type instead of the **string** type when we declare it:

```
int score;
```

This **score** variable is now built to hold int values instead of text.

This type thing is important, so I'll state it again: types matter in C#; every value and every variable you create has a specific type, and the compiler will ensure that you don't mix them up. The following fails to compile because the types don't match:

```
score = "Generic User"; // DOESN'T COMPILE!
```

The text **"Generic User"** is a **string**, but **score**'s type is **int**. This one is more subtle:

```
score = "0"; // DOESN'T COMPILE!
```

At least this *looks* like a number. But enclosed in quotes like that, **"0"** is a string representation of a number, not an actual number. **"0"** is an example of what we call a *literal value*, or simply a *literal*. A literal is where our C# code just outright states a fixed, known value. Literals are an

important way to produce values in your program, with two other ways being user input and evaluating expressions.

All literals have a type. A **`string`** literal is formed by enclosing the desired text in double-quotes, as shown previously. An **`int`** literal is formed by writing out the integer without quotes. The following uses an **`int`** literal to assign an initial value of **`0`** to **`score`**:

```
score = 0; // Works!
```

After this line of code runs, the **`score`** variable—a memory location reserved to hold **`int`**s under the name **`score`**—has a value of **`0`**.

The following shows that you can assign different values to **`score`** over time, as well as negative numbers:

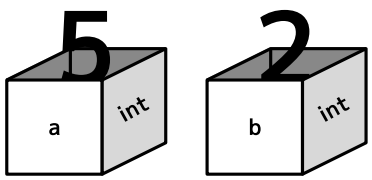
```
score = 4;
score = 11;
score = -1564;
```

READING FROM A VARIABLE DOES NOT CHANGE IT

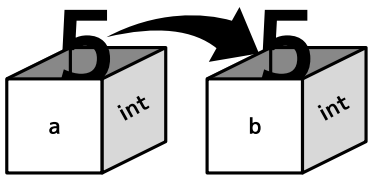
When you read the contents of a variable, the variable's contents are copied out. To illustrate:

```
int a;
int b;

a = 5;
b = 2;

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

The first few lines are pretty straightforward. **`a`** and **`b`** are declared and given an initial value (5 and 2 respectively), which looks something like this:



On that fifth line, **`b = a;`**, the contents of **`a`** are copied out of **`a`** and replicated into **`b`**.



The variables **`a`** and **`b`** are distinct, each with its own copy of the data. **`a = b`** does not mean **`a`** and **`b`** are now always going to be equal! That **`=`** symbol means assignment, not equality. (Though **`a`** and **`b`** will happen to be equal immediately after running line 5.) Once the final line runs, assigning a value of **`-3`** to **`a`**, **`a`** will be updated as expected, but **`b`** retains the **`5`** it already

had. If we displayed the values of **a** and **b** at the end of this program, we would see that **a** is **-3** and **b** is **5**.

There are some nuances to variable assignment, which we elaborate on in Level 14.

CLEVER VARIABLE TRICKS

Declaring and using variables is so common that there are some useful shortcuts to learn before moving on.

The first is that you can declare a variable and initialize it on the same line, like this:

```
int x = 0;
```

This trick is so useful that virtually all experienced C# programmers would use this instead of putting the declaration and initialization on back-to-back lines.

Second, you can declare multiple variables simultaneously if they are the same type:

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

Third, variable assignments are also expressions that evaluate to whatever the assigned value was, which means you can assign the same thing to many variables all at once like this:

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

The value of **10** is assigned to **c**, but **c = 10** is an expression that evaluates to **10**, then assigned to **b**. **b = c = 10** evaluates to **10**, and that value is placed in **a**. The above code is the same as the following:

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

In my experience, this is not common practice, but it does have its uses.

And finally, while types matter, **Console.WriteLine** can display both strings and integers:

```
Console.WriteLine(42);
```

In the next level, we will introduce many more variable types. **Console.WriteLine** can display every single one of them. That is, while types matter and are not interchangeable, **Console.WriteLine** is set up to allow for it to work with any type. We will see how this works and learn to do it ourselves in the future.

VARIABLE NAMES

You have a lot of control over what names you give to your variables, nut the language has a few rules about what you can choose:

1. Variable names must start with a letter or the underscore character (_). (Though C# casts a wide net when defining "letters"—almost anything in any language is allowed.) **taco** and **_taco** are both legitimate variable names, but **1taco** and ***taco** are not.
2. After the start, you can also use numeric digits (**0** through **9**).

3. Most symbols and most whitespace characters are banned because they make it impossible for the compiler to know where a variable name ends and other code begins. (As an example, `taco-poptart` is not allowed because the `-` character is used for subtraction. The compiler assumes this is an attempt to subtract something called `poptart` from something called `taco`.)
4. You cannot name a variable the same thing as a keyword. For example, you cannot call a variable `int` or `string`, as those are reserved, special words in the language.

I also recommend the following guidelines for naming variables:

1. **Accurately describe what the variable holds.** If the variable contains a player's score, `score` or `playerScore` are acceptable. But `number` and `x` are not descriptive enough.
2. **Don't abbreviate or remove letters.** You spend more time reading code you previously wrote than you do writing it, and if you must decode every variable name you encounter, you're doing yourself a disservice. What did `plrscr` (or worse, plain `ps`) stand for again? Plural scar? Plastic Scrabble? No, just player score. Common acronyms like `html` or `dvd` are an exception to this rule.
3. **Don't fret over long names.** It is better to use a descriptive name than to "save characters." With any half-decent IDE, you can use features like AutoComplete to finish long names after typing just a few letters anyway, and skipping the meaningful parts of names makes it harder to remember what it does.
4. **Names ending in numbers are a sign of poor names.** With a few exceptions, variables named `number1`, `number2`, and `number3`, do not distinguish one from another well enough. (If they are part of a set that ought to go together, they should be packaged that way; see Level 12.)
5. **Avoid generic catch-all names.** Names like `item`, `data`, `text`, and `number` are too vague to be helpful in most cases.
6. **Make the boundaries between multi-word names clear.** A name like `playerScore` is easier to read than `playerscore`. Two conventions among C# programmers are `camelCase` (or `lowerCamelCase`) and `PascalCase` (or `UpperCamelCase`), which are illustrated by the way their names are written. In the first, every word but the first starts with a capital letter. In the second, every word, including the first, begins with a capital letter. (The big capital letter in the middle of the word makes it look like a camel's hump.) Most C# programmers use `lowerCamelCase` for variables and use `UpperCamelCase` for other things. I recommend sticking with that convention as you get started, but the choice is yours.

Picking good variable names doesn't guarantee readable code, but it goes a long way.

Knowledge Check — Level 5 — 25 XP

Check your knowledge with the following questions:

1. Name the three things all variables have.
2. **True/False.** Variables must always be declared before being used.
3. How many times must a variable be declared?
4. Which of the following are legal C# variable names? `answer`, `1stValue`, `value1`, `$message`, `delete-me`, `delete_me`, `PI`.

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

Level 6

The C# Type System

Speedrun

- Types of variables and values matter in C#. They are not interchangeable.
- There are eight integer types for storing integers of differing sizes and ranges: **`int`**, **`short`**, **`long`**, **`byte`**, **`sbyte`**, **`uint`**, **`ushort`**, and **`ulong`**.
- The **`char`** type stores single characters.
- The **`string`** type stores longer text.
- There are three types for storing real numbers: **`float`**, **`double`**, and **`decimal`**.
- The **`bool`** type stores truth values (true and false) used in logic.
- These types are the building blocks of a much larger type system.
- Using **`var`** for a variable's type tells the compiler to infer its type from the surrounding code, so you do not have to type it out. (But it still has a specific type.)
- The **`System.Convert`** class is a useful class to convert from one type to another.

In C#, types of variables and values matter (and must match), but we only know about two types so far. In this level, we will introduce a diverse set of types we can use in our programs. These types are called *built-in types* or *primitive types*. They are building blocks for more complex types that we will see later.

Representing Data in Binary

Why do types matter so much?

Every piece of data you want to represent in your programs must be stored in the computer's circuitry, limited to only the 1's and 0's of binary. If we're going to store a number, we need a scheme for using *bits* (a single 1 or 0) and *bytes* (a group of 8 bits and the standard grouping size of bits) to represent the range of possible numbers we want to store. If we're going to represent a word, we need some scheme for using the bits and bytes to represent both letters and sequences (strings) of letters. More broadly, for *anything* we might want to represent in a program, we need a scheme for expressing it in binary.

Each type defines its own rules for representing values in binary, and different types are not interchangeable. You cannot take bits and bytes meant to represent a number and reinterpret those bits and bytes as a string and expect to get meaning out of it. Nor can you take bits and bytes meant to represent text and reinterpret them as an integer and expect it to be meaningful either. They are not the same, and there's no getting around it.

That doesn't mean that each type is a world unto itself that can never interact with the other worlds. We can and will convert from one type to another frequently. But the costs associated with conversion are not free, so we do it conscientiously rather than accidentally.

Notably, C# does not invent new schemes and rules for most of its types. The computing world has developed schemes for common types like numbers and letters, and C# reuses these schemes when possible. The physical hardware of the computer also uses these same schemes. Since it is baked into the circuitry, it can be fast.

The specifics of these schemes are beyond this book's scope, but let's do a couple of thought experiments to explore.

Suppose we want to represent the numbers 0 through 10. We need to invent a way to describe each of these numbers with only 0's and 1's. Step 1 is to decide how many bits to use. One bit can store two possible states (0 and 1), and each bit you add after that doubles the total possibilities. We have 11 possible states, so we will need at least 4 bits to represent all of them. Step 2 is to figure out which bit patterns to assign to each number. 0 can be **`0000`**. 1 can be **`0001`**. Now it gets a little more complicated. 2 is **`0010`**, and 3 is **`0011`**. (We're counting in binary if that happens to be familiar to you.) We've used up all possible combinations of the two bits on the right and need to use the third bit. 4 is **`0100`**, 5 is **`0101`**, and so on, all the way to 10, which is **`1010`**. We have some unused bit patterns. 1011 isn't anything yet. We could go all the way up to 15 without needing any more bits.

We have one problem: the computer doesn't give us a way to deal with anything smaller than full bytes. Not a big deal; we'll just use a full byte of eight bits.

If we want to represent letters, we can do a similar thing. We could assign the letter A to **`01000001`**, B to **`01000010`**, and so on. (C# uses two bytes for every character.)

If we want to represent text (a string), we can use our letters as a building block. Perhaps we could use a full byte to represent how many letters long our text is and then use two bytes for each letter in the word. This is tricky because short words need to use fewer bytes than longer words, and our system has to account for that. But it is a scheme that works.

We don't have to invent these schemes for types ourselves, fortunately. The C# language has taken care of them for us. But hopefully, this illustrates why we can't magically treat an integer and a string as the same thing. (Though we will be able to convert from one type to another.)

Integer Types

Let's explore the basic types available in a C# program, starting with the types used to represent integers. While we used the int type in the previous level, there are eight different types for working with integers. These eight types are called *integer types* or *integral types*. Each uses a different number of bytes, which allows you to store bigger numbers while using more memory or store smaller numbers while conserving memory.

The **`int`** type uses 4 bytes and can represent numbers in the range of roughly -2 billion to +2 billion. (The specific numbers are in the table below.)

In contrast, the **short** type uses 2 bytes and can represent numbers in the range of -32,000 to +32,000. The **long** type uses 8 bytes and can represent numbers in the range of -9 quintillion to +9 quintillion (a quintillion is a billion billion).

Their sizes and ranges tell you when you might choose **short** or **long** over **int**. If memory is tight and a **short**'s range is sufficient, you can use a **short**. If you need to represent numbers larger than what an **int** can handle, you need to move up to a **long**, even at the cost of more bytes.

The **short**, **int**, and **long** types are *signed* types; they include a positive or negative sign and store positive and negative values. If you know you only need positive numbers, you could imagine shifting the range of each of these types upward to exclude any negative values but twice as many positive values. This is what the *unsigned* types are for: **ushort**, **uint**, and **ulong**. Each of these uses the same number of bytes as their signed counterpart, cannot store negative numbers, but can store twice as many positive numbers. Thus **ushort**'s range is roughly 0 to about 65,000, **uint**'s range is roughly 0 to 4 billion, and **ulong**'s range is roughly 0 to 18 quintillion.

The last two integer types are a little different. The first is the **byte** type, using a single byte to represent values from 0 to 255 (unsigned). While integer-like, the byte type is more often used to express a byte or collection of bytes with no specific structure (or none known to the program). The **byte** type has a signed counterpart, **sbyte**, representing values in the range -128 to +127. The sbyte type is not used very often but makes the set complete.

The table below summarizes this information.

Name	Bytes	Allow Negatives	Minimum	Maximum
byte	1	No	0	255
short	2	Yes	-32,768	32,767
int	4	Yes	-2,147,483,648	2,147,483,647
long	8	Yes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
sbyte	1	Yes	-128	127
ushort	2	No	0	65,536
uint	4	No	0	4,294,967,295
ulong	8	No	0	18,446,744,073,709,551,615

Declaring and Using Variables with Integer Types

Declaring variables of these other types is as simple as declaring them with these type names instead of **int** or **string**, as we have done before:

```
byte aSingleByte = 34;
aSingleByte = 17;

short aNumber = 5039;
aNumber = -4354;

long aVeryBigNumber = 395904282569;
aVeryBigNumber = 13;
```

In the past, we saw that writing out a number directly in our code creates an **int** literal. But this brings up an interesting question. How do we create a literal that is a **byte** literal or a **ulong** literal?

For things smaller than an **int**, nothing special is needed to create a literal of that type:

```
byte aNumber = 32;
```

The **32** is an **int** literal, but the compiler is smart enough to see that you are trying to store the literal value in a **byte** and can ensure by inspection that **32** is within the allowed range for a **byte**. The compiler handles it. In contrast, if you used a literal that was too big for a **byte**, you would get a compiler error, preventing you from compiling and running your program.

This same rule also applies to **sbyte**, **short**, and **ushort**.

If your literal value is too big to be an **int**, it will automatically become a **uint** literal, a **long** literal, or a **ulong** literal, the first of those capable of representing the number you typed. (If you make a literal whose value is too big for everything, you will get a compiler error.) To illustrate how these bigger literal types work, consider this code:

```
long aVeryBigNumber = 10000000000; // 10 billion would be a long literal.
```

You may find that you want to force a smaller number to be one of the larger literal types on rare occasions. You can force this by putting a **U** or **L** (or both) at the end of the literal value:

```
ulong aVeryBigNumber = 10000000000U;
aVeryBigNumber = 10000000000L;
aVeryBigNumber = 10000000000UL;
```

A **U** signifies that it is unsigned and must be either a **uint** or **ulong**. **L** indicates that the literal must be a **long** or a **ulong**, depending on the size. A **UL** indicates that it must be a **ulong**. These suffixes can be uppercase or lowercase and in either order. However, avoid using a lowercase **l** because that looks too much like a **1**.

You shouldn't need these suffixes very often.

The Digit Separator

As humans, when we write a long number like 1,000,000,000, we often use a separator like a comma to make interpreting the number easier. While we can't use the comma for that in C#, there is an alternative: the underscore character (_).

```
int bigNumber = 1_000_000_000;
```

The normal convention for writing numbers is to group them by threes (thousands, millions, billions, etc.), but the C# compiler does not care where these appear in the middle of numbers. In situations where a different grouping makes more logical sense, use it that way. All the following are allowed:

```
int a =  123_456_789;
int b = 12_34_56_78_9;
int c = 1_2__3___4____5;
```

Choosing Between the Integer Types

With eight types for storing integers, how do you decide which one to use?

On the one hand, you could carefully consider the possible range of values you might want for any variable and then pick the smallest (to save on memory usage) that can fit the intended range. For example, if you need a player's score and know it can never be negative, you have cut out half of the eight options right there. If the player's score may be in the hundreds of thousands in any playthrough, you can rule out **`byte`** and **`ushort`** because they're not big enough. That leaves you with only **`uint`** and **`ulong`**. If you think a player's score might approach 4 billion, you'd better use **`ulong`**, but if scores will get only into the millions, then a **`uint`** is safe. (You can always change the types of a variable and recompile your program if you got it wrong—software is soft after all—but it is easier to have just been right the first time.)

The strategy of picking the smallest practical range for any given variable has merit, but it has two things going against it that make a different strategy more common. The first is that in modern programming, rarely does saving a single byte of space matter. There is too much memory around to fret over individual bytes. The second is that computers do not have hardware that supports math with smaller types. The computer upgrades them to **`int`**s and runs the math as **`int`**s, forcing you to then go to the trouble of converting the result back to the smaller type. The **`int`** type is more convenient to work with than **`sbyte`**, **`byte`**, **`short`**, and **`ushort`** if you are doing many math operations with them.

Thus, the more common strategy is to use **`int`**, **`uint`**, **`long`**, or **`ulong`** as necessary and only use **`byte`**, **`sbyte`**, **`short`**, and **`ushort`** when there is a clear and obvious benefit.

Binary and Hexadecimal Literals

The integer literals we have written so far have all been written using *base 10*, the normal 10-digit system humans are typically used to. But in the programming world, it is occasionally easier to write out the number using either *base 2* (binary digits) or *base 16* (hexadecimal digits, which are 0 through 9, and then the letters A through F).

To write a binary literal, start your number with a **`0b`**. For example:

```
int thirteen = 0b00001101;
```

For a hexadecimal literal, you start your number with **`0x`**:

```
int theColorMagenta = 0xFF00FF;
```

This example shows one of the places where this might be useful. Colors are often represented as either six or eight hexadecimal digits.

TEXT: CHARACTERS AND STRINGS

There are more numeric types, but let's turn our attention away from numbers for a moment and look at representing single letters and longer text.

In C#, the **`char`** type represents a single character, while our old friend **`string`** represents text of any length.

The **`char`** type is very closely related to the integer types. (It is even lumped into the integral type banner with the rest of the integer types.) Each character of interest is given a number representing it, which amounts to a unique bit pattern. The **`char`** type is not limited to just keyboard characters. The **`char`** type uses two bytes to allow for a total of 65,537 distinct characters. The number assigned to each character follows a widely used standard called

Unicode. This set covers English characters and every character in every human-readable language and a whole slew of other random characters and emoji. A **`char`** literal is made by placing the character in single quotes:

```
char aLetter = 'a';
char baseball = '⚾';
```

You won't find too many uses for the extremely esoteric characters (the console window does not know how to display the baseball character above). Still, the diversity of characters available is nice.

If you know the hexadecimal Unicode number for a symbol and would prefer to use that, you can write that out after a **`\u`**:

```
char aLetter = '\u0061'; // An 'a'
```

The **`string`** type aggregates many characters into a sequence to allow for arbitrary text to be represented. The word "string" comes from the math world, where a string is a sequence of symbols chosen from a defined set of allowed symbols, one after the other, of any length. It is a word that the programming world has stolen from the math world, and most programming languages refer to this idea as strings.

A **`string`** literal is made by placing the desired text in double-quotes:

```
string message = "Hello World!";
```

FLOATING-POINT TYPES

We now return to the number world to look at types that represent numbers besides integers. How do we represent 1.21 gigawatts or the number π?

C# has three types that are called *floating-point data types*. These represent what mathematicians call "real numbers," which includes integers and numbers with a decimal or fractional component. While we cannot represent 3.1415926 as an integer (3 is the best we could do), we can represent it as a floating-point number.

The "point" in the name refers to the decimal point that often appears when writing out these numbers.

The "floating" part comes because it contrasts with fixed-point types. With a fixed-point type, the number of digits before and after the decimal point is locked in place. With a floating-point type, the decimal point may appear anywhere within the number. C# does not have fixed-point types because they prevent you from using very large or very small numbers efficiently. In contrast, floating-point numbers let you represent a specific number of significant digits and scale them to be big or small. For example, they allow you to express the numbers 1,250,421,012.6 and 0.00000000000012504210126 equally well, which is something a fixed-point representation cannot reasonably do.

With floating-point types, some of the bits can store the significant digits (which affects how precise you can be), while other bits define how much to scale it up or down (which affects the magnitudes you can represent). The more bits you use, the more of either you can do. Once again, this scheme is not a C# invention but one broadly used across the computing world.

There are three flavors of floating-point numbers: **`float`**, **`double`**, and **`decimal`**. The **`float`** type uses 4 bytes, while **`double`** uses twice that many (hence the "double") at 8 bytes. The

decimal type uses 16 bytes. While **float** and **double** follow conventions used across the computing world, **decimal** is not. That means **float** and **double** are faster. However, **decimal** uses most of its bits for storing significant figures—it is by far the most precise floating-point type. If you are doing something that needs extreme mathematical precision, even at the cost of speed, **decimal** is the better choice.

All floating-point numbers have ranges that are mind-boggling in size and can only reasonably be represented in *scientific notation*. An example of this notation is 2×10^5. If this format is not familiar to you, a rough approximation is to focus solely on the exponent—the 5 in the previous example. To convert this to a "normal" notation, write a one followed by that many zeroes. 2×10^5 is approximately 100000. If the number is negative, then write out that many zeroes followed by a 1, then put the decimal point after the first zero. 2×10^{-5} is approximately 0.00001. With that in mind, we can look at the ranges floating-point types can represent.

A **float** can store numbers as small as 3.4×10^{-45} and as large as 3.4×10^{38}. That is small enough to measure quarks and large enough to measure the visible universe many times over. A **float** has 6 to 7 digits of precision, depending on the number, meaning it can represent the number 10000 and the number 0.0001, but not quite the resolution to differentiate between 10000 and 10000.0001.

A **double** can store numbers as small as 5×10^{-324} and as large as 1.7×10^{308}, with 15 to 16 digits of precision.

A **decimal** can store numbers as small as 1.0×10^{-28} and as large as 7.9×10^{28}, with 28 to 29 digits of precision.

(I'm not going to write out all of those numbers in normal notation, but it is worth imagining what they might look like.)

All three floating-point representations are insane in size, but seeing the exponents, you should have a feel for how they compare to each other. The **float** type uses the fewest bytes, and its range and precision are good enough for almost everything. The **double** type can store the biggest big numbers and the smallest small numbers with even more precision than a float. The **decimal** type's range is the smallest of the three but is the most precise and is great for calculations where accuracy matters (like financial or monetary calculations).

The table below summarizes how these types compare to each other:

Type	Bytes	Range	Digits of Precision	Hardware Supported
float	4	±1.0e-45 to ±3.4e38	7	Yes
double	8	±5e-324 to ±1.7e308	15-16	Yes
decimal	16	±1.0 × 10e-28 to ±7.9e28	28-29	No

Creating variables of these types is the same as any other type, but it gets more interesting when you make **float**, **double**, and **decimal** literals:

```
double  number1 = 3.5623;
float   number2 = 3.5623f;
decimal number3 = 3.5623m;
```

If a number literal contains a decimal point, it becomes a **double** literal instead of an integer literal. Appending an **f** or **F** onto the end (with or without the decimal point) makes it a **float**

literal. Appending an **`m`** or **`M`** onto makes it into a **`decimal`** literal. (The "m" is for "monetary" or "money." Financial calculations often need extremely high precision.)

All three types can represent a bigger range than any integer type, so if you use an integer literal, the compiler will automatically convert it.

Scientific Notation

As we saw when we first introduced the range floating-point numbers can represent, really big and really small numbers are more concisely represented in scientific notation. For example, 6.022×10^{23} instead of 602,200,000,000,000,000,000,000. (That number, by the way, is called Avogadro's Number, a number with special significance in chemistry.) The × symbol is not one on a keyboard, so for decades, scientists have written a number like 6.022×10^{23} as 6.022e23, where the e stands for "exponent." Floating-point literals in C# can use this same notation by embedding an **`e`** or **`E`** in the number:

```
double avogadrosNumber = 6.022e23;
```

THE BOOL TYPE

The final type that we will cover in this chapter is the **`bool`** type. The **`bool`** type might seem like the strangest type if you are new to programming, but we will see its value before long. The **`bool`** type gets its name from Boolean logic, which was named after its creator, George Boole. The **`bool`** type represents "truth" values. These are used in decision making, which we will cover in Level 9. It has two possible options: **`true`** and **`false`**. Both of those are **`bool`** literals that you can write into your code:

```
bool itWorked = true;
itWorked = false;
```

Some languages treat **`bool`** as nothing more than fancy **`int`**s, with **`false`** being the number **`0`** and **`true`** being anything else. But C# delineates **`int`**s from **`bool`**s because conflating the two is a pathway to lots of common bug categories.

A **`bool`** could theoretically use just a single bit, but it uses a whole byte.

Challenge — The Variable Shop — 100 XP

You see an old shopkeeper struggling to stack up variables in a window display. "Hoo-wee! All these variable types sure are exciting but setting them all up to show them off to excited new programmers like yourself is a lot of work for these aching bones," she says. "You wouldn't mind helping me set up this program with one variable of every type, would you?"

Objectives:

- Build a program with a variable of all fourteen types described in this level.
- Assign each of them a value using a literal of the right type.
- Use **`Console.WriteLine`** to display the contents of each variable.

Challenge **The Variable Shop Returns** **50 XP**

"Hey! Programmer!" It's the shopkeeper from the Variable Shop who hobbles over to you. "Thanks to your help, variables are selling like RAM cakes! But these people just aren't programmers like you. They keep asking how to modify the values of the variables they're buying, and... well... frankly, I have no clue. But you're a programmer, right? Maybe you could show me so I can show my customers?"

Objectives:

- Modify your *Variable Shop* program to assign a new, different literal value to each of the 14 original variables. Do not declare any additional variables.
- Use **`Console.WriteLine`** to display the updated contents of each variable.

This level has introduced the 14 most fundamental types of C#. It may seem a lot to take in, and you may still be wondering when exactly you should use one type over another. But don't worry too much. This level will always be here as a reference when you need it.

These are not the only possible types in C#. They are more like chemical elements, serving as the basis or foundation for producing other types.

TYPE INFERENCE

Types matter greatly in C#. Every variable and each value has a specific, known type. We have been very specific when declaring variables to call out each variable's type. But the compiler is very smart. It can often look at your code and figure out ("infer") what type something is from clues and cues around it. This feature is called *type inference*. It is the Sherlock Holmes of the compiler.

Type inference is used for many language features, but a notable one is that the compiler can infer the type of a variable based on the code that it is initialized with. You don't need to write out a variable's type yourself in many cases. You can use the **`var`** keyword instead:

```
var message = "Hello World!";
```

The compiler can tell that **`"Hello World!"`** is a **`string`**, and therefore, **`message`** must be a **`string`** for this code to work. Using **`var`** tells the compiler, "You've got this. I know you can figure it out. I'm not going to bother writing it out myself."

This only works if you initialize the variable on the same line that you declare it. Otherwise, there is not enough information for the compiler to infer its type. This won't work:

```
var x; // DOES NOT COMPILE!
```

There are no clues to use in performing type inference, so the type inference fails. You will have to fall back to using specific, named types.

In Visual Studio, you can easily see what type the compiler inferred by hovering the mouse over the **`var`** keyword until the tooltip appears, which shows the inferred type.

Many programmers prefer to use **`var`** everywhere they possibly can. It is often shorter and cleaner, especially when we start using things with longer type names.

But there are two potential problems to consider with **`var`**.

The first is that the computer sometimes infers the wrong type. These errors are sometimes subtle.

The second problem is that the computer is faster at inferring a variable's type than a human. Consider this code:

```
var input = Console.ReadLine();
```

The computer can infer that **input** is a **string** since it knows **ReadLine** produces **string**s as a result. As a human, I need to have that information in my head already to infer it.

It is worse when the code comes from the Internet or a book because you don't necessarily have all of the information to figure it out. For that reason, I will avoid **var** in this book.

My recommendation is that you skip **var** and use specific types as you start working in C#. Doing this helps you think about types more carefully. After some practice, if you want to switch to **var**, go for it.

I want to make this next point very clear, so pay attention: a variable that uses **var** still has a specific type. It isn't a mystery type, a changeable type, or a catch-all type. It still has a specific type; we have just left it unwritten. This does not work:

```
var something = "Hello";
something = 3; // ERROR! Cannot store an int in a string-typed variable.
```

THE CONVERT CLASS

With 14 types at our disposal, we will need to be able to convert between types. The easiest way to do this is with the **Convert** class. The **Convert** class is like the **Console** class—a thing in the system that provides you with a set of tasks or capabilities that it can perform. While **Console** is for working with the console window, **Convert** is for converting between these different built-in types. To illustrate:

```
Console.Write("What is your favorite number?");
string favoriteNumberText = Console.ReadLine();
int favoriteNumber = Convert.ToInt32(favoriteNumberText);
Console.Write(favoriteNumber + " is a great number!");
```

You can see that **Convert**'s **ToInt32** method needs a **string** as an input and gives back or returns an **int** as a result, converting the text in the process. The **Convert** class has **ToWhatever** methods to convert among the built-in types, though some names are a bit surprising:

Method Name	Target Type	Method Name	Target Type
ToByte	byte	ToSByte	sbyte
ToInt16	short	ToUInt16	ushort
ToInt32	int	ToUInt32	uint
ToInt64	long	ToUInt64	ulong
ToChar	char	ToString	string
ToSingle	float	ToDouble	double
ToDecimal	decimal	ToBoolean	bool

Most of the names above are straightforward, though a few deserve a bit of explanation. The names are not a perfect match because the **`Convert`** class is part of .NET's Base Class Library, which all .NET languages use. No two languages use the same name for things like **`int`** and **`double`**.

The **`short`**, **`int`**, and **`long`** types, along with their unsigned counterparts, use the word **`Int`** and the number of bits they use. For example, a **`short`** uses 16 bits (2 bytes), so **`ToInt16`** converts to a **`short`**.

The other surprise is that converting to a **`float`** is **`ToSingle`** instead of **`ToFloat`**. But a **`double`** is considered "double precision," and a **`float`** is "single precision," which is where the name comes from.

All input from the console window starts as **`string`**s. Many of our programs will need to convert text that a user enters and process it. The process of analyzing text, breaking it apart, and transforming it into other data is called *parsing*. The **`Convert`** class is a great starting point for parsing text, though we will learn more tools for this over time as well.

Knowledge Check Level 6 25 XP

Check your knowledge with the following questions:

1. **True/False.** The **`int`** type can store any possible integer.
2. Order the following by how large their range is, from smallest to largest: **`short`**, **`long`**, **`int`**, **`byte`**.
3. **True/False**. The **`byte`** type is signed.
4. Which can store higher numbers, **`int`** or **`uint`**?
5. What three types can store floating-point numbers?
6. Which of the options in question 5 can hold the largest numbers?
7. Which of the options in question 5 is the most precise?
8. What type does the literal value **`"8"`** (including the quotes) have?
9. What type stores true or false values?

Answers: (1) false. **(2)** **`byte`**, **`short`**, **`int`**, **`long`**. **(3)** false. **(4)** **`uint`**. **(5)** **`float`**, **`double`**, **`decimal`**. **(6)** **`double`**. **(7)** **`decimal`**. **(8)** **`string`**. **(9)** **`bool`**.

Level 7

Math

Speedrun

- Addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and remainder (`%`) can all be used to do math in expressions: `int a = 3 + 2 / 4 * 6;`
- The `+` and `-` operators can also be used to indicate a sign (or negate a value): `+3`, `-2`, or `-a`.
- Operator precedence (order of operations) matches the math world. Multiplication and division happen before addition and subtraction, and things are evaluated left to right.
- Change the order by using parentheses to group things you want to be done first.
- Compound assignment operators (`+=`, `-=`, `*=`, `/=`, `%=`) are shortcuts that adjust a variable with a math operation. `a += 3;` is the same as `a = a + 3;`
- The increment and decrement operators add and subtract one: `a++; b--;`
- Each of the numeric types defines special values for their ranges (`int.MaxValue`, `double.MinValue`, etc.), and the floating-point types also define `PositiveInfinity`, `NegativeInfinity`, and `NaN`.
- Integer division drops remainders while floating-point division does not. Dividing by zero in either system is bad.
- You can convert between types by casting: `int x = (int)3.3;`
- The `Math` and `MathF` classes contain a collection of useful utility methods for dealing with common math operations such as `Abs` for absolute value, `Pow` and `Sqrt` for powers and square roots, and `Sin`, `Cos`, and `Tan` for the trigonometry functions sine, cosine, and tangent, as well as a definition of π (`Math.PI`)

Computers were built for math, and it is high time we saw how to make the computer do some basic arithmetic.

Operations and Operators

Let's define a few terms to get started. An *operation* is a calculation that takes (usually) two numbers and produces a single result by combining them somehow. Each *operator* indicates how the numbers are combined, and a particular symbol represents each operator. For example, **2 + 3** is an operation. The operation is addition, shown with the **+** symbol. The things that an operation use—the **2** and **3** here—are referred to as operands.

Most operators need two operands. These are called *binary operators* ("binary" meaning "composed of two things"). An operator that needs one operand is a *unary operator*, while one that needs three is a *ternary operator*. C# has many binary operators, a few unary operators, and a single ternary operator.

Addition, Subtraction, Multiplication, and Division

C# borrows the operator symbols from the math world where it can. For example, to add together 2 and 3 and store its result into a variable looks like this:

```
int a = 2 + 3;
```

The **2 + 3** is an operation, but all operations are also expressions. When our program runs, it will take these two values and evaluate them using the operation listed. This expression evaluates to a **5**, which is the result placed in **a**'s memory.

The same thing works for subtraction:

```
int b = 5 - 2;
```

Arithmetic like this can be used in any expression, not just when initializing a variable:

```
int a;          // Declaring the variable a.
a = 9 - 2;      // Assigning a value to a, using some math.
a = 3 + 3;      // Another assignment.

int b = 3 + 1;  // Declaring b and assigning a value to b all at once.
b = 1 + 2;      // Assigning a second value to b.
```

Operators do not need literal values; they can use any expression. For example, the code below uses more complex expressions that contain variables:

```
int a = 1;
int b = a + 4;
int c = a - b;
```

That is important. Operators and expressions allow us to work through some process (sometimes called an *algorithm*) to compute some result that we care about, step by step. Variables can be updated over time as our process runs.

Multiplication uses the asterisk (*****) symbol:

```
float totalPies = 4;
float slicesPerPie = 8;
float totalSlices = totalPies * slicesPerPie;
```

Division uses the forward slash (**/**) symbol.

```
double moneyMadeFromGame = 100000;
double totalProgrammers = 4;
double moneyPerPerson = moneyMadeFromGame / totalProgrammers; // We're rich!
```

These last two examples show that you can do math with any numeric type, not just **`int`**. There are some complications when we intermix types in math expressions and or use the "small" integer types (**`byte`**, **`sbyte`**, **`short`**, **`ushort`**). For the moment, let's stick with a single type and avoid the small types. We'll address those problems before the end of this level.

COMPOUND EXPRESSIONS AND ORDER OF OPERATIONS

So far, our math expressions have involved only a single operator at a time. But like in the math world, our math expressions can combine many operators. For example, the following uses two different operations in a single expression:

```
int result = 2 + 5 * 2;
```

When this happens, the trick is understanding which operation happens first. If we do the addition first, the result is 14. If we do the multiplication first, the result is 12.

There is a set of rules that governs what operators are evaluated first. This ruleset is called the *order of operations*. There are two parts to this: (1) *operator precedence* determines which operation types come before others (multiplication before addition, for example), and (2) *operator associativity* tells you whether two operators of the same precedence should be evaluated from left to right or right to left.

Fortunately, C# steals the standard mathematical order of operations (to the extent that it can), so if you are familiar with the order of operations in math, it will all feel natural to you.

C# has many operators beyond just addition, subtraction, multiplication, and division, so the full ruleset is complicated. (See the Operators table in the back of the book for the whole picture.) For now, it is enough to say that the following two rules apply:

- Multiplication and division are done first, left to right.
- Addition and subtraction are done last, left to right.

With these rules, we can know that the expression **`2 + 5 * 2`** will evaluate the multiplication first, turning it into **`2 + 10`**, and then the addition is done next, for a final result of **`12`**, which is what is stored in **`result`**.

If you ever need to override the natural order of operations, there are two tools you can use. The first is to move the part you want to be done first to its own statement. Statements run from top to bottom, so doing this will force an operation to happen before another:

```
int partialResult = 2 + 5;
int result = partialResult * 2;
```

This trick is handy when a single math expression has grown too big to understand at a glance.

The other option is to use parentheses. Parentheses create a sub-expression that is evaluated first:

```
int result = (2 + 5) * 2;
```

Parentheses force the computer to do **`2 + 5`** before the multiplication. The math world uses this same trick.

In the math world, square brackets (**[** and **]**) and curly braces (**{** and **}**) are sometimes used as more "powerful" grouping symbols. C# uses those symbols for other things, so instead, you just use multiple sets of parentheses inside of each other:

```
int result = ((2 + 1) * 8 - (3 * 2) * 2) / 4;
```

Remember though: the goal isn't to cram it all into a single line, but to write code you'll be able to understand when you come back to it in two weeks.

Let's walk through another example. This code computes the area of a trapezoid:

```
// Some simple code for the area of a trapezoid
(http://en.wikipedia.org/wiki/Trapezoid)

double side1 = 4.5;
double side2 = 3.5;
double height = 1.5;

double areaOfTrapezoid = (side1 + side2) / 2.0 * height;
```

Parentheses are evaluated first, so we start by resolving the expression **`side1 + side2`**. Our program will retrieve the values in each of those variables and then perform the addition (a value of **`8`**). At this point, the overall expression could be thought of as the simplified **`8.0 / 2.0 * height`**. Division and multiplication have the same precedence, so we divide before we multiply because those are done left to right. **`8.0 / 2.0`** is **`4.0`**, and our expression is simplified again to **`4.0 * height`**. Multiplication is now the only operation left to address, so we perform it by retrieving the value in **`height`** (**`1.5`**) and multiplying for a final result of **`6.0`**. That is the value we place into the **`areaOfTrapezoid`** variable.

Challenge The Triangle Farmer 100 XP

As you pass through the fields near Arithmetica City, you encounter P-Thag, the triangle farmer, scratching numbers in the dirt.

"What is all of that writing for?" you ask.

"I'm just trying to calculate the area of all of my triangles. They sell by their size. The bigger they are, the more they are worth! But I have many triangles on my farm, and the math gets tricky, and I sometimes make mistakes. Taking a tiny triangle to town thinking you're going to get 100 gold, only to be told it's only worth three, is not fun! If only I had a program that could help me…." Suddenly, P-Thag looks at you with new eyes. "Wait just a moment. You have the look of a Programmer about you. Can you help me write a program that will compute my triangles' areas for me, so I can quit worrying about math mistakes and get back to tending to my triangles? The equation I'm using is this one here," he says, pointing to the formula, etched in stone beside him:

$$Area = \frac{base \times height}{2}$$

Objectives:

- Write a program that lets you input the triangle's base size and height.
- Compute the area of a triangle by turning the above equation into code.
- Write the result of the computation.

SPECIAL NUMBER VALUES

Each of the 11 numeric types—eight integer types and three floating-point types—defines a handful of special values you may find useful.

All 11 define a **MinValue** and a **MaxValue**, which is the minimum and maximum value they can correctly represent. These are essentially defined as variables (technically properties, which we'll learn more about in Level 20) that you get to through the type name. For example:

```
int aBigNumber = int.MaxValue;
short aBigNegativeNumber = short.MinValue;
```

These things are a little different than the methods we have seen in the past. They are more like variables than methods, and you don't use parentheses to use them. (They are actually properties, which we will discuss in Level 20.)

The **double** and **float** types (but not **decimal**) also define a value for positive and negative infinity called **PositiveInfinity** and **NegativeInfinity**:

```
double infinity = double.PositiveInfinity;
```

Many computers will use the **∞** symbol to represent a numeric value of infinity. This is the symbol used for infinity in the math world. Awkwardly, some computers (depending on operating system and configuration) may use the digit **8** to represent infinity in the console window. That can be confusing if you are not expecting it. (You can tweak settings to get the computer to do better.)

double and **float** also define a weird value called **NaN**, or "not a number." **NaN** is used when a computation results in an impossible value such as division by zero. You can refer to it as shown in the code below:

```
double notAnyRealNumber = double.NaN;
```

INTEGER DIVISION VS. FLOATING-POINT DIVISION

Try running this program and see if the displayed result is what you expected:

```
int a = 5;
int b = 2;
int result = a / b;
Console.WriteLine(result);
```

On a computer, there are two approaches to division. Mathematically, 5/2 is 2.5. If **a**, **b**, and **result** were all floating-point types, that's what would have happened. This division style is called floating-point division because it is what you get with floating-point types.

The other option is *integer division*. When you divide with any of the integer types, fractional bits of the result are dropped. This is different from rounding; even 9/10, which mathematically is 0.9, becomes a simple 0. The code above is dealing with only integers, and so it will use integer division. **5/2** becomes **2** instead of **2.5**, and that is what is placed into **result**.

This does take a little getting used to, and it will catch you by surprise from time to time. If you want integer division, use integers. If you want floating-point division, use floating-point types. Both have their uses. Just make sure you know which one you need and which one you've got.

Division by Zero

In the math world, division by zero is not defined—a meaningless operation without a specified result. When programming, you should also expect problems when dividing by zero. Once again, integer types and floating-point types have slightly different behavior here, though in both cases, the answer is still "bad things."

If you divide by zero with integer types, your program will produce an error that, if left unhandled, will crash your program. (We talk about error handling of this nature in Level 35.)

If you divide by zero with floating-point types, you do not get the same kind of crash. Instead, it assumes that you actually wanted to division with an impossibly tiny but non-zero number (an "infinitesimal" number), and the result will either be positive infinity, negative infinity, or NaN depending on whether the numerator was a positive number, negative number, or zero respectively. Mathematical operations with infinities and NaNs always result in more infinities and NaNs, so it is in your best interest to protect yourself against dividing by zero in the first place when you can.

More Operators

Addition, subtraction, multiplication, and division are not the only operators in C#. There are many more. We will cover a few more here and others throughout this book.

Unary + and - Operators

While **+** and **–** are typically used for addition and subtraction, which requires two operands (**a - b**, for example), both have a unary version, requiring only a single operand:

```
int a = 3;
int b = -a;
int c = +a;
```

For **-**, this indicates the negative version of the value. Since **a** is **3**, **-a** results in **-3**. It changes the sign of **a**. Or you could think of it as multiplying it by **-1**. That is, if **a** were **-5**, **b** would be assigned a value of **+5**. The sign is reversed.

For **+**, nothing actually changes for any of the numeric types we have seen so far. **+a** is the same as **a**. But the operator exists and use it when it adds clarity to the code (to contrast it with **-**). For example:

```
int a = 3;
int b = -(a + 2) / 4;
int c = +(a + 2) / 4;
```

The Remainder Operator

Suppose I bring 23 apples to the apple party (doctors beware) and you, me, and Johnny are at the party. There are two ways we could divide the apples. 23 divided 3 ways does not come out even. We could chop up the apples and have fractional apples (we'd each get 7.67 apples). Alternatively, if apple parts are not valuable (I don't want just a core!), then we can set aside anything that doesn't divide out evenly. This leftover amount is called the *remainder*. That is, each of the three of us would get 7 whole apples, with a remainder of 2.

C#'s *remainder operator* computes remainders in this same fashion using the **%** symbol. (Some call this the modulus operator or the mod operator, though those two terms mean slightly different things for negative numbers.) Computing the leftover apples looks like this in code:

```
int leftOverApples = 23 % 3;
```

The remainder operator may not seem all that useful at first glance, but it has its uses. In this book, we will see that one common use is to decide if some number is a multiple of another number. If so, the remainder would be 0. Consider this code:

```
int remainder = n % 2; // If this is 0, then 'n' is an even number.
```

If **`remainder`** is **`0`**, then the number is divisible by two—which also tells us that it is an even number.

The remainder operator has the same precedence as multiplication and division.

Challenge The Four Sisters and the Duckbear 100 XP

Four sisters own a chocolate farm and collect chocolate eggs laid by chocolate chickens every day. But more often than not, the number of eggs is not evenly divisible among the sisters, and everybody knows you cannot split a chocolate egg into pieces without ruining it. The arguments have grown more heated over time. The town is tired of hearing the four sisters complain, and Chandra, the town's Arbiter, has finally come up with a plan everybody can agree to. All four sisters get an equal number of chocolate eggs every day, and the remainder is fed to their pet duckbear. All that is left is to have some skilled Programmer build a program to tell them how much each sister and the duckbear should get.

Objectives:

- Create a program that lets the user enter the number of chocolate eggs gathered that day.
- Using **/** and **%**, compute how many eggs each sister should get and how many are left over for the duckbear.
- **Answer this question:** What are three total egg counts where the duckbear gets more than each sister does? Use the program you created to help you find the answer.

UPDATING VARIABLES

The **=** operator is the assignment operator, and while it looks the same as the equals sign, it does not imply that the two sides are equal. Instead, it indicates that some expression on the right side should be evaluated and then stored in the variable shown on the left.

It is common for variables to be updated with new values over time. It is also common for the expression used to determine the variable's new value to use the variable's current value. As an example, the following code increases the value of **a** by 1:

```
int a = 5;
a = a + 1; // the variable a will have a value of 6 after running this line.
```

That second line will cause **a** to grow by 1, regardless of what was previously in it.

The above code shows how assignment differs from the mathematical idea of equality. In the math world, $a = a + 1$ is an absurdity. No number exists that is equal to one more than itself. But in C# code, where this is simply a statement to update the variable based on its current

value, it is commonplace. It is so common that there are some shortcuts for it. Instead of doing **`a = a + 1;`**, we could do this instead:

```
a += 1;
```

This code is exactly equivalent to **`a = a + 1;`**, just shorter. The **+=** operator is called a *compound assignment operator* because it does some operation (addition, in this case) and a variable assignment. There are compound assignment operators for each of the binary operators we have seen so far, including **+=**, **-=**, ***=**, **/=**, and **%=**:

```
int a = 0;
a += 5; // The equivalent of a = a + 5; (a is 5 after this line runs.)
a -= 2; // The equivalent of a = a – 2; (a is 3 after this line runs.)
a *= 4; // The equivalent of a = a * 4; (a is 12 after this line runs.)
a /= 2; // The equivalent of a = a / 2; (a is 6 after this line runs.)
a %= 2; // The equivalent of a = a % 2; (a is 0 after this line runs.)
```

Increment and Decrement Operators

Adding one to a variable is called *incrementing* the variable, and subtracting one is called *decrementing* the variable. These two words are derived from the words *increase* and *decrease*. They move the variable up a notch or down a notch.

Incrementing and decrementing are so common (as we will soon see) that there are specific operators for adding one and subtracting one from a variable. These are the increment operator (**++**) and the decrement operator (**--**). These operators are both unary, requiring only a single operand to work, but it must be a variable and not an expression. For example:

```
int a = 0;
a++; // The equivalent of a += 1; or a = a + 1;
a--; // The equivalent of a -= 1; or a = a – 1;
```

We will see many uses for these operators shortly.

Challenge — The Dominion of Kings — 100 XP

Three kings, Melik, Casik, and Balik, are sitting around a table, debating who has the greatest kingdom among them. Each king rules an assortment of provinces, duchies, and estates. Collectively, they agree to a point system that helps them judge whose kingdom is greatest: Every estate is worth 1 point, every duchy is worth 3 points, and every province is worth 6 points. They just need a program that will allow them to enter their current holdings and compute a point total.

Objectives:

- Create a program that allows the user to enter how many provinces, duchies, and estates they have.
- Add up the user's total score, giving 1 point per estate, 3 per duchy, and 6 per province.
- Display the point total to the user.

Prefix and Postfix Increment and Decrement Operators

The way we used the increment and decrement operators above is the way they are typically used. However, assignment statements are also expressions and return the value assigned to the variable. Or at least, it does for normal assignment (with the **=** operator) and compound assignment operators (like **+=** and ***=**).

The same thing is true with the **++** and **--** operators, but with a little nuance. These two operators can be written before or after the variable that they modify. For example, you can write either **x++** or **++x**, and both will increment **x**. The first is called postfix notation, and the second is called prefix notation. When written as a complete statement (**x++;** or **++x;**), there is no meaningful difference between the two. But when you use them as part of an expression, **x++** evaluates to the *original* value of **x**, while **++x** evaluates to the *updated* value of **x**:

```
int x;

x = 5;
int y = ++x;

x = 5;
int z = x++;
```

Whether we do **x++** or **++x**, **x** is incremented and will have a value of **6** after each code block. But in the first part, **++x** will evaluate to **6** (increment first, then determine the value of **x**), meaning **y** will have a value of **6** as well. In contrast, in the second part, **x++** will evaluate to **5** (capture the current value of **x**, increment **x**, then produce the original value), and **z** will have a value of **5**.

The same logic applies to the **--** operator.

C# programmers rarely, if ever, use **++** and **--** as a part of an expression. It is far more common to use it as a standalone statement, so these nuances are rarely significant.

WORKING WITH DIFFERENT TYPES AND CASTING

Earlier, I said doing math that intermixes numeric types is problematic. Let's address that now.

Most math operations are only defined for operands of the same type. For example, addition is defined between two **int**s and two **double**s, but not between an **int** and a **double**.

But we often need to work with different data types in our programs. C# has a system of conversions between types. It allows one type to be converted to another type to facilitate mixing types.

There are two broad categories of conversions. A *narrowing conversion* is one that risks losing data in the conversion process. For example, converting a **long** to a **byte** could lose data if the number is larger than what a **byte** can accurately represent. In contrast, a *widening conversion* is one without the risk of losing information. A **long** can represent everything a **byte** can represent, so there is no risk in making the conversion.

Conversions can also be *explicit* or *implicit*. An explicit conversion is one that the programmer must specifically ask to happen. An implicit conversion is one that will occur automatically without the programmer stating it.

As a general rule, narrowing conversions, which run the risk of losing data, are always explicit. Widening conversions, which have no chance of losing data, are always implicit.

There are conversions defined among all of the numeric types in C#. When it is safe to do so, these are implicit conversions. When it is not safe, these are explicit conversions. Consider this code:

```
byte aByte = 3;
int anInt = aByte;
```

The simple expression **aByte** has a type of **byte**. Yet, it needs to be turned into an **int** to be stored in the variable **anInt**. Converting from a byte to an int is a safe, widening conversion, so the computer will make this conversion happen automatically. The code above compiles without you needing to do anything fancy.

If we are going the other way, from an **int** to a **byte**, the conversion is not safe, and for it to compile, we need to specifically state that we want to use the conversion, knowing the risk involved. To explicitly ask for a conversion, you use the *casting operator*, shown below:

```
int anInt = 3;
byte aByte = (byte)anInt;
```

The type to convert to is placed in parentheses before the expression to convert. This code says, "I know **anInt** is an **int**, but I can deal with any consequences of turning this into a **byte**, so please proceed with that conversion."

You are allowed to write out a specific request for an implicit conversion using this same casting notation (for example, **int anInt = (int)aByte;**), but it isn't strictly necessary.

There are conversions from every numeric type to every other numeric type in C#. When the conversion is a safe, widening conversion, they are implicit. When the conversion is a potentially dangerous narrowing conversion, they are explicit. For example, there is an implicit conversion from **sbyte** to **short**, **short** to **int**, and **int** to **uint**. Likewise, there is an implicit conversion from **byte** to **ushort**, **ushort** to **uint**, and **uint** to **ulong**. There is also an implicit conversion from any of the eight integer types to the floating-point types, not not the other way around.

Casting conversions of this nature are not defined for every possible type, however. For example, you cannot do this:

```
string text = "0";
int number = (int)text; // DOES NOT WORK!
```

There is no conversion defined (explicit or implicit) that goes from **string** to **int**. (But we can always use our friend **System.Convert** and do **int number = Convert.ToInt(text);.**)

Conversions and casting solve the two problems we noted earlier: math operations are not defined for the "small" types, and intermixing types cause issues.

Consider this code:

```
short a = 2;
short b = 3;
int total = a + b; // a and b are converted to ints automatically.
```

There is no addition defined for the **short** type, but one does exist for the **int** type. The computer will implicitly convert both to an **int** and use **int**'s **+** operation. This produces a result that is an **int**, not a **short**, so if we want to get back to a **short**, we need to cast it:

```
short a = 2;
short b = 3;
short total = (short)(a + b);
```

That last line brings up an important point: the casting operator has higher precedence than the other operators we have discussed. To let the addition happen first and the casting second,

we must put the addition in parentheses to force it to happen first. (We could have also separated the addition and the casting conversion onto two separate lines.)

Casting and conversions also fix the second problem that intermixing types can cause. Consider this code:

```
int amountDone = 20;
int amountToDo = 100;
double fractionDone = amountDone / amountToDo;
```

Since **amountDone** and **amountToDo** are both **int**s, the division is done as integer division, giving you a value of **0**. (Integer division ditches fractional values, and **0.2** becomes a simple **0**.) This **int** value of **0** is then implicitly converted to a **double** (0.0). But that's probably not what was intended. If we convert either of the parts involved in the division to a **double**, then the division happens with floating-point division instead:

```
int amountDone = 20;
int amountToDo = 100;
double fractionDone = (double)amountDone / amountToDo;
```

Now, the conversion of **amountDone** to a **double** is performed first. Division is not defined between a **double** and an **int**, but it is defined between two **double**s. The program knows it can implicitly convert **amountToDo** to a double to facilitate that. So amountToDo is "promoted" to a double, and now the division happens between two doubles using floating-point division, and the result is **0.2**. At this point, the expression is already a **double**, so no additional conversion must happen as it is assigned to the variable **fractionDone**.

Keeping track of how complex expressions work can be tricky. It gets easier with practice, but don't be afraid to separate parts onto separate lines to make it easier to think through.

OVERFLOW AND UNDERFLOW

In the math world, numbers can get as big as they need to. Mathematically, integers don't have an upper limit. But our data types do. A **byte** cannot get bigger than 255, and an **int** cannot represent the number 3 trillion. What happens when we try to surpass this (intentionally or accidentally)?

Consider this code:

```
short a = 30000;
short b = 30000;
short sum = (short)(a + b); // Too big to fit into a short. What happens?
```

Mathematically speaking, it should be 60000, but the computer gives a value of -5536.

When some operation causes a value to go beyond what its type can represent on the computer, it is called *overflow*. For integer types, this results in wrapping around back to the start of the range—0 for unsigned types and a large negative number for signed types. Stated differently, **int.MaxValue + 1** exactly equals **int.MinValue**. There is a danger in pushing the limits of a data type: it can lead to weird results. The original Pac-Man game had this issue when you go past level 255 (it must have been using a **byte** for the current level). The game went to an undefined level 0, which was glitchy and unbeatable.

Performing a narrowing conversion with a cast is a fast way to cause overflow, so cast wisely.

With floating-point types, the behavior is a little different. Since all floating-point types have a way to represent infinity, if you go too far up or too far down, the number will switch over to the type's positive or negative infinity representation. Math with infinities just results in more infinities (or NaNs), so even though the behavior is different from integer types, the consequences are just as significant.

Floating-point types have a second category of problems called *underflow*. With a **`float`**, the number 10000 can be correctly represented, as can 0.00001. In the math world, you can safely add those two values together to get 10000.00001. But a **`float`** cannot. It only has six or seven digits of precision and cannot distinguish 10000 from 10000.00001.

```
float a = 10000;
float b = 0.00001f;
float sum = a + b;
```

`sum` will also hold a value of **`10000`** after the addition. Underflow is not a problem in most situations, but every so often, especially when adding many tiny numbers, the lost digits begin to accumulate. In some cases, you can sidestep this by using a more precise type. For example, neither **`double`** nor **`decimal`** have trouble with this specific situation. But all three have it eventually, just at different times.

THE MATH AND MATHF CLASSES

C# also includes two classes, similar to the **`Console`** and **`Convert`** classes, but with the job of helping you do common math operations. These classes are called the **`System.Math`** class and the **`System.MathF`** class. We won't cover everything contained in them, but it is worth a brief overview.

π and e

The special, named numbers e and π are defined in **`Math`** so that you do not have to redefine them yourself (and run the risk of making a typo). These two numbers are **`Math.E`** and **`Math.PI`** respectively. For example, this code calculates the area of a circle (Area = πr^2):

```
double area = Math.PI * radius * radius;
```

Powers and Square Roots

C# does not have a power operator in the same way that it has multiplication and addition. But **`Math`** provides methods for doing both powers and square roots: the **`Pow`** and the **`Sqrt`** method:

```
double x = 3.0;
double xSquared = Math.Pow(x, 2);
```

`Pow` is the first method that we have seen that needs two pieces of information to do its job. The code above shows how to use these methods: everything goes into the parentheses, separated by commas.

For **`Pow`**, you must supply two pieces of information. First is the base, and second is the power to raise it to. So **`Math.Pow(x, 2)`** above is the same as x^2.

To do a square root, you use the **`Sqrt`** method:

```
double y = Math.Sqrt(xSquared);
```

Absolute Value

The *absolute value* of a number is merely the positive version of the number. The absolute value of 3 is 3. The absolute value of -4 is 4. The **Abs** method computes absolute values:

```
int x = Math.Abs(-2); // Will be 2.
```

Trigonometric Functions

The **Math** class also includes many trigonometric functions like sine, cosine, and tangent. It is beyond this book's scope to explain these trigonometric functions, but certain types of programs (including games) use them heavily. If you need them, the **Math** class is where to find them with the names **Sin**, **Cos**, and **Tan**. (There are others as well.) All expect angles in radians, not degrees.

```
double y1 = Math.Sin(0);
double y2 = Math.Cos(0);
```

Min, Max, and Clamp

The **Math** class also has methods for returning the minimum and maximum of two numbers:

```
int smaller = Math.Min(2, 10);
int larger = Math.Max(2, 10);
```

Here, **smaller** will contain a value of **2** while **larger** contains a value of **10**.

There is another related method that is convenient: **Clamp**. This allows you to provide a value and a range. If the value is within the range, that value is returned. If that value is lower than the range, it produces the low end of the range. If that value is higher than the range, it produces the high end of the range:

```
health += 10;
health = Math.Clamp(health, 0, 100); // Keep it in the interval 0 to 100.
```

More

This is a slice of some of the most widely used **Math** class methods, but there is more than what is listed here. Explore the choices when you have a chance so that you are familiar with the other options.

The MathF Class

The **MathF** class provides many of the same methods as **Math** but uses **float**s instead of **double**s. For example, **Math**'s **Pow** method expects **double**s as inputs and returns a **double** as a result. You can cast that result to a **float**, but **MathF** makes casting unnecessary:

```
float x = 3;
float xSquared = MathF.Pow(x, 2);
```

Level 8

Console 2.0

Speedrun

- The **`Console`** class can write a line without wrapping (**`Write`**), wait for just a single keypress (**`ReadKey`**), change colors (**`ForegroundColor`**, **`BackgroundColor`**), clear the entire console window (**`Clear`**), change the window title (**`Title`**), and play retro 80's sounds (**`Beep`**).
- Escape sequences start with a **`\`** and tell the computer to interpret the following letter differently. **`\n`** is a new line, **`\t`** is a tab, **`\"`** is a quote within a string literal.
- An **`@`** before a string ignores any would-be escape sequences: **`@"C:\Users\Me\File.txt"`**.
- A **`$`** before a string means curly braces contain code: **`$"a:{a} sum:{a+b}"`**.

We have come far in our journey. It is time to flesh out our knowledge of the console and learn some tricks to make working with text and the console window easier and more exciting. While a console window isn't as flashy as a GUI or a web page, it doesn't have to be boring.

The Console Class

We've been using the **`Console`** class since our very first Hello World program, but it is time we dug deeper into it to see what else it is capable of. **`Console`** has many methods and provides a few of its own variables (technically properties, as we will see in Level 20) that we can use to do some nifty things.

The `Write` Method

Aside from **`Console.WriteLine`**, another method called simply **`Write`**, does all the same stuff as **`WriteLine`**, without jumping to the next line when it finishes. There are many uses for this, but one I like is being able to ask the user a question and letting them answer on the same line:

```
Console.Write("What is your name, human? "); // Notice the space at the end.
string userName = Console.ReadLine();
```

The resulting program looks like this:

```
What is your name, human? RB
```

The **`Write`** method is also useful for putting many smaller bits of text onto a line in stages.

The `ReadKey` Method

The **`Console.ReadKey`** method does not wait for the user to push enter before completing. It waits for only a single keypress. So if you want to do something like a "Press any key to continue..." thing, you can use **`Console.ReadKey`**:

```
Console.WriteLine("Press any key when you're ready to begin.");
Console.ReadKey();
```

This code has a small problem. If a letter is typed, that letter will still show up on the screen. There is a way around this. The **`ReadKey`** method has two versions defined (called an "overload," but we'll cover that in more detail in Level 13). One version, shown above, has no inputs. The other version has an input whose type is **`bool`**, which indicates whether the text should be "intercepted" or not. If it is intercepted, it will not be displayed in the console window. Using this version looks like the following:

```
Console.WriteLine("Press any key when you're ready to begin.");
Console.ReadKey(true);
```

Changing Colors

The next few items we will talk about are not methods but properties. There are important differences between properties and variables, but for now, it is reasonable for us to just think of them as though they are variables.

The **`Console`** class also provides variables representing (and allow you to change) the colors it uses for displaying text. We're not stuck with just black and white! This is best illustrated with an example, so let's just dive in:

```
Console.BackgroundColor = ConsoleColor.Yellow;
Console.ForegroundColor = ConsoleColor.Black;
```

After assigning new values to these two variables, the console will begin using black text on a yellow background. **`BackgroundColor`** and **`ForegroundColor`** are both variables instead of methods, so we don't use parentheses as we have done in the past. These variables belong to the **`Console`** class, so we access them through **`Console.VariableName`** instead of just by variable name like other variables we have used. These lines assign a new value to those variables, though we have never seen anything like **`ConsoleColor.Yellow`** or **`ConsoleColor.Black`** before. **`ConsoleColor`** is an enumeration, a topic we will learn more about in Level 16. The short version is that an enumeration defines a set of values in a collection and gives them each a name. **`Yellow`** and **`Black`** are the names of two items in the **`ConsoleColor`** collection.

The `Clear` Method

After changing the console's background color, you may notice that it doesn't change the window's entire background, just the background of the new letters you write. You can use

`Console`'s **`Clear`** method to wipe out all text on the screen and changing the entire background to the newly set background color:

```
Console.Clear();
```

For better or worse, this does wipe out all the text currently on the screen (its primary objective, in truth), so you will want to ensure you do it only at the right moments.

Changing the Window Title

`Console` also has a **`Title`** variable, which will change the text displayed in the console window's title bar. Its type is a **`string`**.

```
Console.Title = "Hello World!";
```

Just about anything is better than the default name, which is usually nonsense like "C:\Users\RB\Source\Repos\HelloWorld\HelloWorld\bin\Debug\netcoreapp3.1\ HelloWorld.exe".

The `Beep` Method

The **`Console`** class can even beep! (Before you get too excited, the only sound the console window can make is a retro 80's square wave.) The **`Beep`** method makes the beep sound:

```
Console.Beep();
```

If you're musically inclined, there is a version that lets you choose both frequency and duration:

```
Console.Beep(440, 1000);
```

This **`Beep`** method needs two pieces of information to do its job. List each piece of information in the parentheses, separating them by commas.

The first item is the frequency. The higher the number, the higher the pitch, but 440 is a nice middle pitch. (The Internet can tell you which frequencies belong with which notes.) The second piece of information is the duration, measured in milliseconds (that is, 1000 is a full second, 500 is a half a second, etc.). You could imagine using **`Beep`** to play a simple melody, and indeed, some people have spent a lot of time doing just this and posting their code to the Internet.

SHARPENING YOUR STRING SKILLS

Let's turn our attention to a few features of strings to make them more powerful.

Escape Sequences

Here is a chilling challenge: how do you display a quote mark? This does not work:

```
Console.WriteLine("""); // ERROR: Bad quotation marks!
```

The compiler sees the first double quote as the start of a string and the second as the end. The third begins another string that never ends, and we get compiler errors.

An escape sequence is a sequence of characters that do not mean what they would usually indicate. In C#, you start escape sequences with the backslash character (**`\`**), located above the

<Enter> key on most keyboards. A backslash followed by a double quote (**\"**) instructs the compiler to interpret the character as a literal quote character within the string instead of interpreting it as the end of the string:

```
Console.WriteLine("\"");
```

The compiler sees the first quote mark as the string's start, the middle **\"** as a quote character within the text, and the third as the end of the string.

A quotation mark is not the only character you can escape. Here are a few other useful ones: **\t** is a tab character, **\n** is a new line character (move down to the next line), and **\r** is a carriage return (go back to the start of the line). (In the console window, going down a line with **\n** also goes back to the beginning of the line.)

So what if we want to have a literal **** character in a string? There's an escape sequence for the escape character as well: ****. This allows you to include backslashes in your strings:

```
Console.WriteLine("C:\\Users\\RB\\Desktop\\MyFile.txt");
```

That code displays the following:

```
C:\Users\RB\Desktop\MyFile.txt
```

In some instances, you do not care to do an escape sequence, and the extra slashes to escape everything are just in your way. You can put the **@** symbol before the text (called a *verbatim string literal*) to instruct the compiler to treat everything exactly as it looks:

```
Console.WriteLine(@"C:\Users\RB\Desktop\MyFile.txt");
```

String Interpolation

It is common to mix simple expressions among fixed text. For example:

```
Console.Write("My favorite number is " + myFavoriteNumber + ".");
```

This code uses the **+** operator with strings to combine multiple strings (often called *string concatenation* instead of addition). We first saw this in Level 3, and it is a useful tool. But with all of the different quotes and plusses, it can get hard to read. *String interpolation* allows you to embed expressions within a string by surrounding it with curly braces:

```
Console.WriteLine($"My favorite number is {myFavoriteNumber}.");
```

To use string interpolation, you put a **$** before the string begins. Within the string, enclose any expressions you want to evaluate inside of curly braces like **myFavoriteNumber** is above. It becomes a fill-in-the-blank game for your program to perform. Each expression is evaluated to produce its result. That result is then turned into a string and then placed in its spot in the overall text.

String interpolation usually gives you much more readable code, but be wary of many long expressions embedded into your text. Sometimes, it is better to compute a result and store it into a variable first.

You can combine string interpolation and verbatim strings by using **$** and **@** in either order.

Alignment

While string interpolation is powerful, it is only the beginning. Two other features make string interpolation even better: alignment and formatting.

Alignment lets you can display a string with a specific preferred width. Blank space is added before the value to reach the desired width if needed. Alignment is useful if you try to structure text in a table and need things to line up horizontally. To specify a preferred width, place a comma and the desired width in the curly braces after your expression to evaluate:

```
string name1 = Console.ReadLine();
string name2 = Console.ReadLine();
Console.WriteLine($"#1: {name1,20}");
Console.WriteLine($"#2: {name2,20}");
```

If my names were Steve and Captain America, the output would be:

```
#1:                Steve
#2:      Captain America
```

This code reserves 20 characters for the name's display. If the length is less than 20, it adds whitespace before it to achieve the desired width.

If you want the whitespace to be after the word, use a negative number:

```
Console.WriteLine($"{name1,-20} - 1");
Console.WriteLine($"{name2,-20} - 2");
```

This has the following output:

```
Steve                - 1
Captain America      - 2
```

There are two notable limitations to preferred widths. First, there is no convenient way to center the text. Second, if the text you are writing is longer than the preferred width, it won't truncate your text, but just keep writing the characters, which will mess up your columns. There are ways to work through both of these yourself manually, but there is nothing automatic to do it for you.

Formatting

With interpolated strings, you can also perform formatting. Formatting allows you to provide hints or guidelines about how you want to display data. Formatting is a deep subject that we won't exhaustively cover here, but let's look at a few examples.

You may have seen that when you display a floating-point number, it writes out lots of digits. For example, **`Console.WriteLine(Math.PI);`** displays **`3.141592653589793`**. You often don't care about all those digits and would rather round. The following instructs the string interpolation to write the number with three digits after the decimal place:

```
Console.WriteLine($"{Math.PI:0.000}");
```

Formatting requires putting a format string after a colon after the expression to evaluate. (This also comes after the preferred width if you use both.) This displays **`3.142`**. (It even rounds!)

Any **`0`** in the format indicates that you want a number to appear there even if the number isn't strictly necessary. For example, using a format string of **`000.000`** with the number **`42`** will display **`042.000`**.

In contrast, a **`#`** will leave a place for a digit but will not display a non-significant 0 (a leading or trailing 0):

```
Console.WriteLine($"{42:#.##}");// Displays "42"
Console.WriteLine($"{42.1234:#.##}");// Displays "42.12"
```

You can also use the **%** symbol to make a number be represented as a percent instead of a fractional value. For example:

```
float currentHealth = 4;
float maxHealth = 9;
Console.WriteLine($"{currentHealth/maxHealth:0.0%}"); // Displays "44.4%"
```

Several shortcut formats exist. For example, using just a simple **P** for the format is equivalent to **0.00%**, and **P1** is equal to **0.0%**. Similarly, a format string of **F** is the same as **0.00**, while **F5** is the same as **0.00000**.

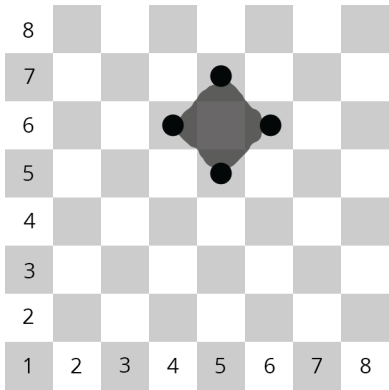
There are quite a few other symbols you can use for format strings, but that is enough to give us a basic toolset to work with as we move onto more challenging programs.

Challenge The Defense of Consolas 200 XP

The Uncoded One has begun an assault on the city of Consolas; the situation is dire. From a moving airship called the *Manticore*, massive fireballs capable of destroying city blocks are being catapulted into the city.

The city is arranged in blocks, numbered like a chessboard.

The city's only defense is a movable magical barrier, operated by a squad of four, that can protect a single city block by putting themselves in each of the target's four adjacent blocks, as shown in the picture to the right.



For example, to protect the city block at (Row 6, Column 5), the crew deploys themselves to (Row 6, Column 4), (Row 5, Column 5), (Row 6, Column 6), and (Row7, Column 5).

The good news is that if we can compute the deployment locations fast enough, the crew can be deployed around the target in time to prevent catastrophic damage to the city for as long as the siege lasts. The city of Consolas needs a program to tell the squad where to deploy, given the anticipated target. Something like this:

```
Target Row? 6
Target Column? 5
Deploy to:
(6, 4)
(5, 5)
(6, 6)
(7, 5)
```

Objectives:

- Ask the user for the target row and column.
- Compute the neighboring rows and columns of where to deploy the squad.
- Display the deployment instructions in a different color of your choosing.
- Change the window title to be "Defense of Consolas".
- Play a sound with **Console.Beep** when the results have been computed and displayed.

Glossary

.NET

The ecosystem that C# is a part of. It encompasses the .NET SDK, the compiler, the Common Language Runtime, Common Intermediate Language, the Base Class Library, and app models for building specific types of applications. (Levels 1 and 50.)

.NET Core

The original name for the current cutting-edge .NET implementation. After .NET Core 3.1, this became simply .NET 5. (Level 50.)

.NET Framework

The original implementation of .NET that worked only on Windows. This flavor of .NET is still broadly used, but most new development happens on the more modern .NET implementation. (Level 50.)

.NET Standard

A specification that defines what types should belong in the Base Class Library so that you can write code that runs on any of the various .NET implementations. (Levels 50.)

.NET Standard Library

See *.NET Standard*.

0-based Indexing

A scheme where indexes for an array or other collection type start with item number 0 instead of 1. C# uses this for almost everything.

Abstract Class

A class that you cannot create instances of; you can only create instances of derived classes. A class must be abstract to contain abstract members. (Level 26.)

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. (Level 26.)

Abstraction

The object-oriented concept where if a class keeps its inner workings private, those internal workings won't matter to the outside world. It also allows those inner workings to change without affecting the rest of the program. (Level 19.)

Accessibility Level

Types and their members indicate how broadly accessible or visible they are. The compiler will then ensure that other code uses it in a compliant manner. Making something more hidden gives you more flexibility to change it later without significantly affecting the rest of the program. Making something less hidden allows it to be used in more places. The **private** accessibility level means something can only be used within the type it is defined in. The `public` accessibility level means it can be used anywhere and is intended for general reuse. The `protected` accessibility level indicates that something can only be used in the class it is defined in or derived classes. The `internal` accessibility level indicates that it can be used in the assembly it is defined in, but not another. The `private protected` accessibility level indicates that it can only be used in derived classes in the same assembly. The `protected internal` accessibility level can be used in derived classes or the assembly it is defined in. (Levels 19, 25, and 47.)

Accessibility Modifier

See *accessibility level*.

Ahead-of-Time Compilation

C# code is compiled to CIL instructions by the C# compiler and then turned into hardware-ready binary instructions as the program runs with the JIT compiler. Ahead-of-time compilation moves the JIT compiler's work to the same time as the main C# compiler. This makes the code operating system- and hardware architecture-specific but speeds up initialization.

Anonymous Type

A class without a formal type name, created with the `new` keyword and a list of properties. E.g., **`new { A = 1, B = 2 }`**. They are immutable. (Level 20.)

AOT Compilation

See *ahead-of-time compilation*.

App Model

One of several frameworks that are a part of .NET, intended to make the development of a specific type of application (web, desktop, mobile, etc.) easy. (Level 50.)

Architecture

This word has many meanings in software development. For hardware architecture, see *Instruction Set Architecture*. For software architecture, see *object-oriented design*.

Argument

The value supplied to a method for one of its parameters.

Arm

A single branch of a switch. (Level 10.)

Array

A collection of multiple values of the same type placed together in a list-like structure. (Level 12.)

ASP.NET

An app model for building web-based applications. This book does not cover any app models in depth. (Level 50.)

Assembler

A simple program that translates assembly instructions into binary instructions. (Level 49.)

Assembly

Represents a single block of redistributable code used for deployment, security, and versioning. A *.dll* or *.exe* file. Each project is compiled into its own assembly. See also *project* and *solution*. (Level 3.)

Assembly Language

A low-level programming language where each instruction corresponds directly to a binary instruction the computer can run. Essentially, a human-readable form of binary. (Level 49.)

Assignment

The process of placing a value in a variable. (Level 5.)

Associative Array

See *dictionary*.

Associativity

See *operator associativity*.

Asynchronous Programming

Allowing work to be scheduled for later after some other task finishes to prevent threads from getting stuck waiting. (Level 44.)

Attribute

A feature for attaching metadata to code elements, which can then be used by the compiler and other code analysis tools. (Level 47.)

Auto-Property

A type of property where the compiler generates the backing field and basic get and set logic automatically. (Level 20.)

Automatic Memory Management

See *managed memory*.

Awaitable

Any type that can be used with the `await` keyword. `Task` and `Task<T>` are the most common. (Level 44.)

Backing Field

A field that a property uses as a part of its getter and setter. (Level 20.)

Base Class

In inheritance, the class that another is built upon. The derived class inherits all members (except constructors) from the base class. Also called a superclass or a parent class. See also *inheritance, derived class*, and *sealed class*. (Level 25.)

Base Class Library

The standard library available to all programs made in C# and other .NET languages. (Level 50.)

BCL

See *Base Class Library*.

Index

Symbols

0

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y