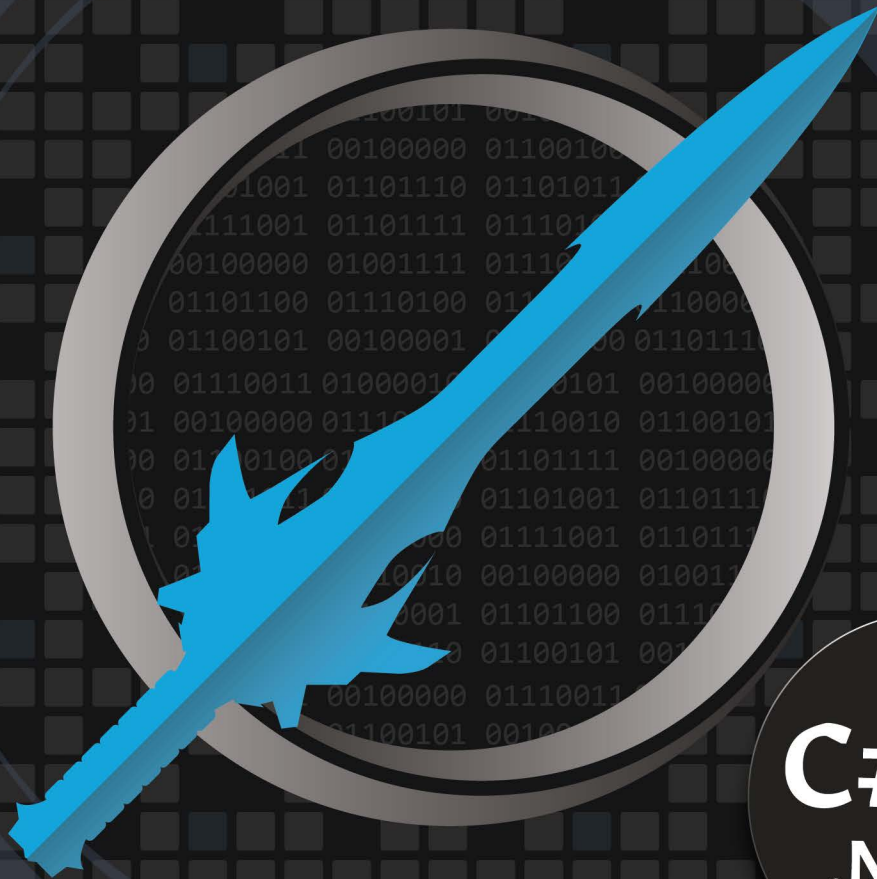


THE C# PLAYER'S GUIDE

FIFTH EDITION



RB WHITAKER

C# 10
.NET 6

THE
**C# PLAYER'S
GUIDE**

FIFTH EDITION



C# 10
.NET 6

RB WHITAKER

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the author and publisher were aware of those claims, those designations have been printed with initial capital letters or in all capitals.

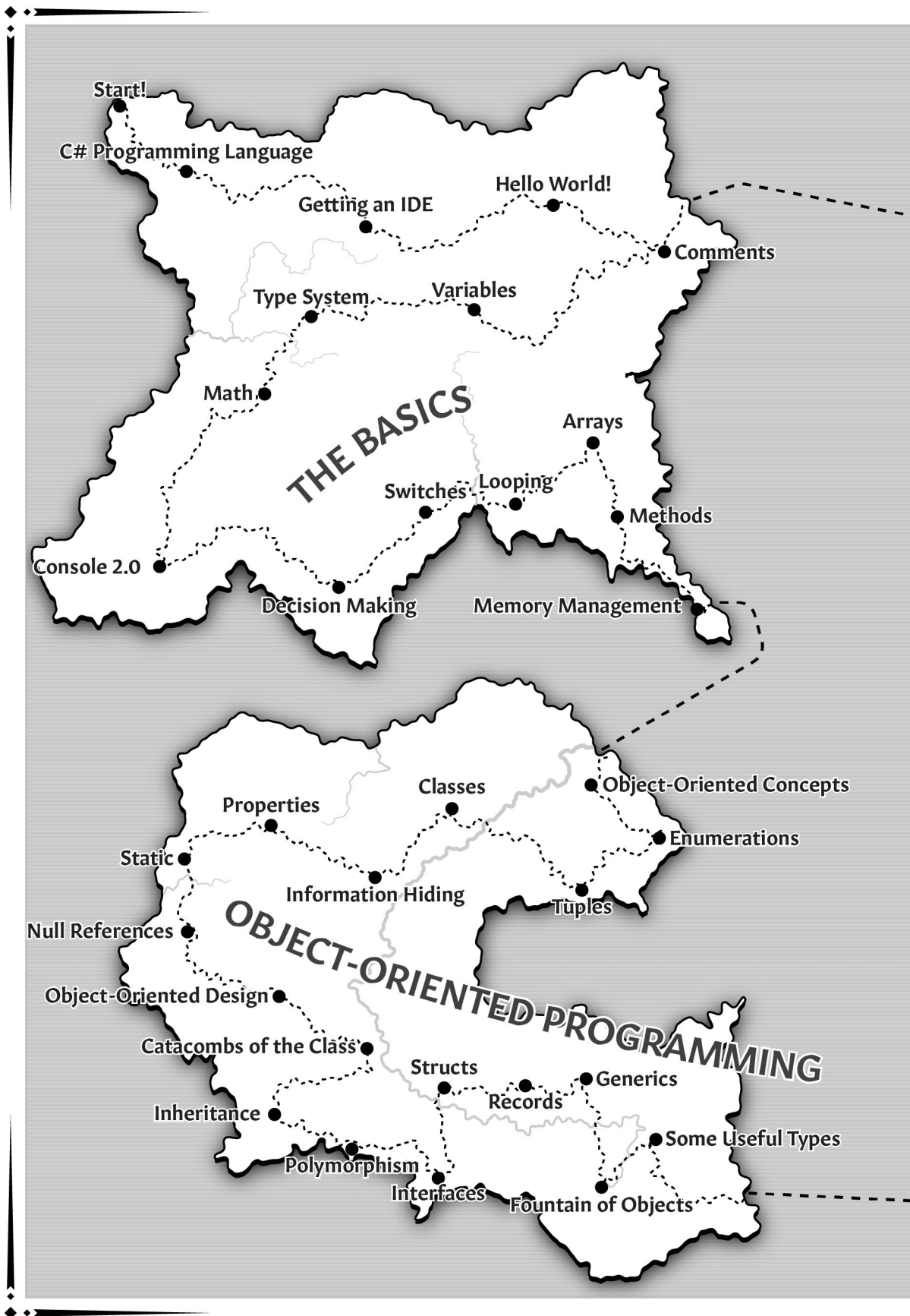
The author and publisher of this book have made every effort to ensure that this book's information was correct at press time. However, the author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

Copyright © 2012-2022 by RB Whitaker

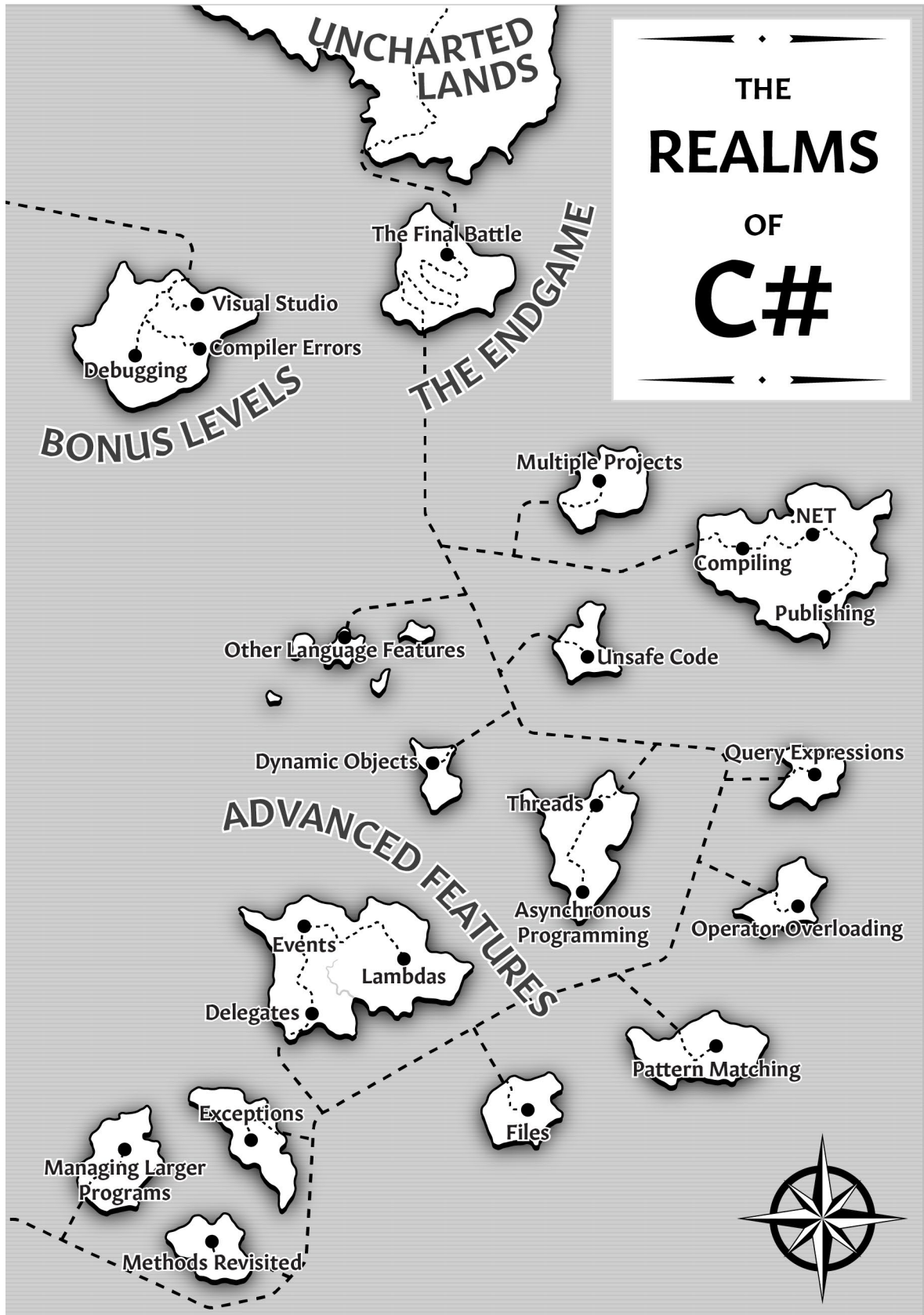
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the author, except for the inclusion of brief quotations in a review. For information regarding permissions, write to:

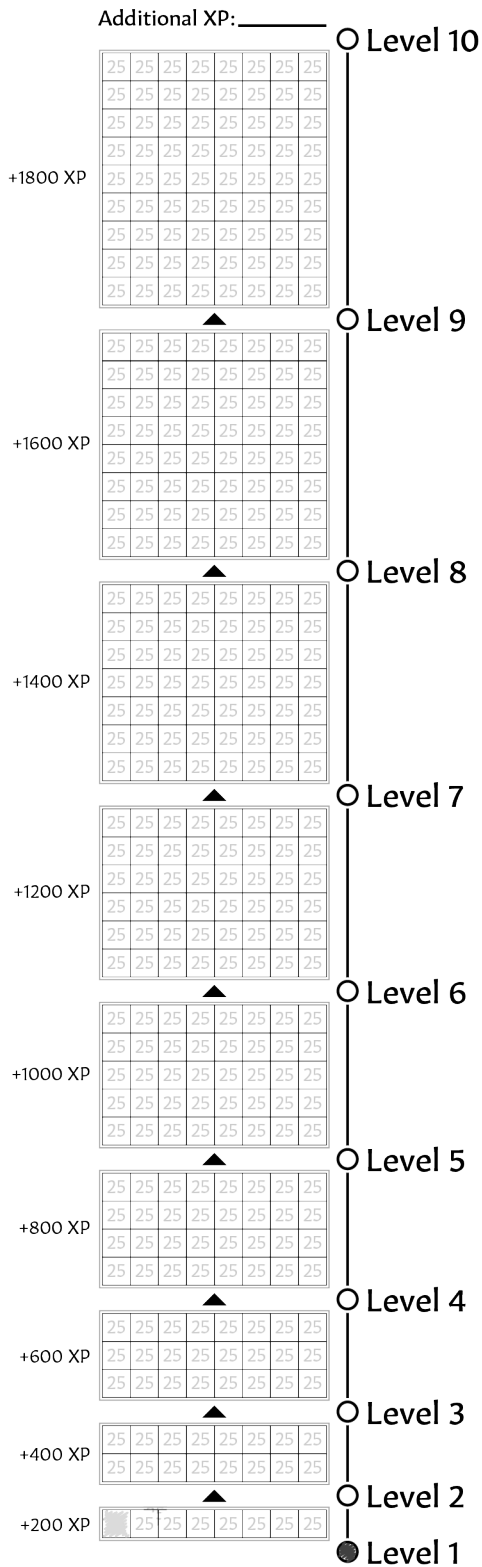
RB Whitaker
rbwhitaker@outlook.com

ISBN-13: 978-0-9855801-5-5



THE REALMS OF C#





Part 1: The Basics

✓ Page	Name	XP
<input type="checkbox"/> 10	Knowledge Check - C#	25
<input type="checkbox"/> 14	Install Visual Studio	75
<input type="checkbox"/> 19	Hello, World!	50
<input type="checkbox"/> 24	What Comes Next	50
<input type="checkbox"/> 24	The Makings of a Programmer	50
<input type="checkbox"/> 26	Consolas and Telim	50
<input type="checkbox"/> 31	The Thing Namer 3000	100
<input type="checkbox"/> 37	Knowledge Check - Variables	25
<input type="checkbox"/> 45	The Variable Shop	100
<input type="checkbox"/> 45	The Variable Shop Returns	50
<input type="checkbox"/> 48	Knowledge Check - Type System	25
<input type="checkbox"/> 53	The Triangle Farmer	100
<input type="checkbox"/> 56	The Four Sisters and the Duckbear	100
<input type="checkbox"/> 57	The Dominion of Kings	100
<input type="checkbox"/> 68	The Defense of Consolas	200
<input type="checkbox"/> 75	Repairing the Clocktower	100
<input type="checkbox"/> 78	Watchtower	100
<input type="checkbox"/> 82	Buying Inventory	100
<input type="checkbox"/> 83	Discounted Inventory	50
<input type="checkbox"/> 88	The Prototype	100
<input type="checkbox"/> 89	The Magic Cannon	100
<input type="checkbox"/> 94	The Replicator of D'To	100
<input type="checkbox"/> 95	The Laws of Freach	50
<input type="checkbox"/> 106	Taking a Number	100
<input type="checkbox"/> 107	Countdown	100
<input type="checkbox"/> 123	Knowledge Check - Memory	25
<input type="checkbox"/> 124	Hunting the Manticore	250

Part 2: Object-Oriented Programming

✓ Page	Name	XP
<input type="checkbox"/> 131	Knowledge Check - Objects	25
<input type="checkbox"/> 135	Simula's Test	100
<input type="checkbox"/> 143	Simula's Soups	100
<input type="checkbox"/> 153	Vin Fletcher's Arrows	100
<input type="checkbox"/> 162	Vin's Trouble	50
<input type="checkbox"/> 168	The Properties of Arrows	100
<input type="checkbox"/> 173	Arrow Factories	100
<input type="checkbox"/> 192	The Point	75
<input type="checkbox"/> 192	The Color	100
<input type="checkbox"/> 192	The Card	100
<input type="checkbox"/> 193	The Locked Door	100
<input type="checkbox"/> 193	The Password Validator	100
<input type="checkbox"/> 194	Rock-Paper-Scissors	150
<input type="checkbox"/> 195	15-Puzzle	150
<input type="checkbox"/> 195	Hangman	150
<input type="checkbox"/> 196	Tic-Tac-Toe	300
<input type="checkbox"/> 206	Packing Inventory	150

✓Page	Name	XP
<input type="checkbox"/> 210	Labeling Inventory	50
<input type="checkbox"/> 211	The Old Robot	200
<input type="checkbox"/> 218	Robotic Interface	75
<input type="checkbox"/> 226	Room Coordinates	50
<input type="checkbox"/> 232	War Preparations	100
<input type="checkbox"/> 241	Colored Items	100
<input type="checkbox"/> 243	The Fountain of Objects	500
<input type="checkbox"/> 245	Small, Medium, or Large	100
<input type="checkbox"/> 245	Pits	100
<input type="checkbox"/> 245	Maelstroms	100
<input type="checkbox"/> 246	Amaroks	100
<input type="checkbox"/> 246	Getting Armed	100
<input type="checkbox"/> 247	Getting Help	100
<input type="checkbox"/> 250	The Robot Pilot	50
<input type="checkbox"/> 252	Time in the Cavern	50
<input type="checkbox"/> 256	Lists of Commands	75

Part 3: Advanced Features

✓Page	Name	XP
<input type="checkbox"/> 269	Knowledge Check - Large Programs	25
<input type="checkbox"/> 270	The Feud	75
<input type="checkbox"/> 270	Dueling Traditions	100
<input type="checkbox"/> 276	Safer Number Crunching	50
<input type="checkbox"/> 278	Knowledge Check - Methods	25
<input type="checkbox"/> 278	Better Random	100
<input type="checkbox"/> 290	Exepti's Game	100
<input type="checkbox"/> 295	The Sieve	100
<input type="checkbox"/> 301	Knowledge Check - Events	25
<input type="checkbox"/> 302	Charberry Trees	100
<input type="checkbox"/> 307	Knowledge Check - Lambdas	25
<input type="checkbox"/> 307	The Lambda Sieve	50
<input type="checkbox"/> 315	The Long Game	100
<input type="checkbox"/> 324	The Potion Masters of Pattren	150
<input type="checkbox"/> 331	Knowledge Check - Operators	25
<input type="checkbox"/> 331	Navigating Operand City	100
<input type="checkbox"/> 332	Indexing Operand City	75
<input type="checkbox"/> 332	Converting Directions to Offsets	50
<input type="checkbox"/> 341	Knowledge Check - Queries	25
<input type="checkbox"/> 342	The Three Lenses	100
<input type="checkbox"/> 349	The Repeating Stream	150
<input type="checkbox"/> 359	Knowledge Check - Async	25
<input type="checkbox"/> 359	Asynchronous Random Words	150
<input type="checkbox"/> 360	Many Random Words	50
<input type="checkbox"/> 365	Uniter of Adds	75
<input type="checkbox"/> 366	The Robot Factory	100
<input type="checkbox"/> 372	Knowledge Check - Unsafe Code	25
<input type="checkbox"/> 392	Knowledge Check - Other Features	25
<input type="checkbox"/> 397	Colored Console	100

XP TRACKER

✓Page	Name	XP
<input type="checkbox"/> 398	The Great Humanizer	100
<input type="checkbox"/> 403	Knowledge Check - Compiling	25
<input type="checkbox"/> 408	Knowledge Check - .NET	25
<input type="checkbox"/> 413	Altar of Publication	100

Part 4: The Endgame

✓Page	Name	XP
<input type="checkbox"/> 419	Core Game: Building Character	300
<input type="checkbox"/> 420	Core Game: The True Programmer	100
<input type="checkbox"/> 420	Core Game: Actions and Players	300
<input type="checkbox"/> 421	Core Game: Attacks	200
<input type="checkbox"/> 421	Core Game: Damage and HP	150
<input type="checkbox"/> 422	Core Game: Death	150
<input type="checkbox"/> 422	Core Game: Battle Series	150
<input type="checkbox"/> 422	Core Game: The Uncoded One	100
<input type="checkbox"/> 423	Core Game: The Player Decides	200
<input type="checkbox"/> 423	Expansion: The Game's Status	100
<input type="checkbox"/> 424	Expansion: Items	200
<input type="checkbox"/> 424	Expansion: Gear	300
<input type="checkbox"/> 425	Expansion: Stolen Inventory	200
<input type="checkbox"/> 426	Expansion: Vin Fletcher	200
<input type="checkbox"/> 426	Expansion: Attack Modifiers	200
<input type="checkbox"/> 426	Expansion: Damage Types	200
<input type="checkbox"/> 427	Expansion: Making it Yours	?
<input type="checkbox"/> 428	Expansion: Restoring Balance	150

Part 5: Bonus Levels

✓Page	Name	XP
<input type="checkbox"/> 441	Knowledge Check - Visual Studio	25
<input type="checkbox"/> 446	Knowledge Check - Compiler Errors	25
<input type="checkbox"/> 451	Knowledge Check - Debugging	25

TABLE OF CONTENTS

Acknowledgments	xix
Introduction	1
The Great Game of Programming	1
Book Features	2
I Want Your Feedback	6
An Overview	6
 PART 1: THE BASICS	
1. The C# Programming Language	9
What is C#?	9
What is .NET?	10
2. Getting an IDE	11
A Comparison of IDEs	11
Installing Visual Studio	13
3. Hello World: Your First Program	15
Creating a New Project	15
A Brief Tour of Visual Studio	17
Compiling and Running Your Program	18
Syntax and Structure	19
Beyond Hello World	24
Compiler Errors, Debuggers, and Configurations	27
4. Comments	29
How to Make Good Comments	30

5. Variables	32
What is a Variable?	32
Creating and Using Variables in C#	33
Integers	34
Reading from a Variable Does Not Change It	35
Clever Variable Tricks	35
Variable Names	36
6. The C# Type System	38
Representing Data in Binary	38
Integer Types	39
Text: Characters and Strings	42
Floating-Point Types	43
The <code>bool</code> Type	45
Type Inference	46
The <code>Convert</code> Class and the <code>Parse</code> Methods	47
7. Math	50
Operations and Operators	50
Addition, Subtraction, Multiplication, and Division	51
Compound Expressions and Order of Operations	52
Special Number Values	54
Integer Division vs. Floating-Point Division	54
Division by Zero	55
More Operators	55
Updating Variables	56
Working with Different Types and Casting	58
Overflow and Roundoff Error	60
The <code>Math</code> and <code>MathF</code> Classes	61
8. Console 2.0	63
The <code>Console</code> Class	63
Sharpening Your String Skills	65
9. Decision Making	69
The <code>if</code> Statement	69
The <code>else</code> Statement	73
<code>else if</code> Statements	73
Relational Operators: <code>==</code> , <code>!=</code> , <code><</code> , <code>></code> , <code><=</code> , <code>>=</code>	74
Using <code>bool</code> in Decision Making	75
Logical Operators	76
Nesting <code>if</code> Statements	77
The Conditional Operator	77
10. Switches	79

Switch Statements	80
Switch Expressions	81
Switches as a Basis for Pattern Matching	82
11. Looping	84
The <code>while</code> Loop	84
The <code>do/while</code> Loop	86
The <code>for</code> Loop	86
<code>break</code> Out of Loops and <code>continue</code> to the Next Pass	87
Nesting Loops	88
12. Arrays	90
Creating Arrays	91
Getting and Setting Values in Arrays	91
Other Ways to Create Arrays	93
Some Examples with Arrays	94
The <code>foreach</code> Loop	95
Multi-Dimensional Arrays	95
13. Methods	97
Defining a Method	97
Calling a Method	99
Passing Data to a Method	101
Returning a Value from a Method	103
Method Overloading	104
Simple Methods with Expressions	105
XML Documentation Comments	106
The Basics of Recursion	107
14. Memory Management	109
Memory and Memory Management	110
The Stack	110
Fixed-Size Stack Frames	115
The Heap	115
Cleaning Up Heap Memory	122
 PART 2: OBJECT-ORIENTED PROGRAMMING	
15. Object-Oriented Concepts	129
Object-Oriented Concepts	129
16. Enumerations	132
Enumeration Basics	133
Underlying Types	136
17. Tuples	137

The Basics of Tuples	138
Tuple Element Names	139
Tuples and Methods	139
More Tuple Examples	140
Deconstructing Tuples	141
Tuples and Equality	142
18. Classes	144
Defining a New Class	145
Instances of Classes	147
Constructors	148
Object-Oriented Design	153
19. Information Hiding	155
The <code>public</code> and <code>private</code> Accessibility Modifiers	156
Abstraction	159
Type Accessibility Levels and the <code>internal</code> Modifier	160
20. Properties	163
The Basics of Properties	163
Auto-Implemented Properties	166
Immutable Fields and Properties	167
Object_INITIALIZER Syntax and <code>Init</code> Properties	168
Anonymous Types	169
21. Static	170
Static Members	170
Static Classes	173
22. Null References	174
Null or Not?	175
Checking for Null	176
23. Object-Oriented Design	178
Requirements	179
Designing the Software	180
Creating Code	185
How to Collaborate	187
Baby Steps	189
24. The Catacombs of the Class	191
The Five Prototypes	191
Object-Oriented Design	194
Tic-Tac-Toe	196
25. Inheritance	198
Inheritance and the <code>object</code> Class	199
Choosing Base Classes	201

Constructors	202
Casting and Checking for Types	204
The <code>protected</code> Access Modifier	205
Sealed Classes	205
26. Polymorphism	207
Abstract Methods and Classes	209
New Methods	210
27. Interfaces	212
Defining Interfaces	213
Implementing Interfaces	214
Interfaces and Base Classes	215
Explicit Interface Implementations	215
Default Interface Methods	216
28. Structs	219
Memory and Constructors	220
Classes vs. Structs	221
Built-In Type Aliases	225
Boxing and Unboxing	226
29. Records	228
Records	228
Advanced Scenarios	230
Struct- and Class-Based Records	231
When to Use a Record	232
30. Generics	233
The Motivation for Generics	233
Defining a Generic Type	236
Generic Methods	238
Generic Type Constraints	238
The <code>default</code> Operator	240
31. The Fountain of Objects	242
The Main Challenge	243
Expansions	245
32. Some Useful Types	248
The <code>Random</code> Class	249
The <code>DateTime</code> Struct	250
The <code>TimeSpan</code> Struct	251
The <code>Guid</code> Struct	252
The <code>List<T></code> Class	253
The <code>IEnumerable<T></code> Interface	256
The <code>Dictionary<TKey, TValue></code> Class	257

The Nullable<T> Struct	259
ValueTuple Structs	259
The StringBuilder Class	260
PART 3: ADVANCED TOPICS	
33. Managing Larger Programs	263
Using Multiple Files	263
Namespaces and using Directives	264
Traditional Entry Points	268
34. Methods Revisited	271
Optional Arguments	271
Named Arguments	272
Variable Number of Parameters	272
Combinations	273
Passing by Reference	273
Deconstructors	276
Extension Methods	277
35. Error Handling and Exceptions	280
Handling Exceptions	281
Throwing Exceptions	283
The finally Block	284
Exception Guidelines	285
Advanced Exception Handling	288
36. Delegates	291
Delegate Basics	291
The Action, Func, and Predicate Delegates	294
MulticastDelegate and Delegate Chaining	295
37. Events	296
C# Events	296
Event Leaks	300
EventHandler and Friends	300
Custom Event Accessors	301
38. Lambda Expressions	303
Lambda Expression Basics	303
Lambda Statements	305
Closures	306
39. Files	308
The File Class	308
String Manipulation	310
File System Manipulation	312

Other Ways to Access Files	313
40. Pattern Matching	316
The Constant Pattern and the Discard Pattern	317
The Monster Scoring Problem	317
The Type and Declaration Patterns	318
Case Guards	319
The Property Pattern	319
Relational Patterns	320
The <code>and</code> , <code>or</code> , and <code>not</code> Patterns	321
The Positional Pattern	321
The <code>var</code> Pattern	322
Parenthesized Patterns	322
Patterns with Switch Statements and the <code>is</code> Keyword	322
Summary	323
41. Operator Overloading	325
Operator Overloading	326
Indexers	327
Custom Conversions	329
42. Query Expressions	333
Query Expression Basics	334
Method Call Syntax	336
Advanced Queries	338
Deferred Execution	340
LINQ to SQL	341
43. Threads	343
The Basics of Threads	343
Using Threads	344
Thread Safety	347
44. Asynchronous Programming	351
Threads and Callbacks	352
Using Tasks	353
Who Runs My Code?	356
Some Additional Details	358
45. Dynamic Objects	361
Dynamic Type Checking	362
Dynamic Objects	362
Emulating Dynamic Objects with Dictionaries	363
Using <code>ExpandoObject</code>	363
Extending <code>DynamicObject</code>	364
When to Use Dynamic Object Variations	365

46. Unsafe Code	367
Unsafe Contexts	368
Pointer Types	368
Fixed Statements	369
Stack Allocations	370
Fixed-Size Arrays	370
The <code>sizeof</code> Operator	370
The <code>nint</code> and <code>nuint</code> Types	371
Calling Native Code with Platform Invocation Services	371
47. Other Language Features	373
Iterators and the <code>yield</code> Keyword	374
Constants	375
Attributes	376
Reflection	378
The <code>nameof</code> Operator	379
Nested Types	379
Even More Accessibility Modifiers	380
Bit Manipulation	380
<code>using</code> Statements and the <code>IDisposable</code> Interface	384
Preprocessor Directives	385
Command-Line Arguments	387
Partial Classes	387
The Notorious <code>goto</code> Keyword	388
Generic Covariance and Contravariance	389
Checked and Unchecked Contexts	391
Volatile Fields	392
48. Beyond a Single Project	393
Outgrowing a Single Project	393
NuGet Packages	396
49. Compiling in Depth	399
Hardware	399
Assembly	401
Programming Languages	401
Instruction Set Architectures	402
Virtual Machines and Runtimes	402
50. .NET	404
The History of .NET	404
The Components of .NET	405
Common Infrastructure	405
Base Class Library	406
App Models	407

51. Publishing	409
Build Configurations	409
Publish Profiles	410
 PART 4: THE ENDGAME	
52. The Final Battle	417
Overview	418
Core Challenges	419
Expansions	423
53. Into Lands Uncharted	429
Keep Learning	429
Where Do I Go to Get Help?	430
Parting Words	431
 PART 5: BONUS LEVELS	
A. Visual Studio	435
Windows	435
The Options Dialog	441
B. Compiler Errors	442
Code Problems: Errors, Warnings, and Messages	442
How to Resolve Compiler Errors	443
Common Compiler Errors	445
C. Debugging Your Code	447
Print Debugging	448
Using a Debugger	448
Breakpoints	449
Stepping Through Code	450
Breakpoint Conditions and Actions	451
 Glossary	452
Index	468

ACKNOWLEDGMENTS

It is hard to separate the 5th Edition from the 4th Edition when it comes to acknowledgments. The 4th Edition kept the bones of earlier editions but otherwise was a complete rewrite (twice!). Despite being 20 years old, C# 9 and 10 have changed the language in meaningful, exciting, and fundamental ways. Indeed, most random code you find on the Internet now looks like “old” C# code. These recent changes are somehow both tiny and game-changing. I don’t have a great way to measure, but I’ve often guessed that the 5th Edition is 98% the same as the 4th Edition. I might have even called this edition 4.1 if that were that a thing books did. Yet that last 2%, primarily reflecting C# 10 changes and the fast-evolving language, was enough to feel a new edition was not only helpful but necessary.

I want to thank the hundreds of people who joined Early Access for 4th and 5th Editions and the readers who have joined the book’s Discord server. The discussions I have had with you have changed this book for the better in a thousand different ways. With so many involved, I cannot thank everyone by name, though you all deserve it for your efforts. Having said that, UD Simon deserves special mention for providing me with a tsunami of suggestions and error reports week after week, rivaling the combined total of all other Early Access readers. The book is immeasurably better because of your collective efforts.

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. She has been patient with me as I’ve done five 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.

I owe all of 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 my will and creating worlds of living software—is entertaining and rewarding. 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. You have to work for it. You apply creativity and explore possibilities. You practice and gain abilities that help you win.

You'll be in good shape if you approach programming with this same mindset because programming requires this same set of skills. 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.

The “game” of programming is a massively multiplayer, open-world sandbox game with role-playing elements. By that, I mean:

- **Massively multiplayer:** While you may tackle specific problems independently, you are never alone. Most programmers are quick to share their knowledge and experience with others. This book and the Internet ensure you are not alone in your journey.
- **An open-world sandbox game:** You have few constraints or limitations; 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 also be so. 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 the challenges they will encounter. They provide time-saving tips and tricks and help prevent players from getting stuck anywhere for too long. This book attempts to be that player's guide for the Great Game of Programming in 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. This makes 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's 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 professional programmers 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. We'll learn some things to make console applications more colorful and exciting, but console applications are, admittedly, not the most exciting type of application.

Why not use something more exciting? The main reason is that regardless of whether you want to build games, smartphone apps, web apps, or desktop apps, the *starting points* in those worlds already expect you to know much about C#. For example, I just looked over the starter code for a certain C# game development framework. 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 in and 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 a better foundation. After building this foundation, learning how to make specific application types will go much more smoothly. Few will be satisfied with just console applications, but spending a few weeks covering the basics before moving on will make the learning process far easier.

BOOK FEATURES

Creating a fun and rewarding book (or at least not a dull and useless one) means adding some features that most programming books do not have. Let's look at a few of these 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 and Boss Battles

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:



When a challenge is especially tough, it is upgraded to a Boss Battle, shown by the icon below:



Boss Battles are sometimes split across multiple parts to allow you to work through them one step at a time.

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 also feel that these challenges should not be the *only* things you program as you learn, especially if you are relatively new to programming. Half of your programming time should come from these challenges and the rest 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 in this specific way."

As you do that, keep in mind the size of the challenges you are inventing for yourself. If you are learning how to draw, you don't go find millennia-old chapel ceilings to paint (or at least you don't expect it 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. And don't be afraid to dive in and make a mess. Continuing the art analogy, you aren't learning if you don't have a few garbage drawings in your sketchbook. Not every line of code you write will be a masterpiece. You have permission to write strange, ugly, and awkward code.

If these specific challenges are not your style, then skip them. But 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, covered in 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. 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 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 working solution, you're doing fine.

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, 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 mark off your progress as you go.

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

As you collect XP, you will accumulate enough points to level up from Level 1 to Level 10. If you reach Level 10, you will have completed nearly every challenge in this book and should have an excellent grasp of C# programming.

The XP assigned to each challenge is not random. Easier challenges have fewer points; more demanding challenges are worth more XP. While measuring difficulty is somewhat subjective, you can generally count on spending more time on challenges with more points and will gain a greater reward for it.

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, shadowy, 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 it felt shoehorned. Narrative sections supplement the descriptions in the challenges but otherwise have no purpose beyond advancing 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 is worth knowing (skilled C# programmers know all of it), some sections are more important than others. Sections that may be skipped in your first pass through this book are marked as Side Quests, indicated with the following icon:



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

Glossary

Programmers have a mountain of unique jargon and terminology. Beyond learning a new programming language, understanding this jargon is a second massive challenge for new programmers. To help you with this undertaking, I have carefully defined new concepts within the book as they arise 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: **<https://csharpplayersguide.com>**. Some of the highlights are below:

- **<https://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. This also contains more thorough explanations for all of the Knowledge Checks in the book.
- **<https://csharpplayersguide.com/errata>**. This page contains errata (errors in the book) that have been reported to clarify what was meant. If you notice something that seems wrong or inconsistent, you may find a correction here.
- **<http://csharpplayersguide.com/articles>**. This page contains a collection of articles that add to this book's content. They often cover more in-depth information beyond what I felt is reasonable to include in this book or answer questions readers have asked me. In a few places in this book, I call out articles with more information for the curious.

Discord

This book has an active Discord server where you can interact with me and other readers to discuss the book, ask questions, report problems, and get feedback on your solutions to the challenges. Go to **<https://csharpplayersguide.com/discord>** to see how to join the server. This server is a guildhall where you can rest from your travels and discuss C# with others on a similar journey as you.

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 past readers helped me know what parts were good and bad.

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 level 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 with your help, I can make it even better for those who follow in our footsteps. Don't hesitate to reach out to me, whether your feedback is positive or negative!

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

AN OVERVIEW

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

- **Part 1—The Basics.** This first part covers many of the fundamental elements of C# programming. It focuses on 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 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 bonus levels that guide you on what to do when you don't know what else to do—dealing with compiler errors and debugging your code. The glossary and index are also back here at the end of the book.

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 pace through Parts 1 and 2, skipping the Side Quests 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. Even if you skim or skip Part 3, you can still attempt the Final Battle in Part 4. If you're making consistent progress and getting good practice, it doesn't matter if your progress feels slow.

If you are an experienced programmer, you will likely be able to race through Part 1, slow down only a little in Part 2 as you learn how C# does object-oriented programming, and then spend most of your time in Part 3, learning the things that make C# unique.

Adapt the journey however you see fit. It is your book and your adventure!

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 what C# and .NET are (Level 1).
 - Install tools to allow us to program 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 which instructions will solve a problem. 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 bridge the two—precise enough for a computer to run but 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, 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 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 10.0, 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. It also allows for dynamic typing and unmanaged code in small doses when needed.

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, the 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 use it in a Visual Basic program. These languages have many similarities because of their shared ecosystem, and I'll point these out in some cases.



Knowledge Check

C#

25 XP

Check your knowledge with the following questions:

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

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. (Or you can go without one and use the raw tools directly; I don't recommend that for new programmers.) We will look at the most popular C# IDEs and discuss their strengths and weaknesses in this level.

We'll use an IDE to program in C#. Unfortunately, every IDE is different, and this book cannot cover them all. While this book focuses on the C# language and not a specific IDE, when necessary, this book will illustrate certain tasks using Visual Studio Community Edition. Feel free to use a different IDE. The C# language itself is the same regardless of which IDE you pick, but you may find slight differences when performing a task in the IDE. Usually, the process is intuitive, and if tinkering fails, Google usually knows.

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 used by most C# developers. Visual Studio is older than even C#, 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 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. But that’s almost certainly not you right now.

Visual Studio Community edition is my recommendation for new C# programmers running on Windows and is what this book uses throughout.

Visual Studio Code

Microsoft Visual Studio Code is a lightweight editor (not a fully-featured IDE) that works on Windows, Mac, and Linux. Visual Studio Code is free and has a vibrant community. It does not have the same expansive feature set as Visual Studio, and in some places, the limited feature set is harsh; you sometimes have to run 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 alien. Visual Studio Code is probably your best bet if Visual Studio isn’t an option for you (Linux and Mac, for example), especially if you have experience using the command line.

Visual Studio Code can also run online (vscode.dev), but as of right now, you can’t run your code. (Except by purchasing a codespace via github.com.) Perhaps this limitation will be fixed someday soon.

Visual Studio for Mac

Visual Studio for Mac 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 comparatively new, but JetBrains is very experienced at making IDEs for other languages. Rider does not have a free tier; 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.

Online Editors

There are a handful of online C# editors that you can use to tinker with C# without downloading tools. These have certain limitations and often do not keep up with the current language version. Still, you may find these useful if you just want to experiment without a huge commitment. An article on the book's website (csharpplayersguide.com/articles/online-editors) points out some of these.

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, you can skip the IDE. I do not recommend this approach for new programmers. It is a bit like building your car from parts before you can drive it. For the seasoned mechanic, this may be part of the enjoyment. Everybody else needs something that they can hop in and go. The IDEs above are in that category. Working without an IDE requires using the **dotnet** command-line tool to create, compile, test, and package your programs. Even if you use an IDE, you may still find the **dotnet** tool helpful. (If you use Visual Studio Code, you will *need* to use it occasionally.) But if you are new to programming, start with an IDE and learn the basics first.

INSTALLING VISUAL STUDIO

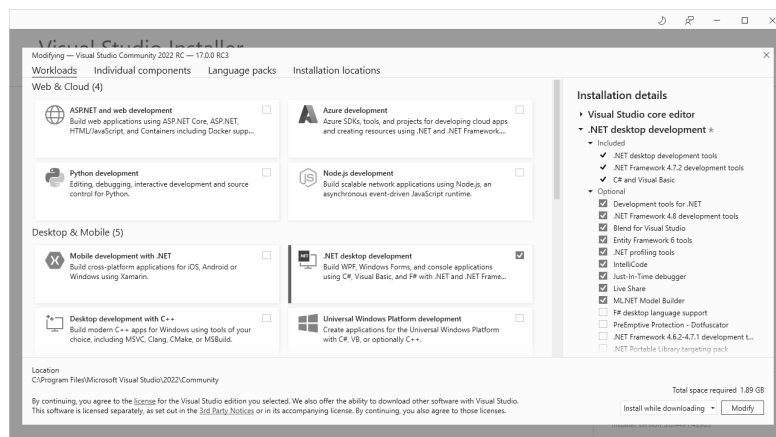
This book's focus is the C# language itself, but when I need to illustrate a task in an IDE, this book uses Visual Studio Community Edition. The Professional and Enterprise Editions should be identical. Other IDEs are usually similar, but you will find differences.

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

You can download Visual Studio Community Edition from <https://visualstudio.microsoft.com/downloads>. You will want to download Visual Studio 2022 or newer to use all of the features in this book.

Note that this will download the Visual Studio *Installer* rather than Visual Studio itself. The Visual Studio Installer lets you customize which components Visual Studio has installed. Anytime you want to tweak the available features, you will rerun the installer and make the desired changes.

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 all possible features of Visual Studio. In fact, for this book, we will only need a small slice of what Visual Studio offers.

You can install anything you find interesting, but there is only one item you *must* install for the code in this book. On the **Workloads** tab, find the one called **.NET desktop development** and click on it to enable it. If you forget to do this, you can always re-run the Visual Studio Installer and change what components you have installed.

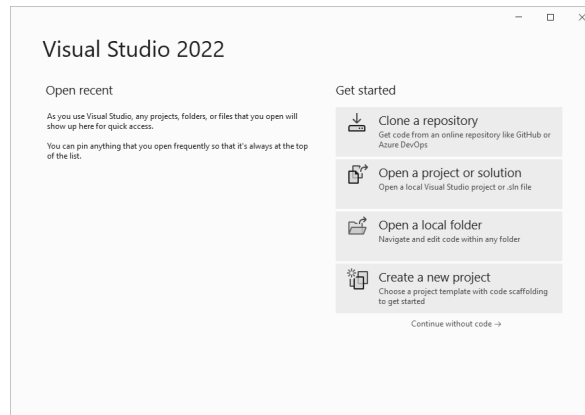
❗ **Warning! Be sure you get the right workload installed. If you don't, you won't be able to use all of the C# features described in this book.**

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 2022.

Visual Studio will ask you to sign in with a Microsoft account, even for the free Community Edition. You don't need to sign in if you don't want to, but it does enable a few minor features like synchronizing your settings across multiple devices.

If you are installing Visual Studio for the first time, you will also get a chance to pick development settings—keyboard shortcuts and a color theme. I have used the light theme in this book because it looks clearer in print. Many developers like 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 get your tools ready to start programming in C#. Install Visual Studio 2022 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: `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 the smallest meaningful program we could make. It gives us a glimpse of what the language looks like and verifies that our IDE is installed and working. *Hello World* is the traditional first program to make, and beginning anywhere else would make the programming gods mad. We don't want that!

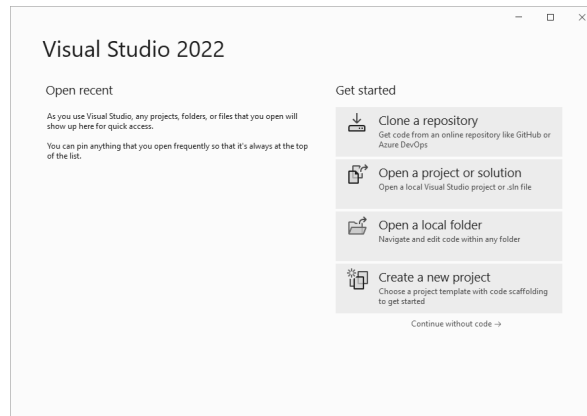
CREATING A NEW PROJECT

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 needed.

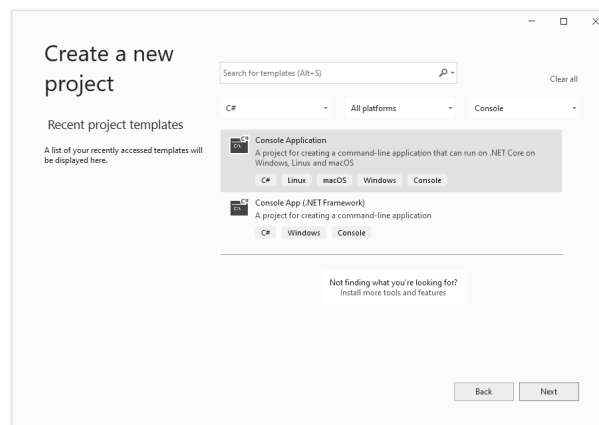
But most C# programs are started by being generated from one of several *templates*. Templates are standard starting points; they help you get the configuration right for specific project types and give you some starting code. We will use a template to create our projects.

You may be tempted to skip over this section, assuming you can just figure it out. Don't! There are several pitfalls here, so don't skip this section.

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:

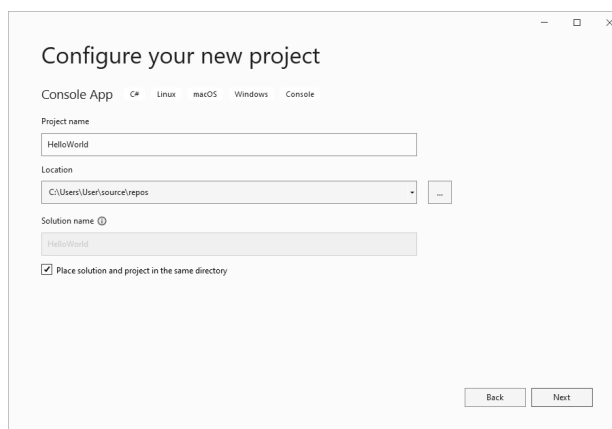


There are many templates to choose from, and your list might not exactly match what you see above. Choose the C# template called **Console Application**.

⚠ Warning! You want the C# project called Console Application. Ensure you aren't getting the Visual Basic one (check the tags below the description). Also, make sure you aren't getting the Console Application (.NET Framework) one, which is an older template. If you don't see this template, re-run the installer and add the right workload.

We will always use this Console Application template in this book, but you will use other templates as you progress in the C# world.

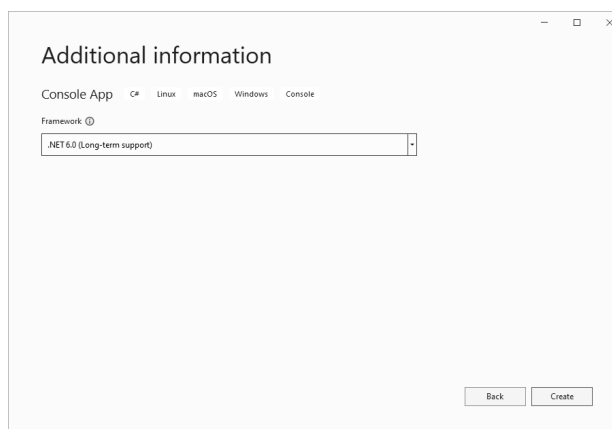
After choosing the C# **Console Application** template, press the **Next** button to advance to a page that lets you enter your new program's details:



Always give 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 note 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. For those, 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:



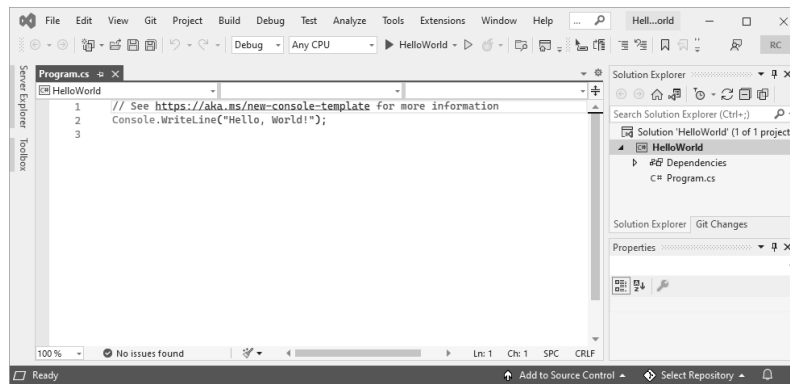
Make sure you pick **.NET 6.0** for this book! We will be using many .NET 6 features. You can change it after creation, but it is much easier to get it right in the first place.

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

❗ **Warning! Make sure you pick .NET 6.0 (or newer), so you can take advantage of all of the C# features covered in this book.**

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 becoming a Visual Studio expert. We won't get into every detail of Visual Studio, but we'll cover some essential elements here and throughout the book.

Right now, there are three things you need to know to get started. First, the big text editor on the left side 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 a 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 more 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, consider reading Bonus Level A before too long. It will give you a better feel for Visual Studio.

- ❗ **Time for a sanity check.** If you don't see code in the Code Window, double click on *Program.cs* in the Solution Explorer. Inspect the code you see in the Code Window. If you see `class Program` or `static void Main`, or if the file has more than a couple of lines of text, you may have chosen the wrong template. Go back and ensure you pick the correct template. If the right template isn't there, re-run the installer to add the right workload.

COMPILING AND RUNNING YOUR PROGRAM

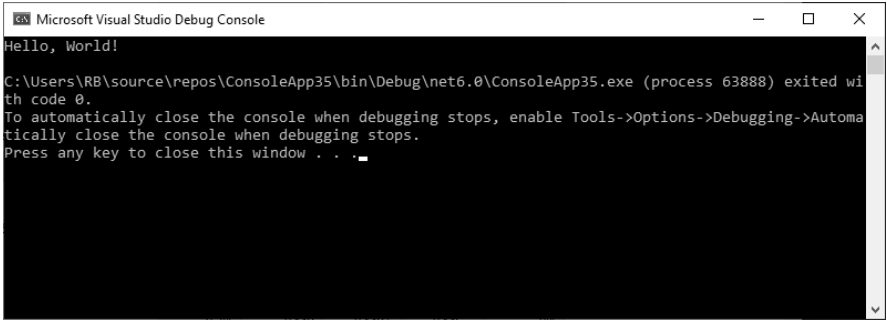
Generating a new project from the template has produced a complete program. Before we start dissecting it, let's run it.

The computer's circuitry cannot run C# code itself. It only runs low-level binary instructions formed out of 1's and 0's. So 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. The result is either a *.exe* or *.dll* file, which the computer can run. (This is an oversimplification, but it's accurate enough for now.)

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 6**.
- Run your program using any of the three methods described above.

SYNTAX AND STRUCTURE

Now that we’ve made and run our first C# program, it is time to look at the fundamental elements of all C# programs. We will touch on many topics in this section, but each is covered in more depth later in this book. You don’t need to master it all here.

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*.

Look in your Code Editor window to find the text shown below:

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

You might also see a line with green text that starts with two slashes (`//`). That is a *comment*. We'll talk about comments in Level 4, but you can ignore or even delete that line for now.

We're going to analyze this one-line program in depth. As short as it is, it reveals a great deal about how C# programming works.

Strings and Literals

First, the `"Hello, World!"` part is the displayed text. You can imagine changing this text to get the program to show something else instead.

In the programming world, we often use the word *string* to refer to text for reasons we'll see later. There are many ways we can work with strings or text, but this is the simplest. This is called a *literal*, or specifically, a *string literal*. A *literal* is a chunk of code that defines some specific value, precisely as written. Any text in double quotes will be a string literal. The quote marks aren't part of the text. They just indicate where the string literal begins and ends. Later on, we'll see how to make other types of literals, such as number literals.

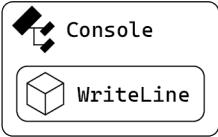
Identifiers

The two other big things in our code are `Console` and `WriteLine`. These are known formally as *identifiers* or, more casually, as *names*. An identifier allows you to refer to some existing code element. As we build code elements of our own, we will pick names for them as well, so we can refer back to them. `Console` and `WriteLine` both refer to existing code elements.

Hierarchical Organization

Between `Console` and `WriteLine`, there is a period (`.`) character. This is called the *member access operator* or the *dot operator*. Code elements like `Console` and `WriteLine` are organized hierarchically. Some code elements live inside of other code elements. They are said to be *members* or *children* of their container. The dot operator allows us to dig down in the hierarchy, from the big parts to their children.

In this book, I will sometimes illustrate this hierarchical organization using a diagram like the one shown below:



I'll refer to this type of diagram as a *code map* in this book. Some versions of Visual Studio can generate similar drawings, but I usually sketch them by hand if I need one.

These code maps can help us see the broad structure of a program, which is valuable. Equally important is that a code map can help us understand when a specific identifier can be used. The compiler must determine which code element an identifier refers to. This process is called *name binding*. But don't let that name scare you. It really is as simple as, "When the code says, `WriteLine`, what exactly is that referring to?"

Only a handful of elements are globally available. We can start with `Console`, but we can't just use `WriteLine` on its own. The identifier `WriteLine` is only available in the context of its container, `Console`.

Classes and Methods

You may have noticed that I used a different icon for **Console** and **WriteLine** in the code map above. Named code elements come in many different flavors. Specifically, **Console** is a *class*, while **WriteLine** is a *method*. C# has rules that govern what can live inside other things. For example, a class can have methods as members, but a method cannot have a class as a member.

We'll talk about both methods and classes at great length in this book, but let's start with some basic definitions to get us started.

For now, think of classes as entities that solve a single problem or perform a specific job or role. It is like a person on a team. The entire workload is spread across many people, and each one performs their job and works with others to achieve the overarching goal. The **Console** class's job is to interact with the console window. It does that well, but don't ask it to do anything else—it only knows how to work with the console window.

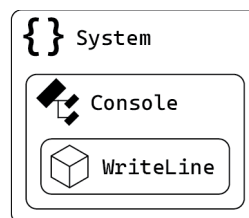
Classes are primarily composed of two things: (1) the data they need to do their job and (2) tasks they can perform. These tasks come in the form of methods, and **WriteLine** is an example. A method is a named, reusable block of code that you can request to run. **WriteLine**'s task is to take text and display it in the console window on its own line.

The act of asking a method to run is called *method invocation* or a *method call*. These method calls or invocations are performed by using a set of parentheses after the method name, which is why our one line of code contains **WriteLine(...)**.

Some methods require data to perform their task. **WriteLine** works that way. It needs to know what text to display. This data is supplied to the method call by placing it inside the parentheses, as we have seen with **WriteLine("Hello, World!")**. Some methods don't need any extra information, while others need multiple pieces of information. We will see examples of those soon. Some methods can also *return* information when they finish, allowing data to flow to and from a method call. We'll soon see examples of that as well.

Namespaces

All methods live in containers like a class, but even most classes live in other containers called namespaces. Namespaces are purely code organization tools, but they are valuable when dealing with hundreds or thousands of classes. The **Console** class lives in a namespace called **System**. If we add this to our code map, it looks like this:



In code, we could have referred to **Console** through its namespace name. The following code is functionally identical to our earlier code:

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

Using C# 10 features and the project template we chose, we can skip the **System**. In older versions of C#, we would have somehow needed to account for **System**. One way to account

for it was shown above. A second way is with a special line called a **using** directive. If you stumble into older C# code online or elsewhere, you may notice that most old C# code files start with a pile of lines that look like this:

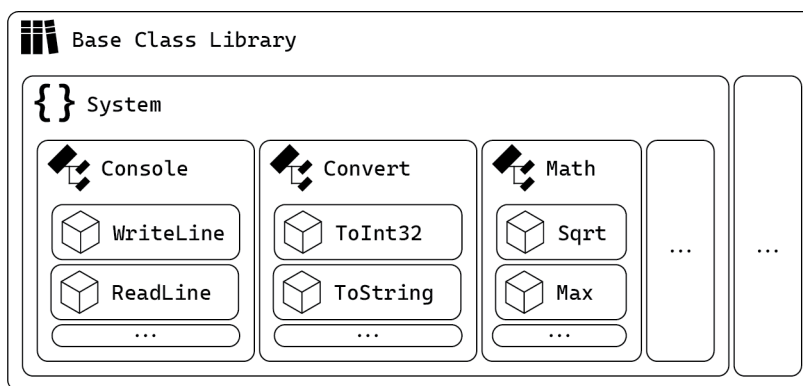
```
using System;
```

These lines tell the compiler, “If you come across an identifier, look in this namespace for it.” It allows you to use a class name without sticking the namespace name in front of it. But with C# 10, the compiler will automatically search **System** and a handful of other extremely common namespaces without you needing to call it out.

For the short term, we can *almost* ignore namespaces entirely. (We’ll cover them in more depth in Level 33.) But namespaces are an important element of the code structure, so even though it will be a while before we need to deal with namespaces directly, I’m still going to call out which namespaces things live in as we encounter them. (Most of it will be the **System** namespace.)

The Base Class Library

Our code map is far from complete. **System**, **Console**, and **WriteLine** are only a tiny slice of the entire collection of code called the *Base Class Library* (BCL). The Base Class Library contains many namespaces, each with many classes, each with many members. The code map below fleshes this out a bit more:

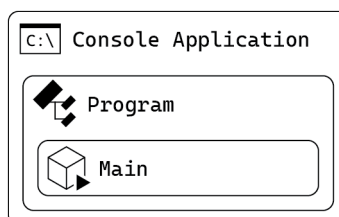


It is huge! If we drew the complete diagram, it might be longer than this whole book!

The Base Class Library provides every C# program with a set of fundamental building blocks. We won’t cover every single method or class in the Base Class Library, but we will cover its most essential parts throughout this book (starting with **Console**).

Program and Main

The code we write also adds new code elements. Even our simple Hello World program adds new code elements that we could show in a code map:



The compiler takes the code we write, places it inside a method called **Main**, and then puts that inside a class called **Program**, even though we don't see those names in our code. This is a slight simplification; the compiler uses a name you can't refer to (**<Main>\$**), but we'll use the simpler name **Main** for now.

In the code map above, the icon for **Main** also has a little black arrow to indicate that **Main** is the program's *entry point*. The entry point or *main method* is the code that will automatically run when the computer runs your program. Other methods won't run unless the main method calls them, as our Hello World program does with **WriteLine**.

In the early days of C#, you had to write out code to define both **Program** and **Main**. You rarely need to do so now, but you can if you want (Level 33).

Statements

We have accounted for every character in our Hello World program except the semicolon (;) at the end. The entire **Console.WriteLine("Hello, World!");** line is called a *statement*. A statement is a single step or command for the computer to run. Most C# statements end with a semicolon.

This particular statement instructs the computer to ask the **Console** class to run its **WriteLine** method, giving it the text **"Hello, World!"** as extra information. This "ask a thing to do a thing" style of statement is common, but it is not the only kind. We will see others as we go.

Statements don't have names, so we won't put them in a code map.

Statements are an essential building block of C# programs. You instruct the computer to perform a sequence of statements one after the next. Most programs have many statements, which are executed from top to bottom and left to right (though C# programmers rarely put more than one statement on a single line).

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 instructions and 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, it isn't that the computer made a mistake. It means what you *thought* you commanded and what you *actually* commanded were not the same. As a new programmer, it is easy to think, "The computer isn't doing what I told it!" Instead, try to train your mind to think, "Why did the computer do that instead of what I expected?" You will be a better programmer with that mindset.

Whitespace

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:

```
(           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 will tolerate.**

**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:

- Change your program to say something besides “Hello, World!”

BEYOND HELLO WORLD

With an understanding of the basics behind us, let’s explore a few other essential features of C# and make a few more complex programs.

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:

```
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 with different data. 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 their value. We use the same word in the math world to refer

to something like $3 + 4$ or -2×4.5 . Expressions describe how to produce a value from smaller elements. The computer can use an expression to compute a value as it runs.

C# programs use expressions heavily. Anywhere a value is needed, an expression can be put in its place. While we could do 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 bits of text to produce the combined text (`"Hi User"`).

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

Expressions are powerful because they can be assembled out of other, smaller expressions. You can think of a single value like `"Hi User"` as the simplest type of expression. But if we wanted, we could split `"User"` into `"Us" + "er"` or even 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.

Every expression, once evaluated, will result in a single new value. That single value can be used in other expressions or other parts of your code.

Variables

Variables are containers for data. They are called variables because their contents can change or vary as the program runs. Variables allow us to store data for later use.

Before using 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 data in the variable to use later. 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. All of this is shown below:

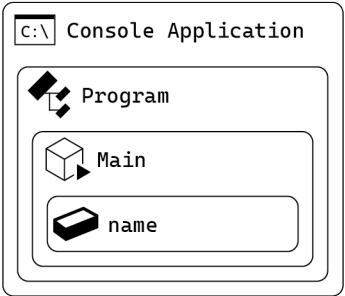
```
string name;  
name = "User";  
Console.WriteLine("Hi " + name);
```

The first line declares the variable with a type and a name. Its type is **string** (the fancy programmer word for text), and its name is **name**. This line ensures we have a place to store text that we can refer to with the identifier **name**.

The second line assigns it a value of `"User"`.

We use the variable in an expression on the final line. As your program runs, it will evaluate the expression `"Hi " + name` by retrieving the current value in the **name** variable, then combining it with the value of `"Hi "`. We'll see plenty more examples of expressions and variables soon.

Anything with a name can be visualized on a code map, and this **name** variable is no exception. The following code map shows this variable inside of **Main**, using a box icon:



In Level 9, we’ll see why it can be helpful to visualize where variables fit on the code map.

You may notice that when you type **string** in your editor, it changes to a different color (usually blue). That is because **string** is a *keyword*. A keyword is a word with special meaning in a programming language. C# has over 100 keywords! We’ll discuss them all as we go.

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 a person 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 text it sends back can be stored in a variable or used in an expression:

```
string name;
Console.WriteLine("What is your 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 greets them by name.

When a method produces a value, programmers say 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 are now crumbling 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 to help.

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 lets 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: compiler errors, debugging, and build configurations. These are more about how programmers construct C# programs than the language itself.

Compiler Errors and Warnings

As you write C# programs, you will sometimes accidentally write code that the compiler cannot figure out. The compiler will not be able to transform your code into something the computer can understand.

When this happens, you will see two things. When you try to 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.

Sometimes, the problem and its solution are apparent. 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.

If you're watching your code closely, you might have already seen the compiler error's less-scary cousin: the compiler warning. A compiler warning means the compiler can make the code work, but it thinks it is suspicious. For example, when we do something like `string name = Console.ReadLine();`, you may have noticed that you get a warning that states, "Converting null literal or possible null value to a non-nullable type." That code even has a green squiggly mark under it to highlight the potential problem.

This particular warning is trying to tell you that `ReadLine` may not give you *any* response back (a lack of value called `null`, which is distinct from text containing no characters). We'll learn how to deal with these missing values in Level 22. For now, you can ignore this particular compiler warning; we won't be doing anything that would cause it to happen.

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, jump over and read this whenever you have an interest or a need.

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. There are two configurations defined by default, and you rarely 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. For 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 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. They can provide information to help 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 bits of text placed in your program, meant to be annotations on the code for humans—you and other programmers. The compiler ignores comments.

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 later.
- 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 and know how it works without needing to study 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. You can temporarily turn the code into a comment until you're 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 add 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 (//). After these two slashes, everything on the line will become a comment, which the compiler will pretend doesn't 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 those two placements. My general rule is to put important comments above the code and use the second placement (on the same line) only for side notes about that line of code.

You can also make a comment by putting it between a `/*` and `*/`:

```
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! */ + name);
```

That second example is awkward but has its uses, such as when 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 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 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 to be 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 easy to understand first, then add just enough comments to clarify any important but unobvious 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:

```
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 else could you 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.
-

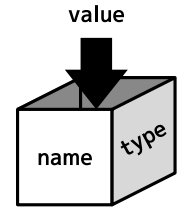
In this level, we will look at variables in more depth. We will also look at some rules around good variable names.

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 access any specific location's contents. But remembering what's in spot #45387 is not practical. Data comes and goes in a program. We might need something for a split second or 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, type, and contents or value. 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 for it in memory of the appropriate size.

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 only work 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 retrieve its current value. The purpose of saving the data was to come back to it later. 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();`).

Retrieving 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, variable declarations tend to gravitate toward the top of the 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 represents something else—say a favorite color—then it is usually 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, value, and expression in your C# programs has a type associated with it. Before now, the only type we have seen has been **strings** (text). But many other types exist, and we can even define our own types. Let's look at a second type: **int**, which represents an integer.

An integer is a whole number (no fractions or decimals) but either positive, negative, or zero. Given the computer's capacity to do math, it should be no surprise that storing numbers is common, 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 concept is important, so I'll state it again: types matter in C#; every value, variable, and expression 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. It is a string literal, even though the characters could be used in numbers. Anything in double quotes will always be a string. To make an **int** literal, you write the number without the quote marks:

```
score = 0; // 0 is an int literal.
```

After this line of code runs, the **score** variable—a memory location reserved to hold **ints** 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:

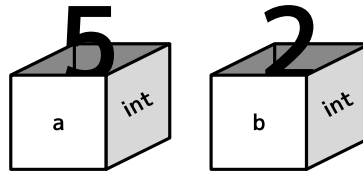
```
score = 4;  
score = 11;  
score = -1564;
```

READING FROM A VARIABLE DOES NOT CHANGE IT

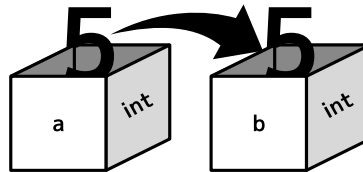
When you read the contents of a variable, the variable's contents are copied out. To illustrate:

```
int a;  
int b;  
  
a = 5;  
b = 2;  
  
b = a;  
a = -3;
```

In the first two lines, **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. **b = a** does not mean **a** and **b** are now always going to be equal! That **=** symbol means assignment, not equality. (Though **a** and **b** will be equal immediately after running that line until something changes.) 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 additional nuances to variable assignment, which we will cover 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**, which is 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 very common, 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 built to allow 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, but the language has a few rules:

1. Variable names must start with a letter or the underscore character (`_`). But C# casts a wide net when defining “letters”—almost anything in any language is allowed. `taco` and `_taco` are 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 whitespace characters are banned because they make it impossible for the compiler to know where a variable name ends and other code begins. (For 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 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 "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, *all* words begin with a capital letter. The big capital letter in the middle of the word makes it look like a camel's hump, which is why it has this name. Most C# programmers use **lowerCamelCase** for variables and **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

Variables

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. Can you redeclare a variable?
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) No. (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 `Convert` class helps convert 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, *anything* we want to represent in a program requires 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 an integer 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. They are not the same. 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 entirely 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 deal well 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# actually 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 this would be a workable scheme.

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 using more memory or store smaller numbers while conserving memory.

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

In contrast, the **short** type uses 2 bytes and can represent numbers between about -32,000 and +32,000. The **long** type uses 8 bytes and can represent numbers between about -9 quintillion and +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 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 only need positive numbers, you could imagine shifting these ranges upward to exclude 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 0 to about 65,000, **uint**'s range is 0 to about 4 billion, and **ulong**'s range is 0 to about 18 quintillion.

The last two integer types are a bit 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	65,535
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 using their 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 it 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). You will get a compiler error if you make a literal whose value is too big for everything. To illustrate how these bigger literal types work, consider this code:

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

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

```
ulong aVeryBigNumber = 100000000000U;  
aVeryBigNumber = 100000000000L;  
aVeryBigNumber = 100000000000UL;
```

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

When humans 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. If 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 only reach a few million, then a `uint` is safe. (You can always change a variable's type 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. 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 `ints` and runs the math as `ints`, forcing you to then go to the trouble of converting the result back to the smaller type. The `int` type is more convenient than `sbyte`, `byte`, `short`, and `ushort` if you are doing many math operations.

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 significant benefit.

Binary and Hexadecimal Literals



So far, the integer literals we have written have all been written using *base 10*, the normal 10-digit system humans typically use. 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 other 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 65,536 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 esoteric characters. The console window doesn't even 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 special number π ?

C# has three types that are called *floating-point data types*. These represent what mathematicians call *real numbers*, encompassing 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. The number of digits before and after the decimal point is locked in place with a fixed-point type. The decimal point may appear anywhere within the number with a floating-point type. C# does not have fixed-point types because they prevent you from efficiently using very large or very small numbers. 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 store the significant digits, affecting how precise you can be, while other bits define how much to scale it up or down, affecting the magnitudes you can represent. The more bits you use, the more of either you can do.

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, including in the computer's circuitry itself, **decimal** does not. That means **float** and **double** are faster. However, **decimal** uses most of its many bits for storing significant figures and is the most precise floating-point type. If you are doing something that needs extreme precision, even at the cost of speed, **decimal** is the better choice.

All floating-point numbers have ranges that are so mind-boggling in size that you wouldn't want to write them out the typical way. The math world often uses *scientific notation* to

compactly write extremely big or small numbers. A thorough discussion of scientific notation is beyond the scope of this book, but you can think of it as writing the zeroes in a number as a power of ten. Instead of 200, we could write 2×10^2 . Instead of 200000, we could write 2×10^5 . As the exponent grows by 1, the number of total digits also increases by 1. The exponent tells us the scale of the number.

The same technique can be used for very tiny numbers, though the exponent is negative. Instead of 0.02, we could write 2×10^{-2} . Instead of 0.00002, we could write 2×10^{-5} .

Now imagine what the numbers 2×10^{20} and 2×10^{-20} would look like when written the traditional way. With that image in your mind, let’s look at what ranges the 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 does not quite have 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 taking a moment to imagine 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	$\pm 1.0 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7	Yes
double	8	$\pm 5 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15-16	Yes
decimal	16	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	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 incredibly 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. (That number, by the way, is called Avogadro's Number—a number with special significance in chemistry.) The \times 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 last type we will cover in this level is the **bool** type. The **bool** type might seem strange 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 **ints**, with **false** being the number 0 and **true** being anything else. But C# delineates **ints** from **bools** 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 correct 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 any good at programming. 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 to 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, value, and expression 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 always need to write out a variable's type yourself. 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 it is declared. 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 facilitate type inference here, 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 types with longer 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** returns **strings**. It is much harder for us humans to pull this information out of memory.

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 usually avoid **var** in this book.

I recommend 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 AND THE PARSE METHODS

With 14 types at our disposal, we will sometimes need to convert between types. The easiest way 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. The `Convert` class 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:

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 some 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, 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`. `ushort`, `uint`, and `ulong` do the same, just with `UInt`.

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 `strings`. Many of our programs will need to convert the user's text to another type to work with 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 also learn additional parsing tools over time.

Parse **Methods**

Some C# programmers prefer an alternative to the `Convert` class. Many of these types have a `Parse` method to convert a string to the type. For example:

```
int number = int.Parse("9000");
```

Some people prefer this style over the mostly equivalent `Convert.ToInt32`. I'll generally use the `Convert` class in this book. But feel free to use this second approach if you prefer it.



Knowledge Check	Type System	25 XP
------------------------	--------------------	--------------

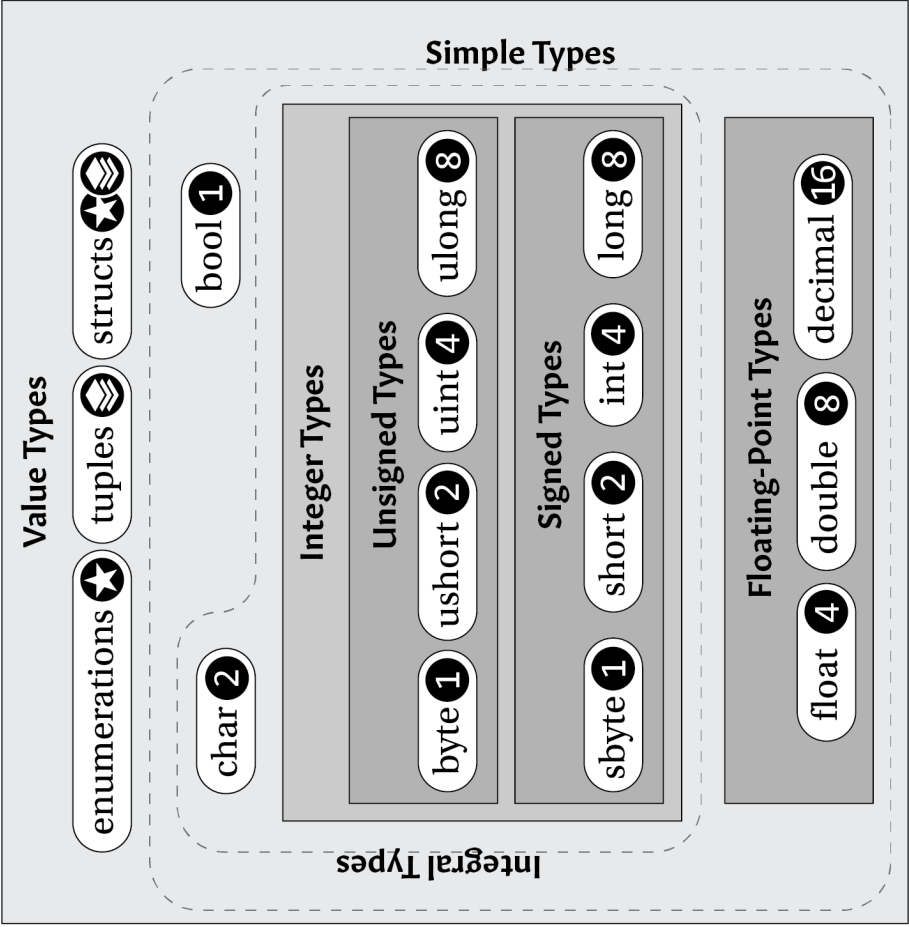
Check your knowledge with the following questions:

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`.

The following page contains a diagram that summarizes the C# type system. It includes everything we have discussed in this level and quite a few other types and categories we will discuss in the future.

C# Types



Integral Type: All integer types and char

Simple Type: A value type that supports literals constants.

Ranges

sbyte	-128	to	127
short	-32,768	to	32,767
int	-2,147,483,648	to	2,147,483,647
long	-9,223,372,036,854,775,808	to	9,223,372,036,854,775,807
byte	0	to	255
ushort	0	to	65,535
uint	0	to	4,294,967,295
ulong	0	to	18,446,744,073,709,551,615
float	$\pm 1.5 \times 10^{-45}$	to	$\pm 3.4 \times 10^{38}$
double	$\pm 5.0 \times 10^{-324}$	to	$\pm 1.7 \times 10^{308}$
decimal	$\pm 1.0 \times 10^{-28}$	to	$\pm 7.9228 \times 10^{28}$
			6 to 9 digits of precision
			15 to 17 digits of precision
			28 to 29 digits of precision

- 124816 Size (Bytes)
- Composed of other elements
- You can define your own

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.
 - The 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 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, and 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 start by defining a few terms. 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 an operation uses—the 2 and 3 here—are called *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 a 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;
```

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 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 it will all feel natural if you are familiar with the order of operations in math.

C# has many operators beyond addition, subtraction, multiplication, and division, so the complete ruleset is complicated. The book’s website has a table that shows the whole picture: csharpplayersguide.com/articles/operators-table. 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 the addition is done after, for a final result of `12`, which 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 is also 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 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 variable 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-Tag, a 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-Tag looks at you with fresh eyes. “Wait just a moment. You have the look of a Programmer about you. Can you help me write a program that will compute the 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 a 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.

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 uses only integers, and so it uses integer division. $5/2$ becomes 2 instead of 2.5, which 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 either way, it 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 divide by 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 you will want 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;
```

The – operator negates the value after it. Since a is 3, $-a$ will be -3 . If a had been -5 , $-a$ would evaluate to $+5$. It reverses the sign of a . Or you could think of it as multiplying it by -1 .

The unary + doesn’t do anything for the numeric types we have seen in this level, but it can sometimes add clarity to the code (in contrast to $-$). 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!), 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 useful initially, but it can be handy. 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? You can 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 to compute a variable's new value based on its current value. For 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, statements that update a variable based on its current value are common. There are even some shortcuts for it. Instead of **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 combines an operation (addition, in this case) with 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 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 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 users 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 the specifics are nuanced. These two operators can be written before or after the modified variable. For example, you can write either `x++` or `++x` to increment `x`. The first is called postfix notation, and the second is called prefix notation. There is no meaningful difference between the two when written as a

complete statement (`x++`; or `++x`;). 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 produce the new value of `x`), so `y` will have a value of **6** as well. The second part, in contrast, evaluates to `x`'s original value of **5**, which is assigned to `z`, even though `x` is incremented to **6**.

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 `ints` and two `doubles` 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* 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* does not risk 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*. A programmer must specifically ask for an explicit conversion to happen. An implicit conversion will occur automatically without the programmer stating it.

As a general rule, narrowing conversions, which risk losing data, are 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—an **int** to a **byte**—the conversion is not safe. To compile, we need to specifically state that we want to use the conversion, knowing the risks 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 convert it.”

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 **long**. Likewise, there is an implicit conversion from **byte** to **ushort**, **ushort** to **uint**, and **uint** to **ulong**. There is also an implicit conversion from all eight integer types to the floating-point types, but not the other way around.

However, casting conversions are not defined between every possible type. 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**. We can always fall back on **Convert** and do **int number = Convert.ToInt32(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.
```

Addition is not defined for the **short** type, but it 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 raises an important point: the casting operator has higher precedence than most other operators. 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 `ints`, 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 `doubles`. 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 is needed to assign the value to `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 ROUNDOFF ERROR

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 surpass this limit?

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 an operation causes a value to go beyond what its type can represent, 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 *roundoff error*. The number 10000 can be correctly represented with a `float`, 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;
```

The result is rounded to 10000, and **sum** will still be **10000** after the addition. Roundoff error is not usually a big deal, but occasionally, the lost digits accumulate, like when adding huge piles of tiny numbers. You can sometimes 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 scales.

THE MATH AND MATHF CLASSES

C# also includes two classes with the job of helping you do common math operations. These classes are called the **Math** class and the **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. **Pow**'s two pieces of information are the base and the power it is raised to. So **Math.Pow(x, 2)** 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 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** will contain 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. 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 **floats** instead of **doubles**. For example, **Math**'s **Pow** method expects **doubles** 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 beep sounds (`Beep`).
- Escape sequences start with a `\` and tell the computer to interpret the next 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}\"`.

In this level, we will 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 `Write`, does all the same stuff as `WriteLine`, without jumping to the following 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 helpful when assembling many small bits of text into a single line.

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 “Press any key to continue...”, 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. There are two versions of the **ReadKey** method (called “overloads,” 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 provides variables that store the colors it uses for displaying text. We’re not stuck with just black and white! This is best illustrated with an example:

```
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, which 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 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 change 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\net6.0\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. 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 to which notes.) The second piece of information is the duration, measured in milliseconds (1000 is a full second, 500 is 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 beginning, 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 following 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 valuable 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 placed 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 in 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 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 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 two 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. You could write code to do either, but there is no special syntax to do it automatically.

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");
```

To format something, after the expression, put a colon and then a format string. 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**.

You can use quite a few other symbols for format strings, but that is enough to give us a basic toolset to work with.



Challenge

The Defense of Consolas

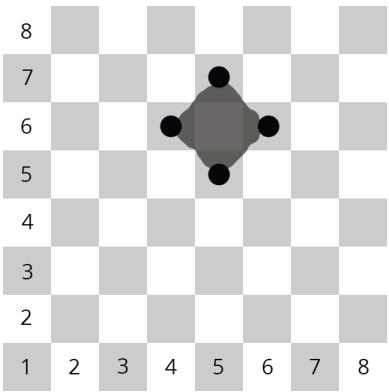
200 XP

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 (Row 7, 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.

**This is a preview. These pages have been
excluded from the preview.**

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. (Level 50.)

.NET Framework

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

.NET Multi-platform App UI

The evolution of Xamarin Forms and an upcoming cross-platform UI framework for mobile and desktop apps.

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 classes derived from it. Only abstract classes can 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 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. (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 automatically generates the backing field and basic get and set logic. (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*.

Binary

Composed of two things. Binary numbers use only 0's and 1's. (Level 3.)

Binary Code

The executable instructions that computers work with to do things. All programs are built out of binary code. (Levels 3 and 49.)

Binary Instructions

See *binary code*.

Binary Literal

A literal that specifies an integer in binary and is preceded by the marker `0b`: `0b00101001`. (Level 6.)

Binary Operator

An operator that works on two operands. Addition and subtraction are two examples. (Level 7.)

Bit

A single binary digit. A 0 or a 1. (Level 6.)

Bit Field

Compactly storing multiple related Boolean values, using only one bit per Boolean value. (Level 47.)

Bit Manipulation

Using specific operators to work with the individual bits of a data element. (Level 47.)

Bitwise Operator

One of several operators used for bit manipulation, including bitwise logical operators and bitshift operators. (Level 47.)

Block

A section of code demarcated with curly braces, typically containing many statements in sequence. (Level 9.)

Block Body

One of two styles of defining the body of a method or other member that uses a block. See also *expression body*. (Level 13.)

Boolean

Pertaining to truth values. A Boolean value can be either true or false. Used heavily in decision making and looping, and represented with the `bool` type in C#. (Level 6.)

Boxing

When a value type is removed from its regular place and placed elsewhere on the heap, accessible through a reference. (Level 28.)

Breakpoint

The marking of a location in code where the debugger should suspend execution so that you can inspect its state. (Bonus Level C.)

Built-In Type

One of a handful of types that the C# compiler knows a lot about and provides shortcuts to make working with them easy. These types have their own keywords, such as `int`, `string`, or `bool`. (Level 6.)

Byte

A block of eight bits. (Level 6.)

C++

A powerful all-purpose programming language. C++'s syntax inspired C#'s syntax. (Level 1.)

Call

See *method call*.

Callback

A method or chunk of code that is scheduled to happen after some other task completes. (Level 44.)

Casting

See *typecasting*.

Catch Block

A chunk of code intended to resolve an error produced by another part of the code. (Level 35.)

Character

A single letter or symbol. Represented by the `char` type. (Level 6.)

Checked Context

A section of code wherein mathematical overflow will throw an exception instead of wrapping around. An unchecked context is the default. (Level 47.)

CIL

See *Common Intermediate Language*.

Class

A category of types, formed by combining fields (data) and methods (operations on that data). The most versatile type you can define. Creates a blueprint used by instances of the type. All classes are reference types. See also *struct*, *type*, *record*, and *object*. (Level 18.)

**This is a preview. These pages have been
excluded from the preview.**

INDEX

Symbols

!= operator, 74
- operator, 51
-- operator, 57
 π , 61
! operator, 76, 177
#define, 387
#elif, 386
#else, 386
#endif, 386
#endregion, 385
#error, 385
#if, 386
#region, 385
#undef, 387
#warning, 385
& operator, 369, 382
&& operator, 76
&= operator, 383
* operator, 51, 368
.. operator, 93
/ operator, 51
?. operator, 176
?? operator, 177
?[] operator, 176
@ symbol, 66
[] operator, 90
^ operator, 93, 382
^= operator, 383
| operator, 382
|| operator, 76
|= operator, 383
~ operator, 382
~= operator, 383
+ operator, 51
< operator, 74
<< operator, 381
<= operator, 383

<= operator, 74
== operator, 70
=> operator, 81, 304
> operator, 74
-> operator, 369
>= operator, 74
>> operator, 381
>>= operator, 383
.cs file, 15
.csproj file, 15, 409
.dll, 456
.NET, 9, 10, 404, 452
.NET Core, 405, 452
.NET Framework, 404, 452
.NET MAUI, 407
.NET Multi-platform App User Interface, 407
.sln file, 409

0

0-based indexing, 91, 452

A

absolute value, 61
abstract class, 209, 452
abstract keyword, 210
abstract method, 209, 452
abstraction, 159, 452
accessibility level, 156, 452
accessibility modifier, 156
acquiring a lock, 349
Action (System), 294
add keyword, 301
addition, 51
Address Of operator, 369
ahead-of-time compilation, 403, 452
algorithm, 51
alias, 266
alignment, 66

allocating memory, 110
and keyword, 321
and pattern, 321
 Android, 10, 407
 anonymous type, 169, 453
 AOT compilation. *See* ahead-of-time compilation
 app model, 407, 453
 architecture, 402, 453
 argument, 102, 453
 arm, 453
 array, 90, 453
as keyword, 205
ascending keyword, 336
 ASP.NET, 407, 453
 assembler, 401, 453
 assembly, 401, 453
 assembly language, 453
 assignment, 453, 460, 464
 associative array, 453
async keyword, 355
 asynchronous programming, 351, 453
 attribute, 376, 453
 defining, 378
 auto property, 166
 auto-implemented property, 166
 automatic memory management, 122, 453
 auto-property, 453
await keyword, 355
 awaitable, 358, 453

B

backing field, 165, 453
 backing store, 165
 base class, 199, 453
 Base Class Library, 10, 22, 248, 403, 406, 453
base keyword, 203
 BCL. *See* Base Class Library
 binary, 400, 453, 454
 binary literal, 454
 binary operator, 51
BinaryReader (System.IO), 314
BinaryWriter (System.IO), 314
 binding, 20
 bit, 38, 400, 454
 bit field, 380, 454
 bit manipulation, 380, 454
 bitshift operator, 381
 bitwise operator, 454
 Blazor, 407
 block, 454
 block body, 105, 454
 block scope, 72
 block statement, 70
 body. *See* method body
bool, 45
 Boolean, 454
Boolean (System), 225
 boxing, 226, 454
 boxing conversion, 226
break keyword, 87
 breakpoint, 449, 454
 conditions and actions, 451
 build configuration, 27, 409
 built-in type, 38, 454
 built-in type alias, 225
by keyword, 339
 byte, 38, 40, 400, 454

Byte (System), 225

C

C, 10
 C#, **9**
 C++, 10, 454
 callback, 352, 454
 camelCase, 37
 captured variable, 306
 case guard, 319
case keyword, 81
 casting, 59, 454
 catch block, 454
 catching exceptions, 281
char, 42
Char (System), 225
 character, 454
 checked context, 391, 454
checked keyword, 391
 child class, 199
 CIL, 402, 405
 clamp, 62
 class, 21, 130, 144, 145, 454
 compared to structs, 221
 creating instances, 147
 default field values, 149
 defining, 145
 defining constructors, 148
 sealing, 205
class keyword, 145
 class library, 406
 clause (query expressions), 334
 ClickOnce, 411
 closure, 306, 455
 CLR. *See* Common Language Runtime
 Code Editor window, 18
 code library, 394
 code map, 20
 Code Window, 436, 455
 collaborator, 181
 collection initializer syntax, 93, 455
 command-line arguments, 387, 455
 comment, 29, 455
 Common Intermediate Language, 402, 405, 455
 Common Language Runtime, 402, 405, 455
 compiler, 18, 401, 455
 compiler error, 27, 442, 455
 suggestions for fixing, 443
 compiler warning, 442, 455
 compile-time constant, 272
 compiling, 18, 399
 composite type, 138, 455
 composition, 138
 compound assignment operator, 57, 455
 concrete class, 210, 455
 concurrency, 343, 455
 concurrency issue, 347
 condition, 70
 conditional compilation symbol, 386, 455
 conditional operator, 77
 const, 375
const keyword, 375
 constant, 375, 455
 constant pattern, 317
 constructor, 147, 148, 455
 default parameterless constructor, 455
 parameterless, 150
 context switch, 344, 455

continuation clause, 339
continue keyword, 87
 contravariant, 390
Convert (System), 47
 cosine, 62
 covariance, 390
 CRC card, 455
 CRC cards, 181
 critical section, 348, 456
 curly braces, 456
 custom conversion, 329, 456

D

dangling pointer, 456
 dangling reference, 122
 data structure, 456
DateTime (System), 250
 deadlock, 349, 456
 deallocating memory, 110
 debug, 456
 debugger, 447, 456
 debugging, 27, 447
decimal, 43
Decimal (System), 225
 declaration, 98, 456
 declaration pattern, 318
 deconstruction, 141, 229, 456
 deconstructor, 276
 decrement, 456
 decrement operator, 57
default keyword, 81, 240
 default operator, 240
 deferred execution, 340, 456
 delegate, 291, 456
 delegate chain, 295
delegate keyword, 292
 dependency, 456
 dependency (project), 394
 derived class, 199, 456
 deriving from classes, 199
descending keyword, 336
 deserialization, 456
 design, 144, 178, 456
 desktop development, 407
 dictionary, 456
Dictionary<TKey, TValue> (System.Collections.Generic), 257
 digit separator, 41, 456
 directed graph, 117
Directory (System.IO), 312
 discard, 142, 456
 discard pattern, 317
 Discord, 5
 divide and conquer, 456
 division, 51
 division by zero, 55, 456
DLLImport (System.Runtime.InteropServices), 372
do/while loop, 86
 dot operator, 20
dotnet command-line interface, 13, 412
double, 43
Double (System), 225
 downcasting, 204
dynamic keyword, 362
 dynamic object, 361, 457
 dynamic objects, 361
 dynamic type checking, 361, 457
DynamicObject (System.Dynamic), 364

E

E notation, 457
 early exit, 104, 457
 Edit and Continue, 451
else if statement, 73
else statement, 73
 encapsulation, 146, 457
 entry point, 23, 268, 457
 enum. *See* enumeration
enum keyword, 133
 enumeration, 132, 457
 equality operator, 70
equals keyword, 338
Equals method, 200
 Error List, 440, 457
 escape sequence, 65
 evaluation, 457
 event, 296, 457
 custom accessors, 301
 leak, 300
 null, 299
 raising, 297
 subscribing, 298
event keyword, 297
 event leak, 457
EventHandler (System), 300
EventHandler<EventArgs> (System), 300
 exception, 280, 457
 guidelines for using, 285
 rethrow, 288
Exception (System), 281
 exception filter, 289
 exception handler, 281
 EXE, 457
ExpandoObject (System.Dynamic), 363
 explicit, 457
 explicit conversion, 58
explicit keyword, 329
 exponent, 61
 expression, 24, 457
 evaluating, 24
 expression body, 105, 457
 extending classes, 199
 extension method, 457
extern keyword, 371

F

F#, 10, 402, 405
 factory method, 172
false keyword, 45
 field, 145, 457
 default value, 149
 initialization, 150
File (System.IO), 308
 files, 308
FileStream (System.IO), 314
 finally block, 284
finally keyword, 284
 fire (event), 297
fixed statement, 369, 457
 fixed-size array, 370, 457
 fixed-size buffer, 370
 flags enumeration, 383
float, 43
 floating-point division, 54, 458
 floating-point type, 43, 458

for loop, 86
 frame. *See* stack frame
 framework-dependent deployment, 412, 458
from clause, 334
from keyword, 334
 fully qualified name, 264, 265, 458
Func (System), 294
 function, 98, 458

G

game development, 408
 garbage collection, 122, 406, 458
 garbage collector, 123
 generic method, 238
 generic type, 233, 236, 458
 inheritance, 237
 generic type argument, 236, 458
 generic type constraint, 458
 generic type constraints, 238
 generic type parameter, 236, 458
 multiple, 237
 generic variance, 389, 458
 generics, 233
 constraints, 238
 inheritance, 389
 motivation for, 233
get keyword, 164
GetHashCode method, 258
 get-only property, 167
 getter, 157, 164, 458
GetType method, 204
global keyword, 266
 global namespace, 264, 458
 global state, 171, 458
 global using directive, 266
goto keyword, 388
 graph, 117
group by clause, 339
 guard expression, 319
Guid (System), 252

H

hash code, 258, 458
 heap, 115, 458
 hexadecimal, 458
 hexadecimal literal, 42

I

IAsyncEnumerable<T> (System.Collections.Generic), 359, 375
 IDE. *See* integrated development environment
 identifier, 20
IDisposable (System), 384
IDynamicMetaObjectProvider (System.Dynamic), 363
IDynamicMetaObjectProvider interface, 362
IEnumerable<T> (System.Collections.Generic), 256, 334
 if statement, 69
 IL, 402
 immutability, 167, 459
 implicit, 459
 implicit conversion, 58
implicit keyword, 329
in keyword, 276

increment, 459
 increment operator, 57
 index, 91, 459
 index initializer syntax, 328
 indexer, 327, 459
 indexer operator, 91
 indirection operator, 369
 infinite loop, 85, 459
 infinity, 54
 information hiding, 155
 inheritance, 198, 459
 constructors, 202
 inheritance hierarchy, 202
 inheritance relationship, 199
init keyword, 168
 initialization, 459
 inner exception, 289
 input parameter, 276
 instance, 130, 145, 459
 instance variable. *See* field
 instruction set architecture, 401, 459
int type, 34
Int16 (System), 225
Int32 (System), 225
Int64 (System), 225
 integer, 34
 integer division, 54, 459
 integer type, 39
 integral type, 39, 459
 integrated development environment, 11, 459
 IntelliSense, 437, 459
 interface, 212, 459
 and base classes, 215
 default methods, 216
 defining, 213
 explicit implementation, 215
 implementing, 214
interface keyword, 213
internal keyword, 160
into clause, 339
into keyword, 339
 iOS, 10
is keyword, 205, 322
 ISA, 402
 iterator, 374, 460

J

jagged array, 96, 460
 Java, 10, 460
 JetBrains Rider, 12
 JIT compiler, 403
 jitter, 403
join clause, 338
join keyword, 338
 Just-in-Time compilation, 406
 Just-in-Time compiler, 403, 460

K

keyword, 26, 460

L

label, 388
 labeled statement, 388
 lambda expression, 303, 460

lambda statement, 305
 Language Integrated Query, 333, 460
 lazy evaluation, 460
let clause, 339
let keyword, 339
 library, 315, 394, 406
 LINQ, 333
 LINQ to SQL, 341
 Linux, 10
List<T> (**System.Collections.Generic**), 253
 listener, 297
 literal, 20
 literal value, 460
 local function, 98, 307, 460
 local variable, 101, 460
lock keyword, 348
 logical operator, 76, 460
long, 40
 loop, 84, 460
 lowerCamelCase, 37

M

macOS, 10
 main method, 23, 457, 460
Main method, 23, 268
 managed code, 460
 managed memory, 460
 math, 50
Math (**System**), 61
MathF (**System**), 62
 MAUI, 407
 maximum, 62
 member, 20, 460
 member access operator, 20
 memory address, 32, 461
 memory allocation, 461
 memory leak, 122, 461
 memory management, 109
 memory safety, 406, 461
 method, 21, 97, 98, 271, 461

- calling, 99
- calling methods, 21
- return type, 98
- returning data, 21
- scope, 100

 method body, 98
 method call, 21, 461
 method call syntax, 336, 461
 method group, 105, 461
 method implementation, 461
 method invocation, 21
 method overload, 104, 461
 method scope, 72
 method signature, 461
 Microsoft Developer Network, 431
 minimum, 62
 mobile development, 407
 Mono, 404, 461
 MonoGame, 408
 MSBuild, 409
 MSIL, 402
MulticastDelegate (**System**), 295
 multi-dimensional array, 95, 461
 multiplication, 51
 multi-threading, 343, 461
 mutex, 348
 mutual exclusion, 348, 461
 MVC, 407

N

name, 20
 name binding, 20
 name collision, 461
 name conflict, 266
 name hiding, 151, 152, 461
 named argument, 272, 461
nameof operator, 379
 namespace, 21, 264, 446, 461
namespace keyword, 267
 namespaces, 267
 NaN, 54, 462
 narrowing conversion, 58, 462
 native code, 367, 462
 native integer types, 371
 nested pattern, 320
 nested type, 379
 nesting, 77, 88, 462
new keyword, 210
 new method, 210
nint, 371
not keyword, 321
not pattern, 321
 noun extraction, 180, 462
 NuGet, 396
 NuGet Package Manager, 462
nuint, 371
 null, 92
 null check, 176
 null conditional operator, 176
null keyword, 174
 null reference, 174, 462
 nullable type, 462
Nullable<T> (**System**), 259
 null-coalescing operator, 177

O

object, 130, 144, 199, 462
Object (**System**), 199, 225
 object initializer syntax, 168
object keyword, 199
 object-initializer syntax, 462
 object-oriented design, 144, 153, 178, 462

- rules, 184

 object-oriented programming, 129, 462
 observer, 297
ObsoleteAttribute (**System**), 376
on keyword, 338
 operation, 50, 462
 operator, 50, 462

- binary, 454
- ternary, 466
- unary, 466

 operator associativity, 52, 462
operator keyword, 326
 operator overloading, 325, 462
 operator precedence, 52, 462
 optional arguments, 271
 optional parameter, 271, 462
or keyword, 321
or pattern, 321
 order of operations, 52, 462
orderby clause, 336
orderby keyword, 336
out keyword, 275, 390
 out-of-order execution, 462

output parameter, 275
 overflow, 60, 462
 overload. *See* method overload
 overload resolution, 105, 463
 overloading, 463

P

P/Invoke, 371
 package, 396, 463
 package manager, 396
 parameter, 101, 149, 463
 variable number of, 272
 parameterful property, 327
 parameterless constructor, 150
params keyword, 272
 parent class. *See* base class
 parentheses, 463
 parse, 463
Parse methods, 48
 parsing, 48
 partial class, 387, 463
partial keyword, 387
 partial method, 388
 PascalCase, 37
 passing, 102
 passing by reference, 273, 463
 passing by value, 273
Path (System.IO), 313
 pattern matching, 82, 316, 463
 pi, 61
 pinning, 369
 Platform Invocation Services, 371, 463
 pointer member access operator, 369
 pointer type, 368, 463
 polymorphism, 207, 463
 positional pattern, 321
 postfix notation, 57
 power (math), 61
 PowerShell, 10
Predicate (System), 294
 prefix notation, 57
 preprocessor directive, 385, 463
 primitive type. *See* built-in type
 print debugging, 448, 463
private keyword, 156
private protected accessibility level, 380
Program class, 23
 program order, 464
 programming language, 9, 401
 project, 464
 project configuration, 15
 project template, 15
 Properties Window, 440
 property, 163, 464
 property pattern, 319
protected accessibility modifier, 205
protected internal accessibility level, 380
protected keyword, 205
 pseudo-random number generation, 249
public keyword, 156
 publish profile, 410
 publishing, 409

Q

query expression, 333, 464
 query syntax, 464

Quick Action, 438

R

raise (event), 297
Random (System), 249
 range operator, 93
 range variable, 335
 Razor Pages, 407
readonly keyword, 167
 read-only property, 167
 record, 228, 464
 struct-based, 231
 rectangular array, 96, 464
 recursion, 107, 464, *See* recursion
ref keyword, 274
ref local variable, 276
ref return, 276
 refactor, 464
 refactoring, 189
 reference, 116, 464
 reference semantics, 121, 464
 reference type, 118, 464
 reflection, 378, 464
 relational operator, 74, 464
 remainder, 55
remove keyword, 301
 requirements, 179, 464
 responsibility, 181
 rethrowing exceptions, 288
 return, 21, 26, 103, 464
return keyword, 103
 return type, 464
 returning early, 104
 Rider. *See* JetBrains Rider
 roundoff error, 60, 464
 runtime, 10, 402, 464

S

sbyte, 40
SByte (System), 225
 scheduler, 344, 465
 scope, 72, 100, 465
 SDK. *See* Software Development Kit
 sealed class, 205, 465
sealed keyword, 205
 seed, 249
select clause, 334
select keyword, 334
 self-contained deployment, 412
 serialization, 310
set keyword, 164
 setter, 157, 164
short, 40
 SignalR, 407
 signed type, 40, 465
 sine, 62
Single (System), 225
sizeof operator, 370
 software design, 144, 178
 Software Development Kit, 10, 406
 solution, 465
 Solution Explorer, 18, 439, 465
 source code, 15, 465
Span<T> (System), 276
 square brackets, 465

square root, 61
 stack, 110, 465
 stack allocation, 370, 465
 stack frame, 111, 465
 stack trace, 288, 465
stackalloc keyword, 370
 standard library, 406, 465
 statement, 23, 465
 static, 170, 465
 static class, 173
 static constructor, 172
 static field, 170
static keyword, 170
 static method, 172
 static property, 171
 static type checking, 361, 465
 static using directive, 266
 stream, 313
Stream (System.IO), 314
StreamWriter (System.IO), 314
string, 42, 465
String (System), 225
 string formatting, 67
 string interpolation, 66
 string manipulation, 310
string type, 20
 string literal, 20
StringBuilder (System.Text), 260
 struct, 219, 465
 compared to classes, 221
 constructors, 220
 memory, 220
struct keyword, 219
 subclass, 199
 subtraction, 51
 superclass, 199
 switch, 79, 466
 switch arm, 79
 switch expression, 81
 guard, 319
switch keyword, 80
 switch statement, 80
 symbol, 386
 synchronization context, 357
 synchronization issue, 347
 synchronous programming, 351, 466
 syntax, 19
System namespace, 21

T

tangent, 62
 task, 353, 466
Task (System.Threading.Tasks), 353
Task<T> (System.Threading.Tasks), 353
 ternary operator, 51, 77
this keyword, 152
 thread, 343, 466
Thread (System.Threading), 344
 thread pool, 356, 466
 thread safety, 347, 348, 466
Thread.Sleep, 346
 threading, 343
 threading issue, 347
ThreadPool (System.Threading), 356
throw keyword, 283
 throwing exceptions, 281
TimeSpan (System), 251
 top-level statement, 268, 466

ToString method, 200
 trigonometric functions, 62
true keyword, 45
try keyword, 281
 tuple, 137
 deconstruction, 141
 element names, 139
 equality, 142
 in parameters and return types, 139
 type, 130, 466
Type (System), 378
 type inference, 46, 466
 type pattern, 318
 type safety, 406, 466
 typecasting, 466
typeof keyword, 204

U

uint, 40
UInt16 (System), 225
UInt32 (System), 225
UInt64 (System), 225
ulong, 40
 UML, 180
 unary operator, 51
 unboxing, 226, 466
 unboxing conversion, 226
 unchecked context, 466
unchecked keyword, 391
 underlying type, 136, 466
 Unicode, 42
 Unified Modeling Language, 180
 Unity game engine, 408
 Universal Windows Platform, 408, 466
 unmanaged code, 367, 466
 unmanaged type, 368
 unpacking, 141, 466
 unsafe code, 367, 466
 unsafe context, 368, 466
unsafe keyword, 368
 unsigned type, 40, 466
 unverifiable code, 368
 UpperCamelCase, 37
 user-defined conversion, 467
ushort, 40
using directive, 22, 265, 467
using statement, 384, 467
 UWP, 408

V

value keyword, 164
 value semantics, 121, 229, 467
 value type, 118, 467
ValueTask (System.Threading.Tasks), 359
ValueTask<T> (System.Threading.Tasks), 359
var, 46
var pattern, 322
 variable, 25, 32, 467
 assignment, 33
 declaration, 25, 33
 initialization, 33
 naming, 36
 variance, 390
 verbatim string literal, 66
 virtual machine, 402, 467
 virtual method, 467

Visual Basic, 10, 402, 405, 467
Visual Studio, 12, 18, 435, 467
 Community Edition, 11
 Enterprise Edition, 12
 installing, 13
 Professional Edition, 12
Visual Studio Code, 12, 467
Visual Studio for Mac, 12
volatile field, 392, 467
volatile keyword, 392

W

Web API, 407
web development, 407
where clause, 335
where keyword, 335
while keyword, 84
while loop, 84

whitespace, 23
widening conversion, 58
Windows, 10
Windows Forms, 407, 467
Windows Presentation Foundation, 407, 467
WinForms, 407
with keyword, 229
WPF, 407

X

Xamarin Forms, 407, 467
XML Documentation Comment, 106, 467

Y

yield keyword, 374

THE C# PLAYER'S GUIDE

IT'S DANGEROUS TO CODE ALONE!
TAKE THIS.



The book in your hands is a [different kind of programming book](#). Like an entertaining video game, programming is an often challenging but always rewarding experience. This book shakes off the dusty, dull, dryness of the typical programming book, replacing it with something more exciting and flavorful: a bit of humor, a casual tone, and examples involving dragons and asteroids instead of bank accounts and employees.

And since you learn to program by doing instead of just reading, this book contains [over 100 hands-on programming challenges](#). You will be building software instead of just reading about it. By completing the challenges, you'll earn experience points, level up, and become a True C# Programmer!

This book covers the C# language from the ground up. It doesn't assume you've been programming for years, but it also doesn't hold back on exciting, powerful language features.

- The journey begins by getting you set up to program in C#.
- We will then explore the [basic mechanics of C#](#): statements, expressions, variables, if statements, loops, and methods.
- Next, we dive deep into a powerful and central feature of C#: [object-oriented programming](#), which is an essential tool needed to build larger programs.
- We then look at the [advanced C# features](#) that make the language unique, elegant, and powerful.

With this book as your companion, you will soon be off to save the world (or take it over) with your own C# programs!