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

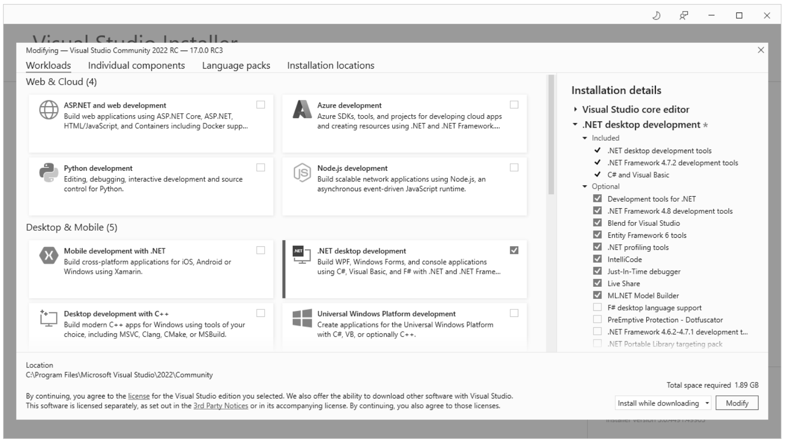
This book's focus is the C# language itself, but when I need to illustrate a task in an IDE, this book uses Visual Studio Community Edition. The Professional and Enterprise Editions should be identical. Other IDEs are usually similar, but you will find differences.

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:



Modifying — Visual Studio Community 2022 RC — 17.0.0 RC3

Workloads | Individual components | Language packs | Installation locations

Web & Cloud (4)

- ☐ **ASP.NET and web development** — Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...
- ☐ **Azure development** — Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET and .NET Framework....
- ☐ **Python development** — Editing, debugging, interactive development and source control for Python.
- ☐ **Node.js development** — Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Desktop & Mobile (5)

- ☐ **Mobile development with .NET** — Build cross-platform applications for iOS, Android or Windows using Xamarin.
- ☑ **.NET desktop development** — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...
- ☐ **Desktop development with C++** — Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.
- ☐ **Universal Windows Platform development** — Create applications for the Universal Windows Platform with C#, VB, or optionally C++.

Installation details

- Visual Studio core editor
- .NET desktop development *
  - Included
    - ✓ .NET desktop development tools
    - ✓ .NET Framework 4.7.2 development tools
    - ✓ C# and Visual Basic
  - Optional
    - ☑ Development tools for .NET
    - ☑ .NET Framework 4.8 development tools
    - ☑ Blend for Visual Studio
    - ☑ Entity Framework 6 tools
    - ☑ .NET profiling tools
    - ☑ IntelliCode
    - ☑ Just-In-Time debugger
    - ☑ Live Share
    - ☑ ML.NET Model Builder
    - ☐ F# desktop language support
    - ☐ PreEmptive Protection - Dotfuscator
    - ☐ .NET Framework 4.6.2-4.7.1 development t...
    - ☐ .NET Portable Library targeting pack

Location
C:\Program Files\Microsoft Visual Studio\2022\Community

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Total space required 1.89 GB

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.

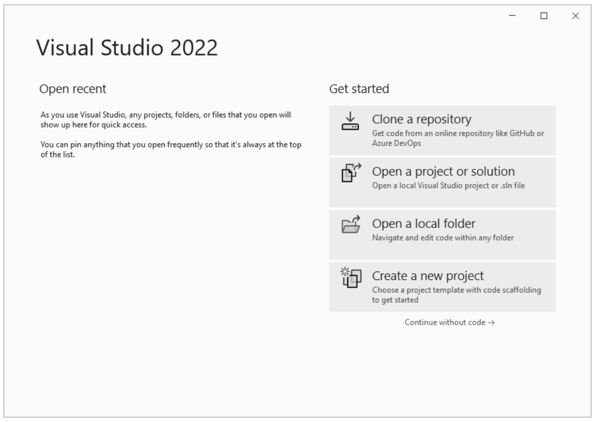
Warning! Be sure you get the right workload installed. If you don't, you won't be able to use all of the C# features described in this book.

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:


Visual Studio 2022
Open recent
Get started
Clone a repository
Get code from an online repository like GitHub or Azure DevOps
Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Continue without code →

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

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!

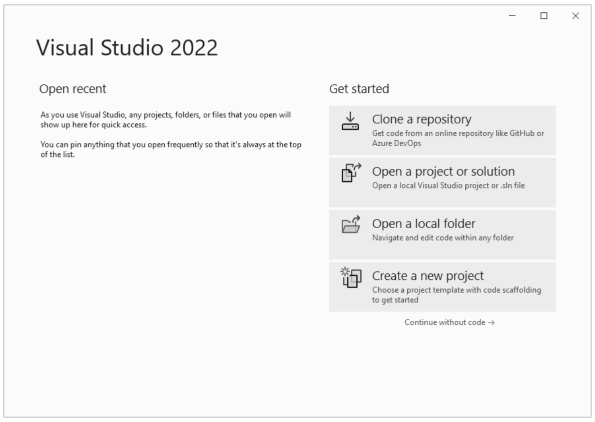
CREATING A NEW PROJECT

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:



As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

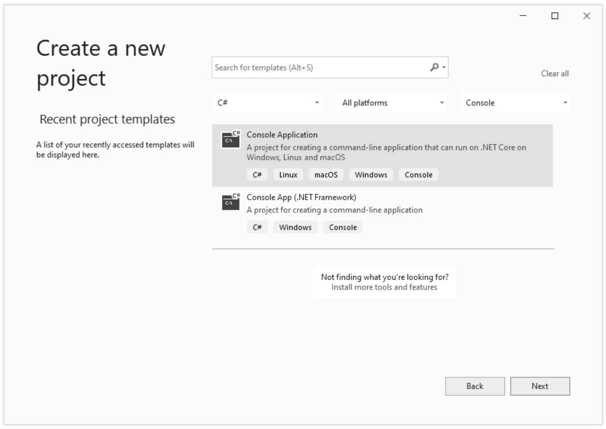
Get code from an online repository like GitHub or Azure DevOps

Open a local Visual Studio project or .sln file

Navigate and edit code within any folder

Choose a project template with code scaffolding to get started

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



A list of your recently accessed templates will be displayed here.

A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS

A project for creating a command-line application

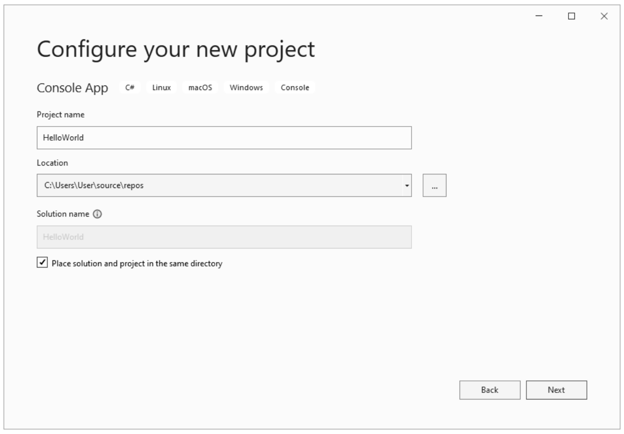
Not finding what you're looking for?
Install more tools and features

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

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

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

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



Configure your new project

Console App C# Linux macOS Windows Console

Project name: HelloWorld

Location: C:\Users\User\source\repos

Solution name: HelloWorld

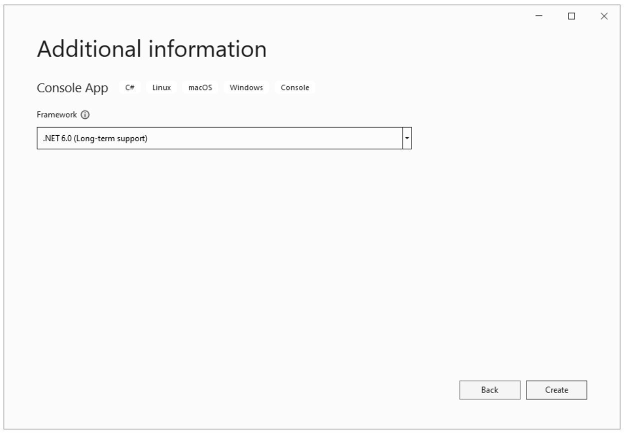
☑ Place solution and project in the same directory

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:



Additional information

Console App C# Linux macOS Windows Console

Framework: .NET 6.0 (Long-term support)

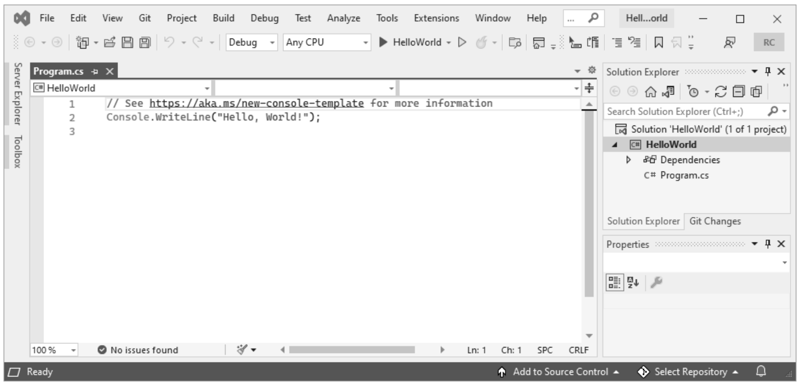
Make sure you pick **.NET 6.0** for this book! We will be using many .NET 6 features. You can change it after creation, but it is much easier to get it right in the first place.

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

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

A Brief Tour of Visual Studio

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



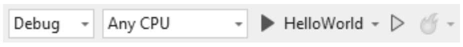
```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

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

Right now, there are three things you need to know to get started. First, the big text editor on the left side is the Code Window or the Code Editor. You will spend most of your time working here.

Second, on the right side is the Solution Explorer. That shows you a high-level view of your code and the configuration needed to turn it into working code. You will spend only a little time here initially, but you will use this more as you begin to make larger programs.

Third, we will run our programs using the part of the Standard Toolbar shown below:



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

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

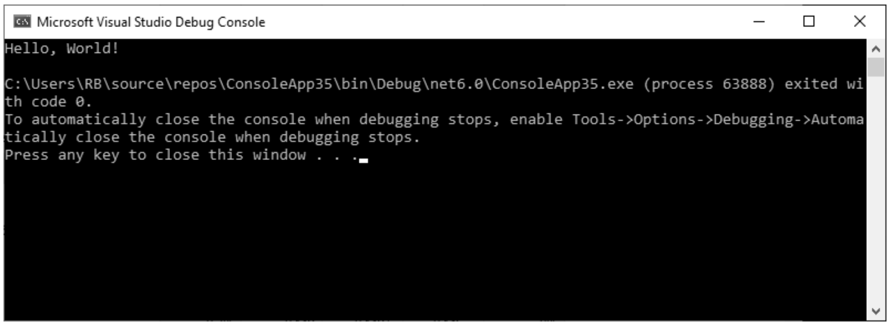
Compiling and Running Your Program

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

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

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

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



```
Hello, World!

C:\Users\RB\source\repos\ConsoleApp35\bin\Debug\net6.0\ConsoleApp35.exe (process 63888) exited wi
th code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automa
tically close the console when debugging stops.
Press any key to close this window . . .
```

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.

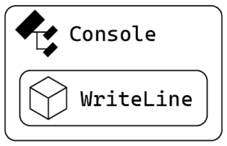
Identifiers

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.

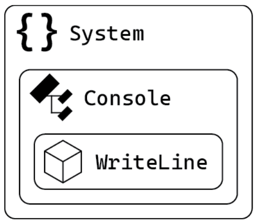
Classes are primarily composed of two things: (1) the data they need to do their job and (2) tasks they can perform. These tasks come in the form of methods, and **`WriteLine`** is an example. A method is a named, reusable block of code that you can request to run. **`WriteLine`**'s task is to take text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

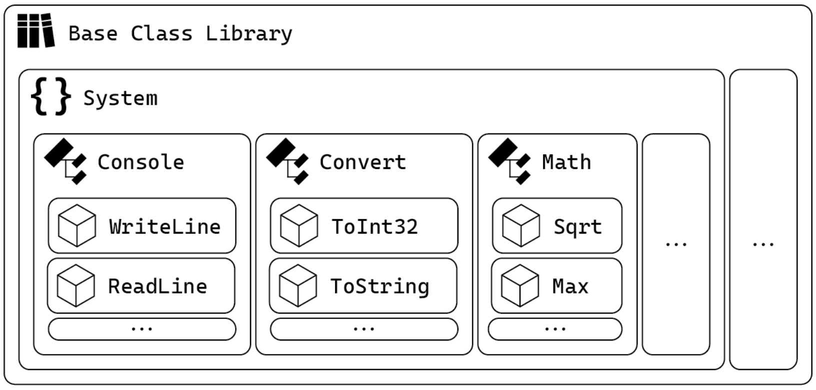
```
using System;
```

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:

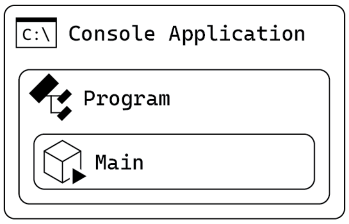


It is huge! If we drew the complete diagram, it might be longer than this whole book!

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:

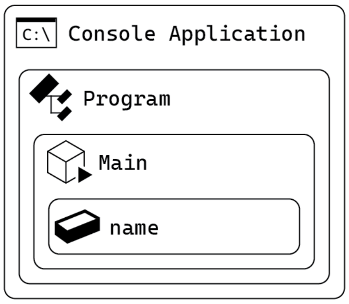
```
string name;
name = "User";
Console.WriteLine("Hi " + name);
```

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.

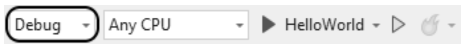
Debugging

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.

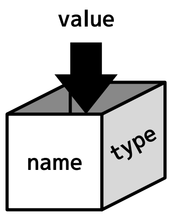
In this level, we will look at variables in more depth. We will also look at some rules around good variable names.

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:

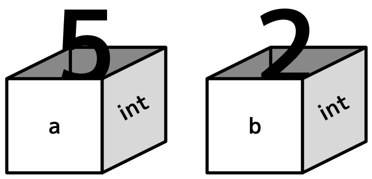
```
score = 4;
score = 11;
score = -1564;
```

Reading from a Variable Does Not Change It

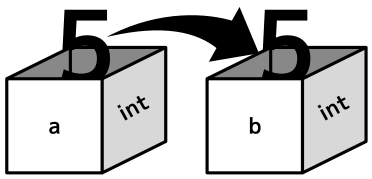
When you read the contents of a variable, the variable's contents are copied out. To illustrate:

```
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 π?

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

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

The "floating" part comes because it contrasts with fixed-point types. The number of digits before and after the decimal point is locked in place with a fixed-point type. The decimal point may appear anywhere within the number with a floating-point type. C# does not have fixed-point types because they prevent you from efficiently using very large or very small numbers. In contrast, floating-point numbers let you represent a specific number of significant digits and scale them to be big or small. For example, they allow you to express the numbers 1,250,421,012.6 and 0.0000000000012504210126 equally well, which is something a fixed-point representation cannot reasonably do.

With floating-point types, some of the bits store the significant digits, affecting how precise you can be, while other bits define how much to scale it up or down, affecting the magnitudes you can represent. The more bits you use, the more of either you can do.

There are three flavors of floating-point numbers: **`float`**, **`double`**, and **`decimal`**. The **`float`** type uses 4 bytes, while **`double`** uses twice that many (hence the "double") at 8 bytes. The **`decimal`** type uses 16 bytes. While **`float`** and **`double`** follow conventions used across the computing world, including in the computer's circuitry itself, **`decimal`** does not. That means **`float`** and **`double`** are faster. However, **`decimal`** uses most of its many bits for storing significant figures and is the most precise floating-point type. If you are doing something that needs extreme precision, even at the cost of speed, **`decimal`** is the better choice.

All floating-point numbers have ranges that are so mind-boggling in size that you wouldn't want to write them out the typical way. The math world often uses *scientific notation* to compactly write extremely big or small numbers. A thorough discussion of scientific notation is beyond the scope of this book, but you can think of it as writing the zeroes in a number as a power of ten. Instead of 200, we could write 2×10^2. Instead of 200000, we could write 2×10^5. As the exponent grows by 1, the number of total digits also increases by 1. The exponent tells us the scale of the number.

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

Now imagine what the numbers 2×10^{20} and 2×10^{-20} would look like when written the traditional way. With that image in your mind, let's look at what ranges the floating-point types can represent.

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

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

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

I'm not going to write out all of those numbers in normal notation, but it is worth taking a moment to imagine what they might look like.

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

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

Type	Bytes	Range	Digits of Precision	Hardware Supported
`float`	4	$\pm1.0\times10^{-45}$ to $\pm3.4\times10^{38}$	7	Yes
`double`	8	$\pm5\times10^{-324}$ to $\pm1.7\times10^{308}$	15-16	Yes
`decimal`	16	$\pm1.0\times10^{-28}$ to $\pm7.9\times10^{28}$	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

SIDE QUEST

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

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

THE BOOL TYPE

The 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.

Knowledge Check **Type System** **25 XP**

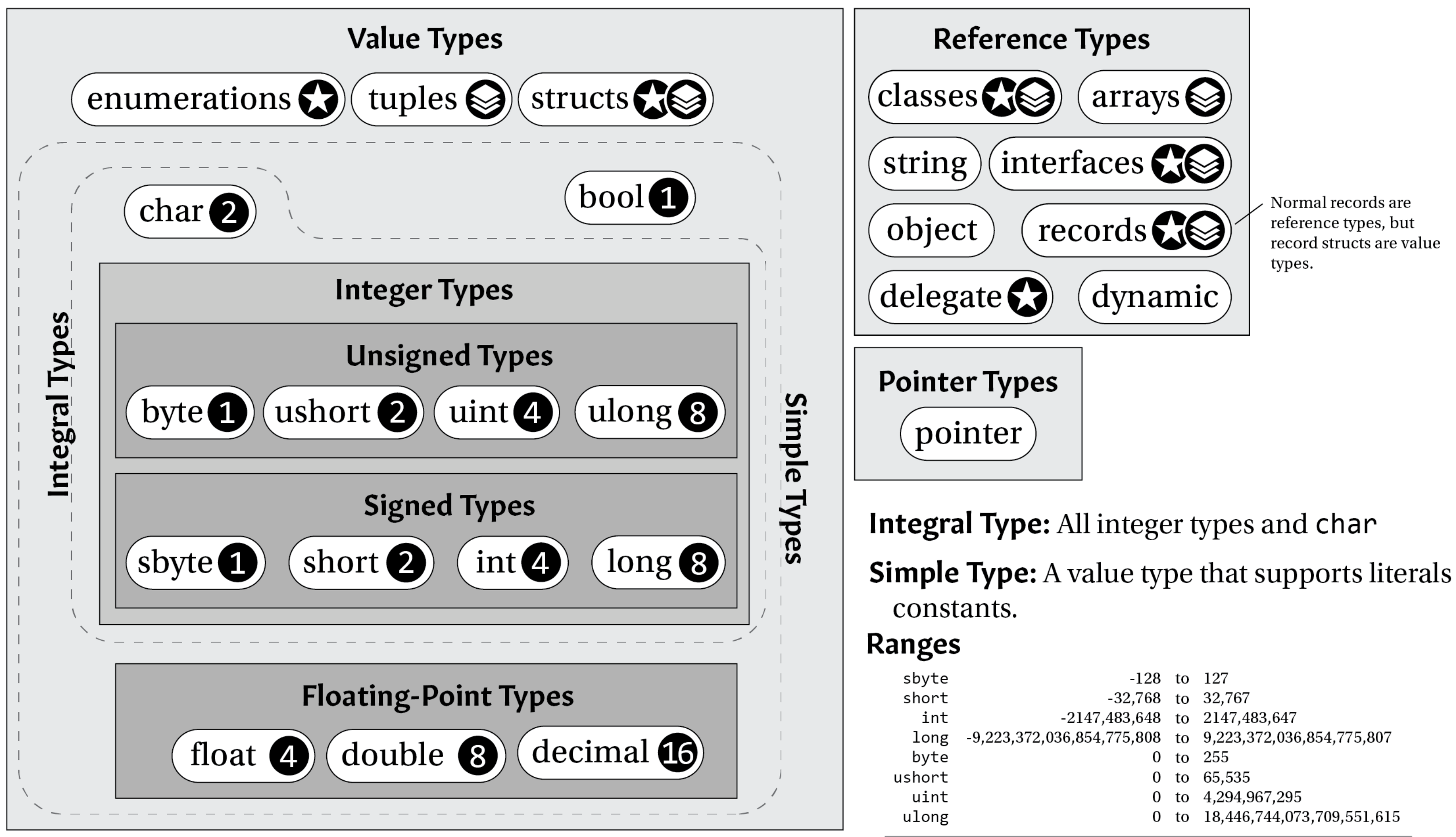
Check your knowledge with the following questions:

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

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

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

C# Types



Integral Type: All integer types and `char`

Simple Type: A value type that supports literals constants.

Ranges

`sbyte`	-128	to	127
`short`	-32,768	to	32,767
`int`	-2147,483,648	to	2147,483,647
`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

❶❷❹❽⓰ **Size (Bytes)**

Composed of other elements

★ **You can define your own**

Level 7

Math

Speedrun

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

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

Operations and Operators

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

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

Addition, Subtraction, Multiplication, and Division

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

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

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

The same thing works for subtraction:

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

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

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

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

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

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

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

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

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

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

```
double moneyMadeFromGame = 100000;
double totalProgrammers = 4;
double moneyPerPerson = moneyMadeFromGame / totalProgrammers;
```

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

Compound Expressions and Order of Operations

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

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

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

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

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

C# has many operators beyond addition, subtraction, multiplication, and division, so the complete ruleset is complicated. The book's website has a table that shows the whole picture: **csharpplayersguide.com/articles/operators-table**. For now, it is enough to say that the following two rules apply:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Challenge The Triangle Farmer 100 XP

As you pass through the fields near Arithmetica City, you encounter P-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`**.

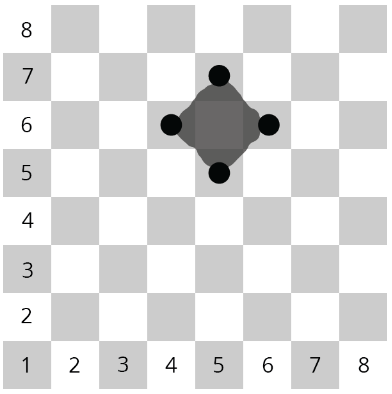
You can use quite a few other symbols for format strings, but that is enough to give us a basic toolset to work with.

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.

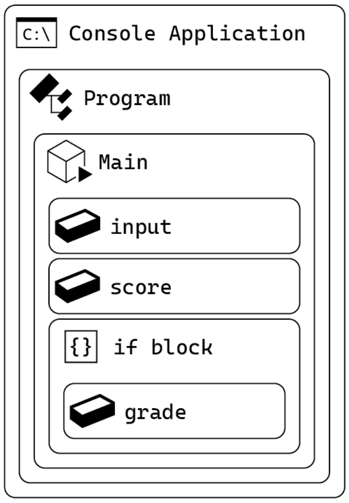
Blocks, Variables, and Scope

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:



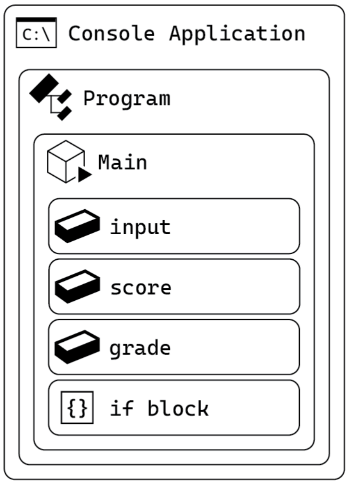
The **input** and **score** variables live directly in our main method, but the **grade** variable lives in the **if** block. We can use **grade** within the **if** block. And, importantly, we can reach outward and use **input** and **score** as well. But for code in our main method outside the **if** 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.

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!"

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

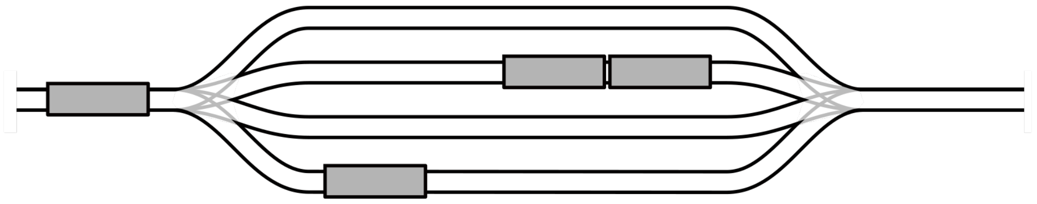
LEVEL 10

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.

LEVEL 11

LOOPING

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]`**.

❗ **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.

```
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:

```
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.

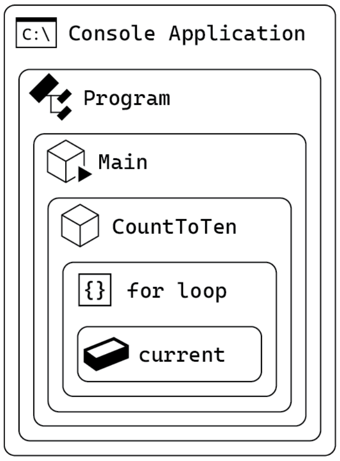
The **`void`** part, before the name, is the method's *return type*. We'll deal with this in more depth later. For now, all we need to know is that **`void`** means the method does not produce a result.

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.

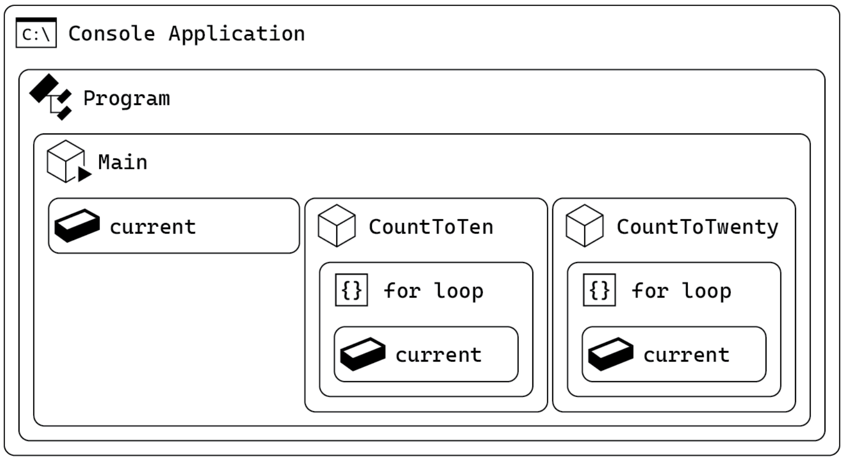
Methods Get Their Own Variables

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.

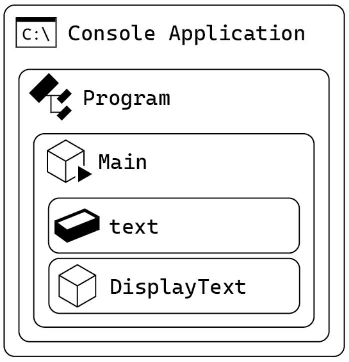
The following code map shows this organization:



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

SIDE QUEST

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 **`MethodThatUsesRecursion`**, 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.

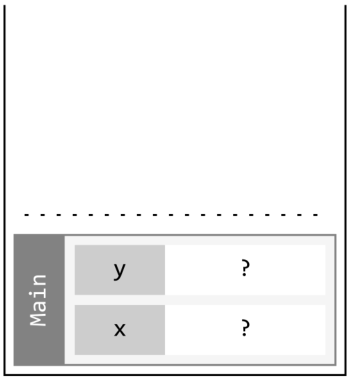
But when returning from a method, the method we are going back to is still alive. Data from any method in the chain back to the program's entry point is still alive because we will eventually return there.

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;
}
```

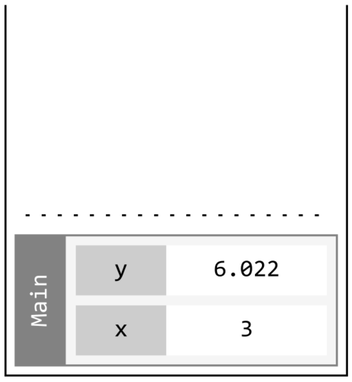
As our main method (referred to as **`Main`** below) starts, the stack looks something like this:



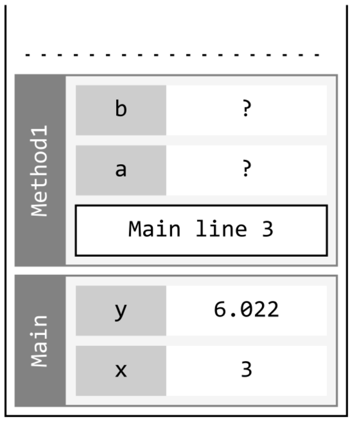
Four bytes are reserved for **`x`** because it is an **`int`**. Eight bytes are reserved for **`y`** because it is a **`double`**. So 12 bytes total are set aside for **`Main`** in a bundle. The image above shows them grouped and labeled on the side with **`Main`**. A collection of data needed for a 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:



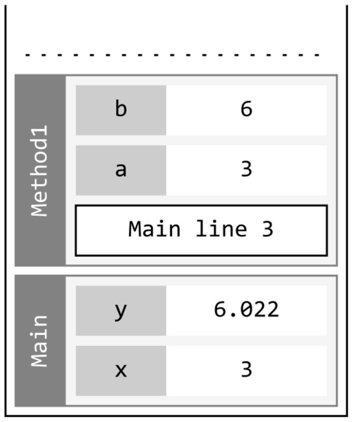
At this point, we're ready to run the third line, which invokes **`Method1`**. As this method is called, several things happen. We reserve more space for the variables in the method we are calling. We also note where we are at, so we can return to that spot. Both pieces of information go into the new 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.



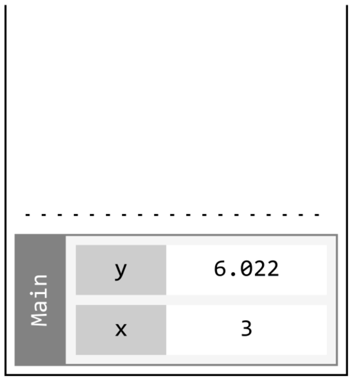
`Main`'s frame is buried beneath the one for **`Method1`**. The memory is still reserved for **`Main`**'s variables, but it is generally inaccessible. That's a good thing. **`Method1`** can work with its own variables as needed without interfering with **`Main`**'s variables.

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.

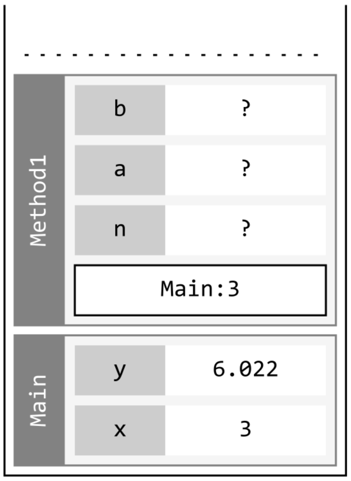
The code above only shows the main method calling a single other method, but if **`Method1`** called another method, another frame would be placed on top of the stack for it. Frames are added and removed from the top of the stack as needed, as methods are called and completed.

Parameters

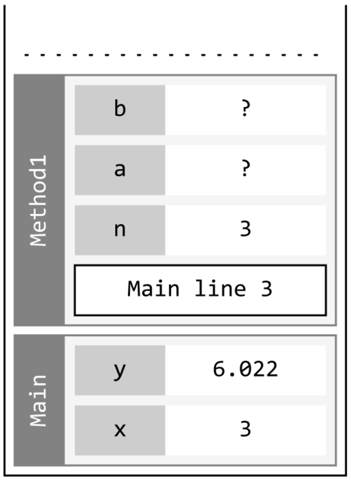
How do parameters work with the stack? Let's give **`Method1`** a parameter:

```
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.

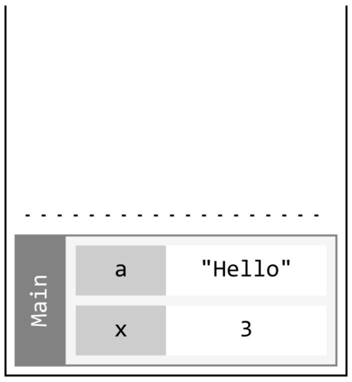
We could develop a scheme to allow stacks to deal with ever-changing sizes, but instead of doing that, C# uses a different approach for things of this nature and let the stack use predictable sizes for stack frames. This different approach is called a heap and is the subject of the next section.

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";
```

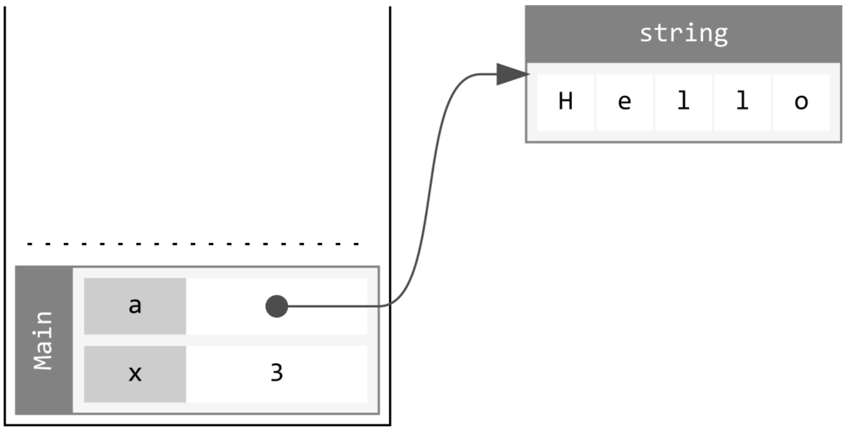
Knowing how the stack works, we might have assumed memory looks something like this:



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:

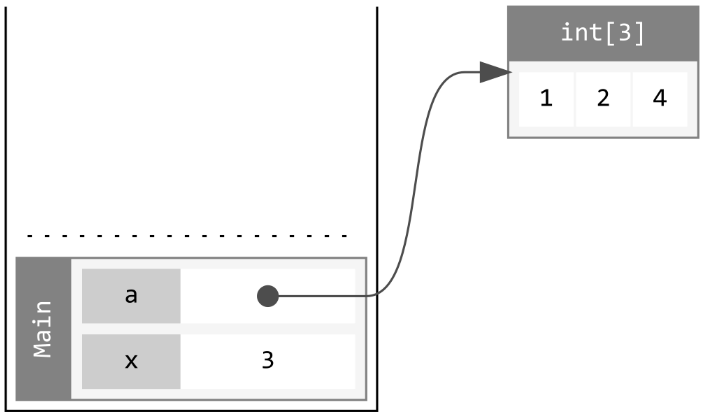


While the variable **`x`** contains its own data, **`a`** contains only the reference. All references are the same size, so we can count on frames for a method being known sizes. (References are 8 bytes (64 bits) on a 64-bit computer and 4 bytes (32 bits) on a 32-bit computer.)

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:

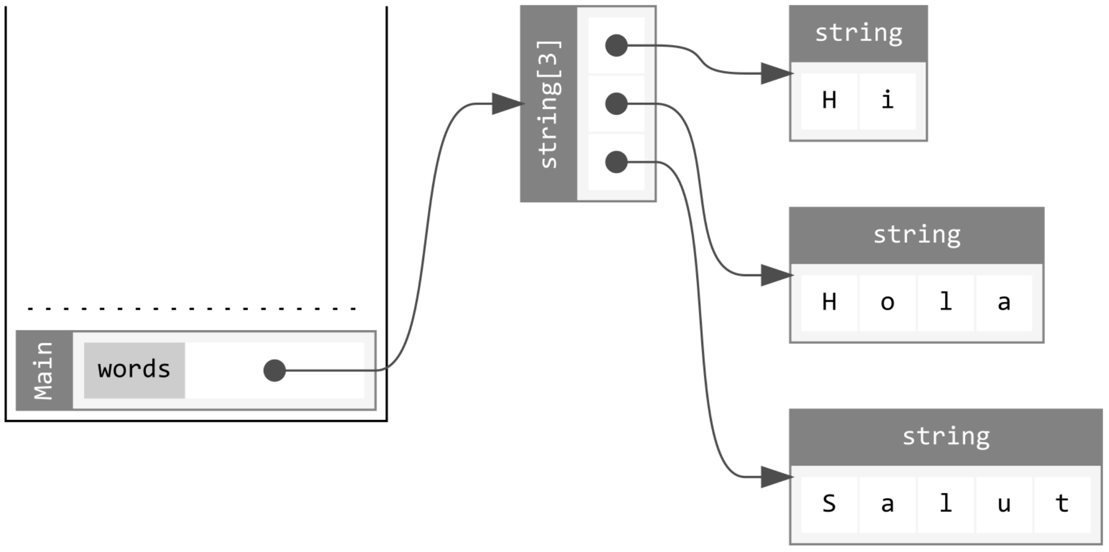


The Heap as a Graph of Objects

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.

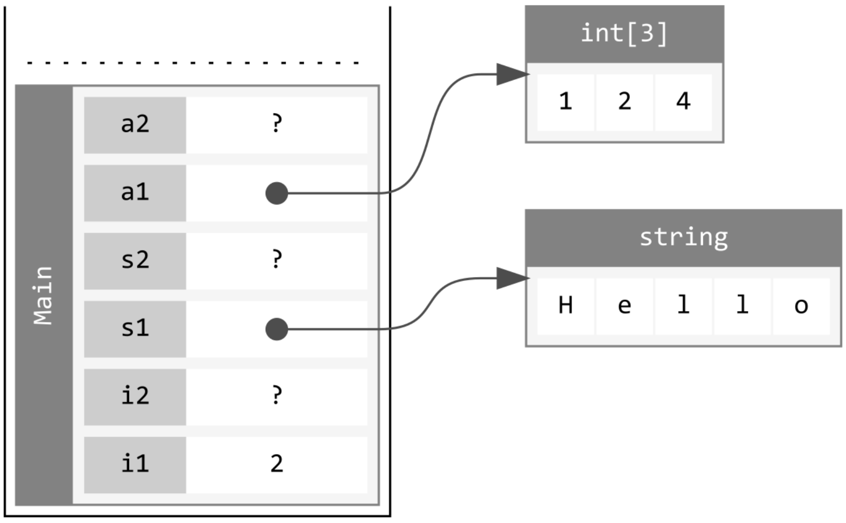
Most types we have discussed are value types. All the integer types (**`byte`**, **`sbyte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**), all the floating-point types (**`float`**, **`double`**, **`decimal`**), **`char`**, and **`bool`** are value types.

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;
```

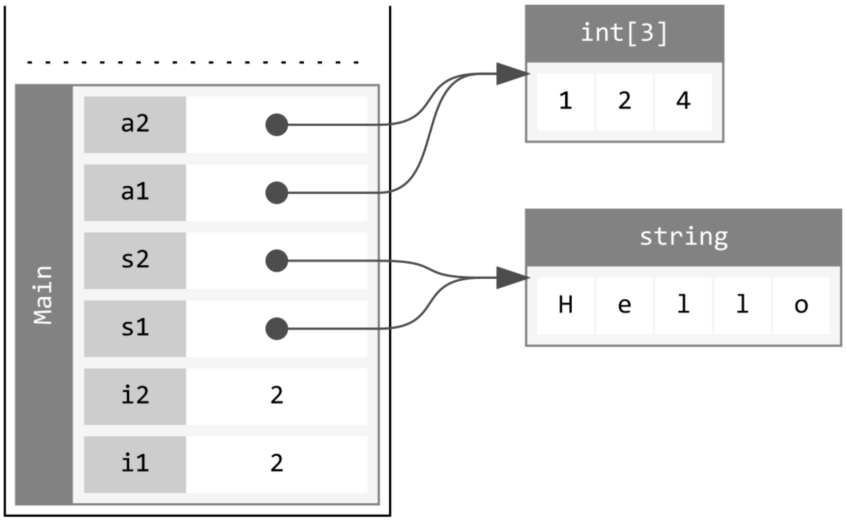
This code creates two **`int`**s, two **`string`**s, and two arrays, initializes new values for one set (the ones ending with a **`1`**), and then copies the first set's contents to the second's. Afterward, it makes two additional changes to the variables. After running through the first set of assignments (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**.



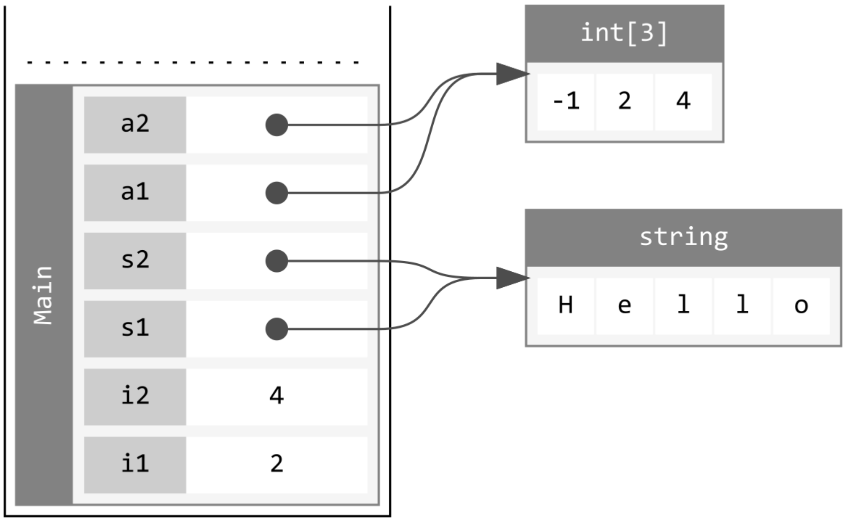
There is still only one occurrence of the string **"Hello"** on the heap, and only one **int** array on the heap, even though we have two variables with a reference to each.

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:

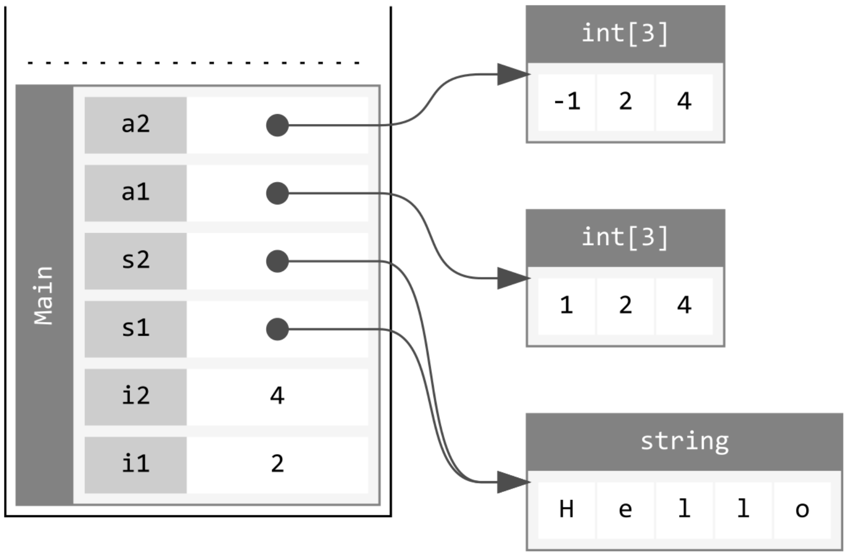


Even though we changed the array through **a2**'s reference, we can see the change through **a1** as well. **a1** has its own storage location, but it stores a reference that leads you to the same object on the heap. If we displayed **a1[0]** afterward, we would see that it is **-1**. The two variables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