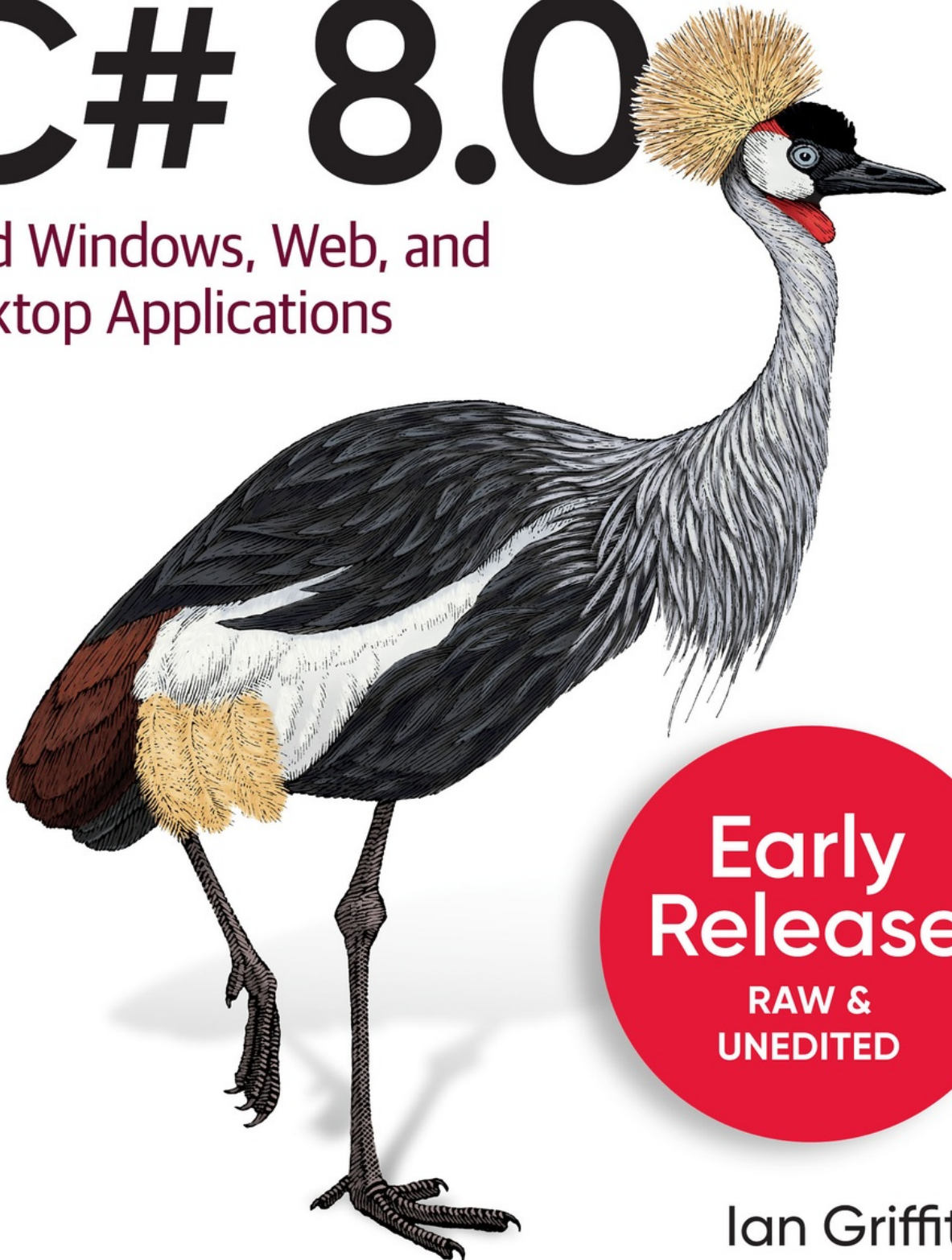


O'REILLY®

Programming C# 8.0

Build Windows, Web, and
Desktop Applications



Early
Release

RAW &
UNEDITED

Ian Griffiths

1. Preface

- a. Who This Book Is For
- b. Conventions Used in This Book
- c. Using Code Examples
- d. O'Reilly Online Learning
- e. How to Contact Us
- f. Acknowledgments

2. 1. Introducing C#

- a. Why C#?
- b. Why Not C#?
- c. C#'s Defining Features
 - i. Managed Code and the CLR
 - ii. Prefer Generality over Specialization
- d. Visual Studio and Visual Studio Code
- e. Anatomy of a Simple Program
 - i. Adding a Project to an Existing Solution
 - ii. Referencing One Project from Another
 - iii. Referencing External Libraries
 - iv. Writing a Unit Test
 - v. Namespaces
 - vi. Classes
 - vii. Program Entry Point

viii. Unit Tests

f. Summary

3. 2. Basic Coding in C#

a. Local Variables

i. Scope

b. Statements and Expressions

i. Statements

ii. Expressions

c. Comments and Whitespace

d. Preprocessing Directives

i. Compilation Symbols

ii. #error and #warning

iii. #line

iv. #pragma

v. #region and #endregion

e. Intrinsic Data Types

i. Numeric Types

ii. Booleans

iii. Strings and Characters

iv. Tuples

v. Dynamic

vi. Object

f. Operators

g. Flow Control

- i. Boolean Decisions with if Statements
- ii. Multiple Choice with switch Statements
- iii. Loops: while and do
- iv. C-Style for Loops
- v. Collection Iteration with foreach Loops

h. Patterns

- i. Summary

Programming C# 8.0

Build Windows, Web,
and Desktop Applications

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Ian Griffiths



Programming C# 8.0

by Ian Griffiths

Copyright © 2020 Ian Griffiths. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales
promotional use. Online editions are also available for most titles
(<https://oreilly.com>). For more information, contact our
corporate/institutional sales department: 800-998-9938 or
corporate@oreilly.com.

Editors: Tyler Ortman and Corbin Collins

Production Editor: Deborah Baker

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

November 2019: First Edition

Revision History for the Early Release

- 2019-05-24: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492056812> for release

details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Programming C# 8.0*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-05674-4

[LSI]

Dedication

I dedicate this book to my excellent wife Deborah, and to my wonderful daughters, Hazel and Victoria. Thank you for enriching my life.

Preface

C# is now well into its second decade. It has grown steadily in both power and size, but Microsoft has always kept the essential characteristics intact—C# still feels like the same language as was first unveiled back in 2000. Each new capability is designed to integrate cleanly with the rest, enhancing the language without turning it into an incoherent bag of miscellaneous features. This philosophy is evident in the most important new addition to C#—its support for asynchronous programming. It has always been possible to use asynchronous APIs in C#, but in the past, this tended to involve convoluted code. In C# 5.0, you can write asynchronous code that looks almost exactly like normal code, so instead of adding weight to the language, this new asynchronous programming support makes things simpler.

Even though C# continues to be a fairly straightforward language at its heart, there is a great deal more to say about it now than in its first incarnation. Successive editions of this book have responded to the language's progress with ever-increasing page counts, but this latest edition does not merely try to cram in yet more details. It expects a somewhat higher level of technical ability from its readers than before.

Who This Book Is For

I have written this book for experienced developers—I’ve been programming for years, and I’ve set out to make this the book I would want to read if that experience had been in other languages, and I were learning C# today. Whereas previous editions explained some basic concepts such as classes, polymorphism, and collections, I am assuming that readers will already know what these are. The early chapters still describe how C# presents these common ideas, but the focus is on the details specific to C#, rather than the broad concepts. So if you have read previous editions of this book, you will find that this one spends less time on these basic concepts, and goes into rather more detail on everything else.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the

user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://oreil.ly/programmingcsharp>.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly

books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *Programming C# 8.0* by Ian Griffiths (O'Reilly). Copyright 2020 by Ian Griffiths, 978-1-492-05681-2.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning

NOTE

For almost 40 years, *O'Reilly* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://www.oreilly.com/catalog/9781492056812>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

Many thanks to the book's official technical reviewers: Glyn Griffiths, Alex Turner, and Chander Dhall. I'd also like to give a big thank you to those who reviewed individual chapters, or otherwise offered help or information that improved this book: Brian Rasmussen, Eric Lippert, Andrew Kennedy, Daniel Sinclair, Brian Randell, Mike Woodring, Mike Taulty, Mary Jo Foley, Bart De Smet, and Stephen Toub.

Thank you to everyone at O'Reilly whose work brought this book into existence. In particular, thanks to Rachel Roumeliotis for encouraging me to write this new edition, and thank you also to Kristen Borg, Rachel Monaghan, Gretchen Giles, and Yasmina Greco for your excellent support. Finally, thank you to John Osborn, for taking me on as an O'Reilly author back when I wrote my first book.

Chapter 1. Introducing C#

The C# programming language (pronounced “see sharp”) can be used for many kinds of applications, including websites, desktop applications, embedded controllers, mobile apps, games, and command-line utilities. C# has been center stage for Windows developers for almost two decades, but in recent years, it has also made inroads into other platforms. In June 2016, Microsoft released the first version of .NET Core, a cross-platform version of the runtime and libraries behind C#, enabling web apps, microservices, and console applications written in C# to run on macOS, and Linux, as well as on Windows.

This push into other platforms has gone hand in hand with Microsoft’s embrace of open source development. In C#’s early history, Microsoft guarded all of its source code closely.¹ But today, pretty much everything surrounding C# is developed in the open, with code contributions from outside of Microsoft being welcome. In 2014, the .NET Foundation (<https://dotnetfoundation.org/>) was created to foster the development of open source projects in the .NET world, and many of Microsoft’s most important C# and .NET projects are now under the foundation’s governance (in addition to many non-Microsoft projects). This includes Microsoft’s C# compiler, which can be found at <https://github.com/dotnet/roslyn>, and also .NET Core, which is at <https://github.com/dotnet/core>, comprising the runtime, framework class library, and tooling for creating .NET projects.

Why C#?

Although there are many ways you can use C#, other languages are always an option. Why might you choose C# over them? It will depend on what you need to do, and what you like and dislike in a programming language. I find that C# provides considerable power and flexibility, and works at a high enough level of abstraction that I don't expend vast amounts of effort on little details not directly related to the problems my programs are trying to solve. (I'm looking at you, C++.)

Much of C#'s power comes from the range of programming techniques it supports. For example, it offers object-oriented features, generics, and functional programming. It supports both dynamic and static typing. It provides powerful list- and set-oriented features, thanks to Language Integrated Query (LINQ). It has intrinsic support for asynchronous programming.

Some of the greatest benefits of using C# come from its runtime, which provides services such as security sandboxing, runtime type checking, exception handling, and thread management. The runtime also provides a garbage collector (GC) that frees developers from much of the work associated with recovering memory that the program is no longer using. A GC is a common feature in modern programming languages, and while it is a boon for most programs, there are some specialized scenarios where its performance implications are problematic, so C# 7.2 (released in 2017) added various features that enable more explicit memory management, trading runtime performance for ease of development, but all without the loss of type safety. This enables C# to move into certain performance critical applications that for years were the preserve of less safe languages such as C and C++.

Of course, languages do not exist in a vacuum—high-quality libraries with

a broad range of features are essential. There are some elegant and academically beautiful languages that are glorious right up until you want to do something prosaic, such as talking to a database or determining where to store user settings. No matter how strong a set of programming idioms a language offers, it also needs to provide full and convenient access to the underlying platform's services. C# is on very strong ground here, thanks to its runtime, the framework class library, and extensive third-party library support.

Before I can talk about the .NET runtime and class library, I need to address the fact that there are two current versions of .NET: the .NET Framework (Windows only) and .NET Core (cross-platform, open-source). As the sidebar [“Two .NETs \(Temporarily\)”](#) describes, these use slightly different names for the runtime and library. The names will change again when .NET Core becomes the only option, at which point there will be no need to qualify which particular flavour is in use, so we can just say “.NET”. That is the terminology I will use for the rest of this book, but be aware that unless stated otherwise, everything I'm saying applies both to .NET Core and the .NET Framework, and to the .NET that will replace them both.

TWO .NETS (TEMPORARILY)

For many years, there was only one current version of .NET at any given time, but the introduction of .NET Core confused things a little, because as I write this, Microsoft ships two current versions of .NET, with similar but different names for all the elements. In May of 2019 Microsoft announced that they intend to revert to a single current version in November 2020. In the long run this will reduce confusion, but in the near term it complicates matters further by introducing a third set of names.

Before 2016, *.NET Framework* meant the combination of two things: a runtime, and the class library. The runtime was called the Common Language Runtime (CLR) because it supports multiple languages (including C#, VB, and F#). The class library went by various names including Base Class Library (BCL; a slightly confusing name, because the ECMA specifications define the term BCL as something much narrower), or the Framework Class Library.

Now we have .NET Core. Its runtime is called the .NET Core Common Language Runtime (or just CoreCLR), which is a straightforward enough name: we can talk about the .NET Core CLR or the .NET

Framework CLR, and it's obvious which one we mean. The problem is the class library, which in the world of .NET Core is called the .NET Core Framework (or CoreFX). This is somewhat unhelpful, because before .NET Core, "Framework" meant the combination of the CLR and the library. And just to muddy the waters further, many people at Microsoft now refer the .NET Framework as the "desktop" framework just to make it clear that they're not talking about .NET Core (which is somewhat unhelpful because plenty of people use this "desktop" version for server applications). Just in case that's not all crystal clear, here's a table:

Platform	Runtime	Class library
.NET Core	.NET Core CLR	.NET Core Framework
.NET Framework (aka .NET desktop)	.NET CLR	.NET Framework Class Library

In 2020, the names will all adjust again, with both .NET Core and .NET Framework being superceded by plain ".NET". Microsoft has not settled on definitive names for the corresponding runtime and library parts at the time of writing this.

Each platform can do things the other cannot, which is why both have shipped concurrently. The .NET Framework only runs on Windows, and supports some Windows-specific features. For example, there is a section of the .NET Framework Class Library dedicated to working with Windows speech synthesis and recognition services. This isn't possible on .NET Core, because it might be running on Linux, where equivalent features either don't exist or are too different to be presented through the same .NET API.

The new .NET due to ship in 2020 is essentially .NET Core renamed. .NET Core has been where most of the new development of .NET has occurred for the last few years. .NET Framework is still fully supported, but is already falling behind. For example, version 3.0 of Microsoft's web application framework, ASP.NET Core, will only run on .NET Core, and not .NET Framework. So .NET Frameworks' retirement, and .NET Core's promotion to the one true .NET is the inevitable conclusion of a process that has been underway for a few years.

.NET Core encompasses both the runtime and the library that C# programs use. The runtime part is called the *Common Language Runtime* (usually abbreviated to CLR) because it supports not just C#, but any .NET language. Microsoft also offers Visual Basic, F#, and .NET extensions for C++, for example. The CLR has a *Common Type System* (CTS) that enables code from multiple languages to interoperate freely, which means that .NET libraries can usually be used from any .NET language—F# can consume libraries written in C#, C# can use Visual Basic libraries, and so on.

In addition to the runtime, there is an extensive class library. This library

provides wrappers for many features of the underlying operating system (OS), but it also provides a considerable amount of functionality of its own, such as collection classes or JSON processing.

The class library built into .NET is not the whole story—many other systems provide their own .NET libraries. For example, there are extensive libraries that enable C# programs to use popular cloud services. As you'd expect, Microsoft provides comprehensive .NET libraries for working with services in their *Azure* cloud platform. Likewise, Amazon provides a fully-featured SDK for using Amazon Web Services (AWS) from C# and other .NET languages. And libraries do not have to be associated with frameworks. There's a large ecosystem of .NET libraries, some commercial and some free and open source, including mathematical utilities, parsing libraries, and user interface components, to name just a few.

Even if you get unlucky and need to use an OS feature that doesn't have any .NET library wrappers, C# offers various mechanisms for working with other kinds of APIs, such as the C-style APIs available in Win32 and Linux, or with COM (although since COM is Windows-specific, you can only use that on Windows). Some aspects of the interoperability mechanisms are a little clunky, and if you need to deal with an existing component, you might need to write a thin wrapper that presents a more .NET-friendly face. (You can still write the wrapper in C#. You'd just be putting the awkward interoperability details in one place, rather than letting them pollute your whole codebase.)

In summary, with C# we get a strong set of abstractions built into the language, a powerful runtime, and easy access to an enormous amount of library and platform functionality.

Why Not C#?

To understand a language, it's useful to compare it with alternatives, so it's worth looking at some of the reasons why you might choose some other language. One of C#'s nearest competitors is Visual Basic (VB), another native .NET language that offers most of the same benefits as C#. For many years, the choice here was mostly a matter of syntax. C# is part of the C family of languages, and if you are familiar with at least one language from that group (which includes C, C++, Objective-C, Java, and JavaScript), you will feel instantly at home with C#'s syntax. However, if you do not know any of those languages, but you are at home with pre-.NET versions of Visual Basic, or with the scripting variants such as Microsoft Office's Visual Basic for Applications (VBA), then the .NET version of VB would certainly be easier to learn. However, VB's popularity has gradually declined to the point where Microsoft no longer provides feature parity with C#. For example, the ASP.NET Core web application framework supports the use of C# code fragments mixed in with HTML in its *Razor* engine, but not VB. So it would be hard to recommend VB for new projects, except for developers who are already at home with it, and who do not need features for which VB support is unavailable.

Microsoft offers another language designed specifically for .NET, called F#. This is a very different language from C# and Visual Basic, and it seems to be most popular in computation-intensive applications such as engineering, and the more technical areas of finance. F# is primarily a functional programming language, but with strong support for object-oriented programming, and low-friction interoperability with .NET components. (You can use any .NET library from F# just as easily as you can from C#.) It is good for expressing particularly complex computations,

so if you're working on applications that spend much more of their time thinking than doing, F# may be for you.

Then there's C++, which has always been a mainstay of Windows development. The C++ language is always evolving, and in its current standard, C++17 (ISO/IEC standard 14882:2017, to use its formal name), the language has numerous features that make it significantly more expressive than earlier versions (particularly compared versions of the language from before the 2011 revision). It's now much easier to use functional programming idioms, for example. In many cases, C++ code can provide significantly better performance than .NET languages, partly because C++ lets you get closer to the underlying machinery of the computer, and partly because the CLR has much higher overheads than the rather frugal C++ runtime. (That said, the features added in C# 7.2 narrow this gap considerably.) Also, many Win32 APIs are less of a hassle to use in C++ than C#, and the same is true of some (although not all) COM-based APIs. For example, C++ is the language of choice for using the most recent versions of Microsoft's advanced graphics API, DirectX. Microsoft's C++ compiler even includes extensions that allow C++ code to integrate with the world of .NET, meaning that C++ can use the entire .NET class library (and any other .NET libraries). So on paper, C++ is a very strong contender. But one of its greatest strengths is also a weakness: the level of abstraction in C++ is much closer to the underlying operation of the computer than in C#. This is part of why C++ can offer better performance and make certain APIs easier to consume, but it also tends to mean that C++ requires considerably more work to get anything done. Even so, the trade-off can leave C++ looking preferable to C# in some scenarios.

NOTE

Because the CLR supports multiple languages, you don't have to pick just one for your whole project. It's common for primarily C#-based projects to use C++ to deal with a non-C#-friendly API, using the .NET extensions for C++ (officially called *C++/CLI*) to present a C#-friendly wrapper. The freedom to pick the best tool for the job is useful, but there is a price. The mental "context switch" developers have to make when moving between languages takes its toll, and could outweigh the benefits. Mixing languages works best when each language has a very clearly defined role in the project, such as dealing with gnarly APIs.

Of course, there are many languages that don't use .NET. A natural competitor in this category is Java, along with languages that run on the Java Virtual machine (JVM) such as Scala and Clojure. There are many similarities between the .NET and Java world, not least because some design decisions in the CLR were made to make interoperability with Java a possibility. Early in C#'s history it was sometimes described pejoratively as a lightly modified copy of Java, although over time, Java's rate of development has slowed, and now Java is playing catching, having introduced a few features copied from C#. But while C# now has the expressive edge, they're still pretty closely matched in terms of basic capabilities. Historically, one big reason for choosing a JVM-based language over a .NET one was Java's better cross-platform support. However, the emergence of .NET Core, with its first class support for non-Windows platforms, has removed this difference. The main reason for choosing Java over C# these days would typically be the expertise your developers have. And it's the same for many other languages. If you have a development shop full of Ruby experts, choosing C# for your next web application might not be the most effective use of the available talent.

So not every project will use C#. But since you've read this far,

presumably you're still considering using C#. So what is C# like?

C#'s Defining Features

Although C#'s most superficially obvious feature is its C-family syntax, perhaps its most distinctive feature is that it was the first language designed to be a native in the world of the CLR. As the name suggests, the Common Language Runtime is flexible enough to support many languages, but there's an important difference between a language that has been extended to support the CLR, and one that puts it at the center of its design. The .NET extensions in Microsoft's C++ compiler illustrate this—the syntax for using those features is visibly different from standard C++, making a clear distinction between the native world of C++ and the outside world of the CLR. But even without different syntax,² there would still be friction when two worlds have different ways of working. For example, if you need a dynamically resizable collection of numbers, should you use a standard C++ collection class such as `vector<int>`, or one from .NET such as `List<int>`? Whichever you choose, it will be the wrong type some of the time: C++ libraries won't know what to do with a .NET collection, while .NET APIs won't be able to use the C++ type.

C# embraces .NET, both the runtime and the class library, so these dilemmas do not arise. In the scenario just discussed, `List<int>` has no rival. There is no friction when using .NET's class library because it is built for the same world as C#.

That much is also true of Visual Basic, but that language retains links to a pre-.NET world. The .NET version of Visual Basic is in many respects a quite different language than its predecessors, but Microsoft went to some

lengths to retain many aspects of older versions. The upshot is that it has carried over several language features that are somewhat removed from how the CLR works. There's nothing wrong with that, of course. That's what compilers usually do, and in fact C# has steadily added its own abstractions. But the first version of C# presented a model that was very closely related to the CLR's own model, and the abstractions added since have been designed to fit well with the CLR. This gives C# a distinctive feel.

This means that if you want to understand C#, you need to understand the CLR, and the way in which it runs code. (By the way, I will mainly talk about Microsoft's implementations in this book, but there are specifications that define language and runtime behavior for all C# implementations.)

C#, THE CLR, AND STANDARDS

The CLR is the name for Microsoft's implementations of the runtime for .NET languages such as C# and Visual Basic. (As discussed earlier, there are currently two versions: the .NET Framework CLR, and the .NET Core CLR. But either way "CLR" refers to a Microsoft runtime.) Other implementations of .NET, such as Mono (popular in cross-platform mobile app development), do not use the CLR, but they have something equivalent. The standards body ECMA has published OS-independent specifications for the various elements required by a C# implementation, and these define more generic names for the various parts. There are two documents: ECMA-334 is the C# Language Specification and ECMA-335 defines the *Common Language Infrastructure* (CLI), the virtual environment in which C# programs run. These have also been published by the International Standards Organization as ISO/IEC 23270:2018 and ISO/IEC 23271:2012 respectively. The '2018' gives the impression that the C# specification is more up to date than it really is. In fact, both the ECMA and ISO language standards correspond to version 5.0 of C#. At the time of writing this, ECMA is working on an updated language specification, but be aware that these standards are typically several years behind the state of the art. While the IEC CLI standard has an even older date, 2012, (the ECMA spec is of the same age) the runtime specifications change less often than the language, so the CLI spec is much closer to current implementations, despite the name suggesting the opposite.

Version drift notwithstanding, it's not quite accurate to say that the CLR is Microsoft's implementation of the CLI because the scope of the CLI is slightly broader. ECMA-335 defines not just the runtime behavior (which it calls the *Virtual Execution System*, or VES), but also the file format for executable and library files, the Common Type System, and a subset of the CTS that languages are expected to be able to support to guarantee interoperability between languages, called the Common Language Specification (CLS).

So you could say that Microsoft's CLI is all of .NET rather than just the CLR, although .NET includes a lot of additional features not in the CLI specification. (For example, the class library that the CLI demands

makes up only a small subset of .NET's much larger library.) The CLR is effectively .NET's VES, but you hardly ever see the term VES used outside of the specification, which is why I mostly talk about the CLR in this book. However, the terms CTS and CLS are more widely used, and I'll refer to them again in this book.

Managed Code and the CLR

For years, the most common way for a compiler to work was to process source code, and to produce output in a form that could be executed directly by the computer's CPU. Compilers would produce *machine code*—a series of instructions in whatever binary format was required by the kind of CPU the computer had. Many compilers still work this way, but the C# compiler does not. Instead, it uses a model called *managed code*.

With managed code, the runtime generates the machine code that the CPU executes, not the compiler. This enables the runtime to provide services that are hard or even impossible to provide under the more traditional model. The compiler produces an intermediate form of binary code, the *intermediate language* (IL), and the runtime provides the executable binary at runtime.

Perhaps the most visible benefit of the managed model is that the compiler's output is not tied to a single CPU architecture. You can write a .NET component that can run on the 32-bit x86 architecture that PCs have used for decades, but that will also work well in the newer 64-bit update to that design (x64), and also on completely different architectures such as ARM. (For example, .NET Core can run on ARM-based devices such as the Raspberry Pi.) With a language that compiles directly to machine code, you'd need to build different binaries for each of these. But with .NET, you can compile a single component that can run on any of them, and it would even be able to run on platforms that weren't supported at the time you compiled the code if a suitable runtime became available in the future.

More generally, any kind of improvement to the CLR's code generation—whether that's support for new CPU architectures, or just performance improvements for existing ones—is instantly of benefit to all .NET languages.

The exact moment at which the CLR generates executable machine code can vary. Typically, it uses an approach called *just in time* (JIT) compilation, in which each individual function is compiled at runtime, the first time it runs. However, it doesn't have to work this way. In principle, the CLR could use spare CPU cycles to compile functions it thinks you may use in the future (based on what your program did in the past). Or you can get more aggressive: there are various ways in which .NET code can be compiled *ahead of time* (AoT), although in some cases the CLR may choose to recompile code at runtime if doing so can improve performance. Even without AoT, the CLR can sometimes regenerate code at runtime some time after the initial JIT compilation. Diagnostics tools can trigger this, but the CLR could also choose to recompile to better optimize code for the way it is being used. The virtualized nature of managed execution is designed to make such things possible in a way that's invisible to your code. Occasionally, it can make its presence felt through more than just performance. For example, virtualized execution leaves some latitude for when and how the runtime performs certain initialization work, and you can sometimes see the results of its optimizations causing things to happen in a surprising order.

Processor-independent JIT compilation is not the main benefit offered by managed code. The greatest payoff is the set of services the runtime provides. One of the most important of these is memory management. The runtime provides a *garbage collector* (GC), a service that automatically frees memory that is no longer in use. This means that in most cases, you

do not need to write code that explicitly returns memory to the operating system once you have finished using it. Depending on which languages you have used before, either this will be wholly unremarkable, or it will profoundly change how you write code.

WARNING

Although the garbage collector does take care of most memory handling issues, you can defeat its heuristics, and that sometimes happens by accident. I describe the GC's operation in detail later in the book.

Managed code has ubiquitous type information. The file formats dictated by the CLI require this to be present, because it enables certain runtime features. For example, .NET offers various automatic serialization services, in which objects can be converted into binary or textual representations of their state, and those representations can later be turned back into objects, perhaps on a different machine. This sort of service relies on a complete and accurate description of an object's structure, something that's guaranteed to be present in managed code. Type information can be used in other ways. For example, unit test frameworks can use it to inspect code in a test project and discover all of the unit tests you have written. This relies on the CLR's *reflection* services.

Although C#'s close connection with the runtime is one of its main defining features, it's not the only one. Visual Basic has a similar connection with the CLR, but C# is distinguished from Visual Basic by more than just syntax: it also has a somewhat different philosophy.

Prefer Generality over Specialization

C# favors general-purpose language features over specialized ones. Over the years, Microsoft has expanded C# several times, and the language's designers always have specific scenarios in mind for new features. However, they have always tried hard to ensure that each new element they add is useful beyond the scenario for which it was designed.

For example, a few years ago Microsoft decided to add features to C# to make database access feel well integrated with the language. The resulting technology, Language Integrated Query (LINQ), certainly supports that goal, but Microsoft achieved this without adding any direct support for data access to the language. Instead, they introduced a series of quite diverse-seeming capabilities. These included better support for functional programming idioms, the ability to add new methods to existing types without resorting to inheritance, support for anonymous types, the ability to obtain an object model representing the structure of an expression, and the introduction of query syntax. The last of these has an obvious connection to data access, but the rest are harder to relate to the task at hand. Nonetheless, these can be used collectively in a way that makes certain data access tasks significantly simpler. But the features are all useful in their own right, so as well as supporting data access, they enable a much wider range of scenarios. For example, these additions (which arrived in C# 3.0) made it very much easier to process lists, sets, and other groups of objects, because the new features work for collections of things from any origin, not just databases.

Perhaps the clearest illustration of this philosophy of generality was a language feature that C# chose not to implement, but that Visual Basic did. In VB, you can write XML directly in your source code, embedding expressions to calculate values for certain bits of content at runtime. This compiles into code that generates the completed XML at runtime. VB also

has intrinsic support for queries that extract data from XML documents. These same concepts were considered for C#. Microsoft Research developed extensions for C# that supported embedded XML, which were demonstrated publicly some time before the first release of Visual Basic that did so. Nevertheless, this feature didn't ultimately make it into C#. It is a relatively narrow facility, only useful when you're creating XML documents. As for querying XML documents, C# supports this functionality through its general-purpose LINQ features, without needing any XML-specific language features. XML's star has waned since this language concept was mooted, having been usurped in many cases by JSON (which will doubtless be eclipsed by something else in years to come). Had embedded XML made it into C#, it would by now feel like a slightly anachronistic curiosity.

The new features added in subsequent versions of C# continue in the same vein. For example, the deconstruction and pattern matching features added in C# versions 7 and 8 are aimed at making life easier in subtle but useful ways, and not at any particular application area.

I've now described some of the defining features of C#, but Microsoft provides more than just a language and runtime. There are also two development environments that can help you write, test, debug, and maintain your code.

Visual Studio and Visual Studio Code

Microsoft offers three development environments: Visual Studio, Visual Studio for Mac, and Visual Studio Code. Visual Studio has been around the longest. There are various editions of it, ranging from free to eye-wateringly expensive. All versions provide the basic features—such as a

text editor, build tools, and a debugger—as well as visual editing tools for user interfaces. Visual Studio provides the most extensive support for developing C# applications, whether those applications will run on Windows or other platforms. It has been around for as long as C#, so it comes from the pre-open source days, and has not moved over to open source development.

Visual Studio is an Integrated Development Environment (IDE), so it takes an “everything included” approach. In addition to a fully-featured text editor, there is deep integration with source control systems such as git, and with online systems providing source repositories, issue tracking, and other Application Lifecycle Management (ALM) features such as GitHub and Microsoft’s Azure DevOps system. Visual Studio offers built-in performance monitoring and diagnostic tools. It has various features for working with applications developed for and deployed to Microsoft’s Azure cloud platform. Its *Live Share* feature offers a convenient way for remote developers to work simultaneously to aid pairing or code review.

In 2017 Microsoft released Visual Studio for Mac. Despite what the name suggests, it is not quite the same product, not least because it was designed as a Mac application and not a port of the Windows version. It grew out of a product called Xamarin, a Mac-based development environment specializing in building mobile apps in C#. Xamarin was originally an independent product, but Microsoft acquired it, and integrated various features from the Windows version of Visual Studio when they moved product under the Visual Studio brand.

Visual Studio Code (often shortened to VS Code) was first released in 2015. It is open source and cross platform, supporting Linux as well as Windows and Mac. It is based on the Electron platform and is written

predominantly in TypeScript. (This means it really is the same program on all platforms.) VS Code is a more lightweight product than Visual Studio: a basic installation of VS Code has little more than basic text editing support. However, as you open up files, it will discover downloadable extensions which, if you choose to install them, add in support for C#, F#, TypeScript, PowerShell, Python, and a wide range of other languages. So although in its initial form it is less of an Integrated Development Environment (IDE) and more like a simple text editor, its extensibility model makes it pretty powerful. The wide range of extensions has led to VS Code becoming remarkably popular outside of the world of Microsoft languages, and this in turn has encouraged a virtuous cycle of even greater growth in the range of extensions.

Visual Studio offers the most straightforward path to getting started in C#—you don't need to install any extensions or modify any configuration to get up and running. So I'll start with a quick introduction to working in Visual Studio.

NOTE

You can download the free version, called Visual Studio Community, from <https://www.visualstudio.com/>.

Any nontrivial C# project will have multiple source code files, and in Visual Studio, these will belong to a *project*. Each project builds a single output, or *target*. The build target might be as simple as a single file—a C# project could produce an executable file or a library,³ for example—but some projects produce more complicated outputs. For instance, some project types build websites. A website will normally contain multiple

files, but collectively, these files represent a single entity: one website. Each project's output will typically be deployed as a unit, even if it consists of multiple files.

Project files usually have extensions ending in *proj*. For example, most C# projects have a *.csproj* extension, while C++ projects use *.vcxproj*. If you examine these files with a text editor, you'll find that they usually contain XML. (That's not always true. Visual Studio is extensible, and each type of project is defined by a *project system* that can use whatever format it likes, but the built-in languages use XML.) These files list the contents of the project and configure how it should be built. The XML format that Visual Studio uses for C# project files can also be processed by the *msbuild* tool, which enables you to build projects from the command line. VS Code can also work with these files.

You will often want to work with groups of projects. For example, it is good practice to write tests for your code, but most test code does not need to be deployed as part of the application, so you would typically put automated tests into separate projects. And you may want to split up your code for other reasons. Perhaps the system you're building has a desktop application and a website, and you have common code you'd like to use in both applications. In this case, you'd need one project that builds a library containing the common code, another producing the desktop application executable, another to build the website, and three more projects containing the unit tests for each of the main projects.

Visual Studio helps you to work with multiple related projects through what it calls a *solution*. A solution is simply a collection of projects, and while they are usually related, they don't have to be—a solution is really just a container. You can see the currently loaded solution and all of its

projects in Visual Studio's *Solution Explorer*. [Figure 1-1](#) shows a solution with two projects. (I'm using Visual Studio 2019 here, which is the latest version at the time of writing this.) The body of this panel is a tree view, and you can expand each project to see the files that make up that project. This panel is normally open at the top right of Visual Studio, but it's possible to hide or close it. You can reopen it with the View → Solution Explorer menu item.

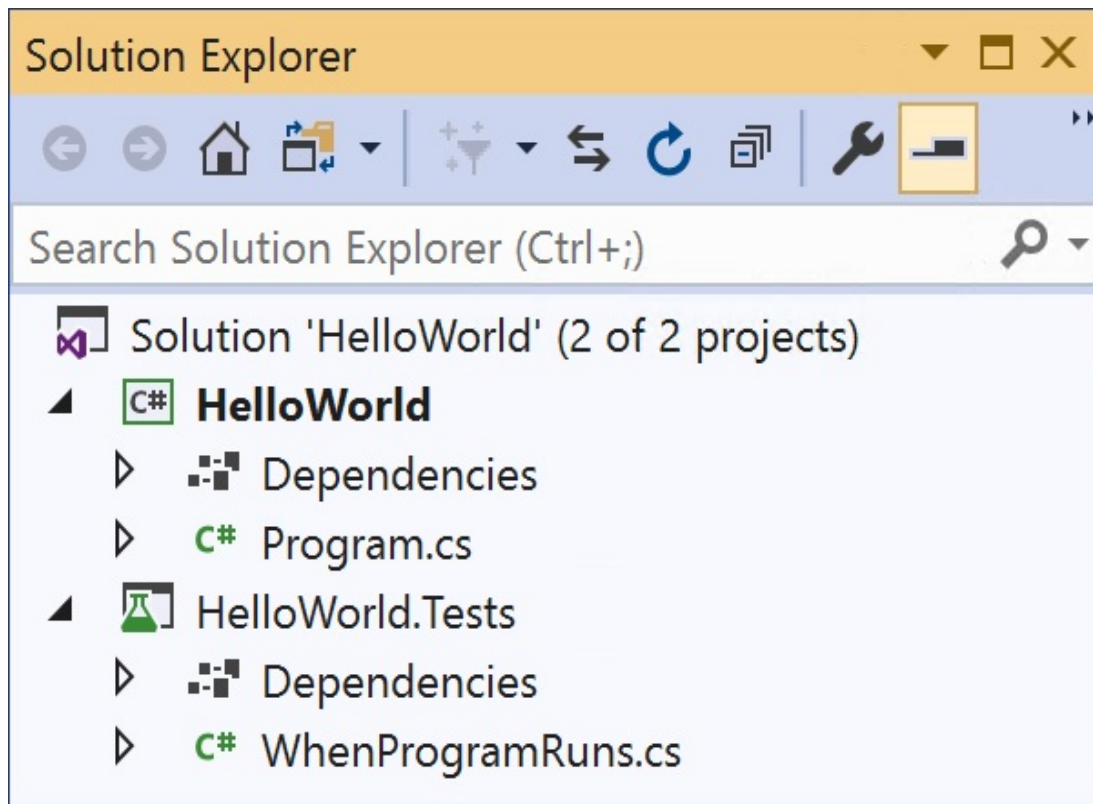


Figure 1-1. Solution Explorer

Visual Studio can load a project only if it is part of a solution. When you create a brand-new project, you can add it to an existing solution, but if you don't, Visual Studio will create one for you; if you try to open an existing project file, Visual Studio will look for an associated solution, and if it can't find one, it will insist that you either provide one or let it create one. That's because lots of operations in Visual Studio are scoped to the

currently loaded solution. When you build your code, it's normally the solution that you build. Configuration settings, such as a choice between Debug and Release builds, are controlled at the solution level. Global text searches can search all the files in the solution.

A solution is just another text file, with an *.sln* extension. Curiously, it's not an XML file—solution files contain plain text, although also in a format that *msbuild* understands, as does VS Code. If you look at the folder containing your solution, you may also notice a *.vs* folder or. (Visual Studio marks this as hidden, but if you have configured Windows File Explorer to show hidden files, as developers often do, you'll see it.) This contains per-user settings, such as a record of which files you have open, and which project or projects to launch when starting debug sessions. That ensures that when you open a project, everything is more or less where you left it when you last worked on the project. Because these are per-user settings, you do not normally check *.vs* folders into source control.

A project can belong to more than one solution. In a large codebase, it's common to have multiple *.sln* files with different combinations of projects. You would typically have a master solution that contains every single project, but not all developers will want to work with all the code all of the time. Someone working on the desktop application in our hypothetical example will also want the shared library, but probably has no interest in loading the web project.

I'll show how to create a new project and solution, and I'll then walk through the various features Visual Studio adds to a new C# project as an introduction to the language. I'll also show how to add a unit test project to the solution.

NOTE

This next section is intended for developers who are new to Visual Studio. This book is aimed at experienced developers, but does not assume any prior experience in C# or Visual Studio, so if you are already familiar with Visual Studio's basic operation, you might want to skim through this next section quickly.

Anatomy of a Simple Program

If you're using Visual Studio 2019, the simplest way to create a new project is through the Get Started window that appears when you first run it, as shown in [Figure 1-2](#).



Visual Studio 2019

Open recent



ConvertToAsciiDoc.sln

03/05/2019 12:09

C:\dev\oreillymedia\HtmlBookToAsciiDoc\src\ConvertToAsciiDoc

Get started



Clone or check out code

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 →](#)

Figure 1-2. The Get started window

If you click the “Create a new project” button at the bottom right, it will open the new project dialog. Alternatively, if Visual Studio is already running (or if you’re using an older version that doesn’t show this Get started window), you can use Visual Studio’s File → New → Project menu item, or if you prefer keyboard shortcuts, type Ctrl-Shift-N. Any of these actions opens the “Create a new project” dialog, shown in [Figure 1-3](#).

Create a new project

Search for project templates



Language

Platform

Project type

Recent project templates

Azure Resource Group

C#

ASP.NET Core Web Application

C#

Console App (.NET Core)

C#

MSTest Test Project (.NET Core)

C#

xUnit Test Project (.NET Core)

C#

Blank Solution

NUnit Test Project (.NET Core)

C#



Console App (.NET Core)

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

C#

Linux

macOS

Windows

Console



ASP.NET Core Web Application

Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.

C#

Windows

Linux

macOS

Web



WPF App (.NET Framework)

Windows Presentation Foundation client application

C#

Windows

Desktop



Class Library (.NET Standard)

A project for creating a class library that targets .NET Standard.

C#

Android

iOS

Linux

macOS

Windows

Library



Azure Functions

A template to create an Azure Function project.

C#

Azure

Cloud



Empty Project

Start from scratch with C++ for Windows. Provides no starting files.

Back

Next

Figure 1-3. The Create a new project dialog

This window offers a list of application types. The exact set will depend on what edition of Visual Studio you have installed, and also which development workloads you chose when you installed Visual Studio. As long as you have installed at least one of the workloads that includes C# you should see the option to create a Console App (.NET Core). If you select this and click Next, you will see the dialog shown in [Figure 1-4](#).




Configure your new project

Console App (.NET Core) C# Linux macOS Windows Console

Project name

Location

...

Solution name 

☐ Place solution and project in the same directory

Back

Create

Figure 1-4. The Configure your new project dialog

This lets you choose a name for new for your new project, and also for its containing solution (which will have the same name by default, but you can change this). You can also choose the location on disk for the project. The Project name field affects three things. It controls the name of the `.csproj` file on disk. It also determines the filename of the compiled output. Finally, it sets the default namespace for newly created code, which I'll explain when I show the code. (You can change any of these later if you wish.)

Visual Studio offers a “Place solution and project in the same directory” checkbox that lets you decide how the associated solution is created. If you check it, the project and solution will have the same name and will live in the same folder on disk. But if you plan to add multiple projects to your new solution, you will typically want the solution to be in its own folder, with each project stored in a subfolder. If you leave this checkbox unchecked, Visual Studio will set things up that way, and it also enables the “Solution name” text box so you can give the solution a different name from the first project if necessary. I'm intending to add a unit test project to the solution as well as the program, so I've left the checkbox unchecked. I've set the project name to HelloWorld, and Visual Studio has set the solution name to match, which I'm happy with here. Clicking Create creates my new C# project. So I currently have a solution with a single project in it.

Adding a Project to an Existing Solution

To add a unit test project to the solution, I can go to the Solution Explorer panel, right-click on the solution node (the one at the very top), and choose Add → New Project. This shows a dialog almost identical to the one

in [Figure 1-3](#), but with the title showing “Add a new project” instead. I want to add a test project. I could scroll through the list of project types, but there are faster ways. I could type “Test” into the search box at the top of the dialog. Or, I could click on the “Project type” button at the top right, and select Test from its dropdown. Either approach will show several different test project types. If you see ones for languages other than C#, you can click the Language button next to the search box to filter down to just C#. Even then you’ll see a few, because Visual Studio supports several different test frameworks. I’ll choose MSTest Test Project (.NET Core).

Clicking Next shows the “Configure your new project” dialog again. This new project will contain tests for my HelloWorld project, so I’ll call it HelloWorld.Tests. (Nothing demands that naming convention, by the way—I could have called it anything.) When I click OK, Visual Studio creates a second project, and both are now listed in Solution Explorer, which will look similar to [Figure 1-1](#).

The purpose of this test project will be to ensure that the main project does what it’s supposed to. I happen to prefer the style of development where you write your tests before you write the code being tested, so we’ll start with the test project. To be able to do its job, my test project will need access to the code in the HelloWorld project. Visual Studio has no way of guessing which projects in a solution may depend on which other projects. While there are only two here, even if it were capable of guessing, it would most likely guess wrong, because HelloWorld will produce an *.exe* file, while unit test projects happen to produce a *.dll*. The most obvious guess would be that the *.exe* would depend on the *.dll*, but here we have the somewhat unusual requirement that our library (which is actually a test project) depends on the code in our application.

Referencing One Project from Another

To tell Visual Studio about the relationship between these two projects, I right-click on the HelloWorld.Test project's Dependencies node in Solution Explorer and select the Add Reference menu item. This shows the Reference Manager dialog, which you can see in [Figure 1-5](#). On the left, you choose the sort of reference you want—in this case, I'm setting up a reference to another project in the same solution, so I have expanded the Solution section and selected Projects. This lists all the other projects in the middle, and there's just one in this case, so I check the HelloWorld item and click OK.

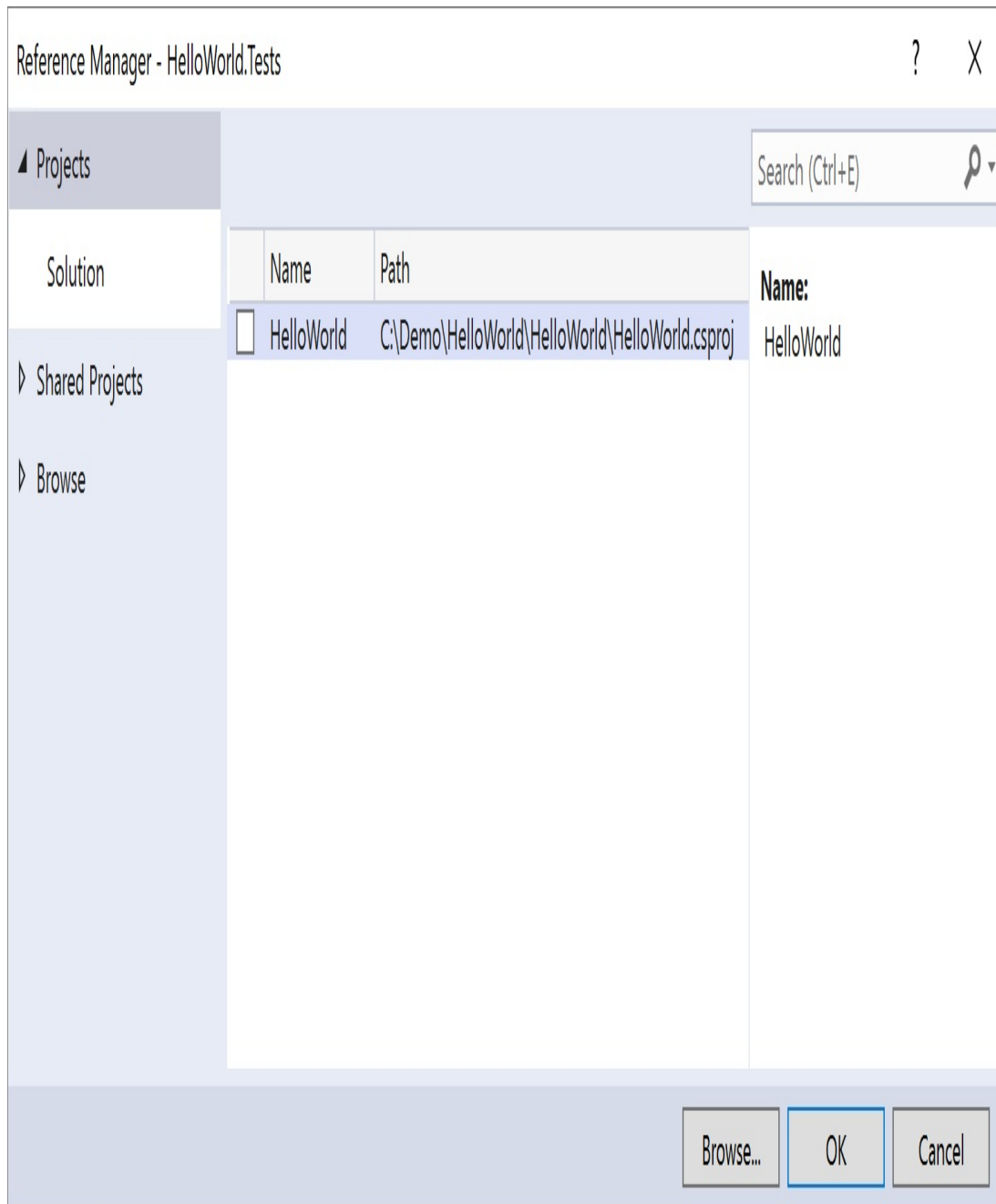


Figure 1-5. The Reference Manager dialog

Referencing External Libraries

Extensive though the .NET class library may be, it does not cover all eventualities. There are hundreds of useful libraries available for .NET, many of them free. In fact, Microsoft is increasingly shipping more and

more libraries separately from .NET. Visual Studio supports adding references using a system called NuGet. In, fact the example is already using it—although we chose Microsoft’s own “MSTest” test framework, that’s not built into .NET. (You generally don’t need unit testing services at runtime, so there’s no real need to build them into the class library that ships with the platform.) If you expand the Dependencies node for the HelloWorld.Tests project in Solution Explorer, and then expand the NuGet child node, you’ll see various NuGet packages as [Figure 1-6](#) shows.

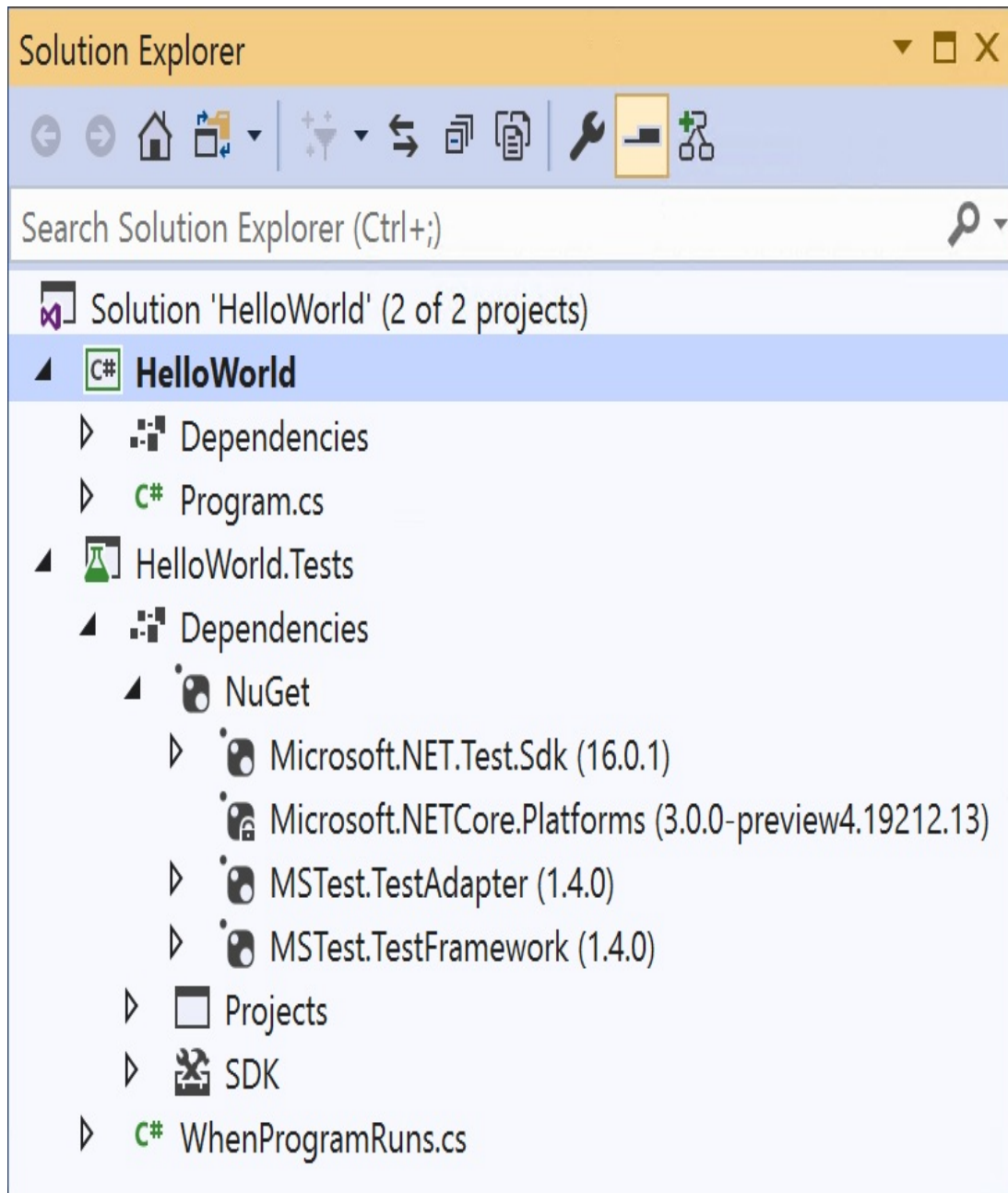


Figure 1-6. NuGet references

You can see three test-related packages, all added for us as part of Visual Studio's test project template. NuGet is a package-based system, so rather than adding a reference to a single DLL, you add a reference to a package that may contain multiple DLLs, and any other files that may be required to use the library.

Microsoft runs a public repository of packages on the <http://nuget.org> website. This hosts copies of all of the libraries that Microsoft does not include directly in the .NET class library, but which they nonetheless fully support. (The testing framework here is one example. The ASP.NET Core web framework is another.) This central NuGet repository is not just for Microsoft. Anyone can make packages available on this site, so this is where you will find the vast majority of free .NET libraries.

Visual Studio can search in the main NuGet repository. If you right-click on a project, or on its Dependencies node, and select Manage NuGet Packages, it will open the NuGet Package Manager window, shown in [Figure 1-7](#). On the left is a list of packages from the NuGet repository. If you select Installed at the top, it will show you information about the packages you are already using. If you click Browse it shows popular available packages by default, but it provides a textbox with which you can search for specific libraries.

Browse Installed Updates

NuGet Package Manager: HelloWorld.Tests

Search (Ctrl+L)



Include prerelease

Package source: nuget.org



Microsoft.NET.Test.Sdk

by Microsoft

v16.0.1



The MSbuild targets and properties for building .NET test projects.



Microsoft.NETCore.Platforms

by Microsoft

v3.0.0-preview4.19212.13

Prerelease

Provides runtime information required to resolve target framework, platform, and run...



MSTest.TestAdapter

by Microsoft

v1.4.0



The adapter to discover and execute MSTest Framework based tests.



MSTest.TestFramework

by Microsoft

v1.4.0



This is MSTest V2, the evolution of Microsoft's Test Framework.

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.



Do not show this again



MSTest.TestFramework



Installed: 1.4.0

Uninstall

Version: 1.4.0

Update



Options

Description

This is MSTest V2, the evolution of Microsoft's Test Framework.

Supported platforms:

- .NET 4.5.0+
- .NET Core 1.0+ (Universal Windows Apps 10+, DNX Core 5+)
- ASP.NET Core 1.0+

To discover and execute tests install MSTest.TestAdapter.

To discover and execute tests for project.json based projects install dotnet-test-mstest.

Figure 1-7. NuGet Package Manager

It is also possible to host your own NuGet repositories. For example, many companies run repositories behind their firewalls to make internally developed packages available to other employees, without having to make them publicly available. The <https://myget.org> site specializes in online hosting, and package hosting is a feature of Microsoft's Azure DevOps and also GitHub. You can configure NuGet to search any number of repositories in addition to the main public one.

One very important feature of NuGet packages is that they can specify dependencies on other packages. For example, if you use the `Microsoft.AspNetCore` package, which adds the ASP.NET Core web service framework to your project, the package declares that it depends on various other packages. For example, it depends on the `Microsoft.Extensions.Logging` package. Because packages describe their dependencies, Visual Studio can automatically fetch all of the packages you require.

Writing a Unit Test

Now I need to write a test. Visual Studio has provided me with a test class to get me started, in a file called *UnitTest1.cs*. I want to pick a more informative name. There are various schools of thought as to how you should structure your unit tests. Some developers advocate one test class for each class you wish to test, but I like the style where you write a class for each *scenario* in which you want to test a particular class, with one method for each of the things that should be true about your code in that scenario. As you've probably guessed from the project names I've chosen, my program will only have one behavior: it will display a "Hello, world!" message when it runs. So I'll rename the *UnitTest1.cs* source file to

WhenProgramRuns.cs. This test should verify that the program prints out the required message when it runs. The test itself is very simple, but unfortunately, getting to the point where we can run this particular test is a bit more involved. Example 1-1 shows the whole source file; the test is near the end, in bold.

Example 1-1. A unit test class for our first program

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HelloWorld.Tests
{
    [TestClass]
    public class WhenProgramRuns
    {
        private string _consoleOutput;

        [TestInitialize]
        public void Initialize()
        {
            var w = new System.IO.StringWriter();
            Console.SetOut(w);

            Program.Main(new string[0]);

            _consoleOutput =
w.GetStringBuilder().ToString().Trim();
        }

        [TestMethod]
        public void SaysHelloWorld()
        {
            Assert.AreEqual("Hello, world!", _consoleOutput);
        }
    }
}
```

I will explain each of the features in this file once I've shown the program itself. For now, the most interesting part of this example is the

`SaysHelloWorld` method, which defines some behavior we want our program to have. The test states that the program's output should be the message "Hello, world!" If it's not, this test will report a failure. The test itself is pleasingly simple, but the code that sets things up for the test is a little awkward. The problem here is that the obligatory first example that all programming books are required by law to show isn't very amenable to unit testing of individual classes or methods, because you can't really test anything less than the whole program. We want to verify that the program prints out a particular message to the console. In a real application, you'd probably devise some sort of abstraction for output, and your unit tests would provide a fake version of that abstraction for test purposes. But I want my application (which [Example 1-1](#) merely tests) to keep to the spirit of the standard "Hello, world!" example. To avoid overcomplicating the main program, I've made my test intercept console output so that I can check that the program printed what it was supposed to.

There's a second challenge. Normally, a unit test will, by definition, test some isolated and usually small part of the program. But in this case, the program is so simple that there is only one feature of interest, and that feature executes when we run the program. This means my test will need to invoke the program's entry point. I could have done that by launching my HelloWorld program in a whole new process, but capturing its output would have been rather more complex than the in-process interception done by [Example 1-1](#). Instead, I'm just invoking the program's entry point directly. In a C# application, the entry point is usually a method called `Main` defined in a class called `Program`. [Example 1-2](#) shows the relevant line from [Example 1-1](#), passing an empty array to simulate running the program with no command-line arguments.

Example 1-2. Calling a method

```
Program.Main(new string[0]);
```

Unfortunately, there's a problem with that. A program's entry point is typically only accessible to the runtime—it's an implementation detail of your program, and there's not normally any reason to make it publicly accessible. However, I'll make an exception here, because that's where the only code in this example will live. So to get the code to compile, we'll need to make a change to our main program. [Example 1-3](#) shows the relevant code from the *Program.cs* file in the HelloWorld project. (I'll show the whole thing shortly.)

Example 1-3. Making the program entry point accessible

```
public class Program
{
    public static void Main(string[] args)
    {
        ...
    }
}
```

I've added the `public` keyword to the start of two lines to make the code accessible to the test, enabling [Example 1-1](#) to compile. There are other ways I could have achieved this. I could have left the class as it is, made the method `internal`, and then applied the `InternalsVisibleToAttribute` to my program to grant access just to the test suite. But internal protection and assembly-level attributes are topics for later chapters, so I decided to keep it simple for this first example.

WARNING

Microsoft's unit testing framework defines a helper class called `PrivateType`, which provides a way to invoke private methods for test purposes, and I could have used that instead of making the entry point public. However, many consider it bad practice to invoke private methods directly from tests, because a test should have to

verify only the observable behavior of the code under test. Testing specific details of how the code has been structured is rarely helpful. In any case, this type is only available if you target .NET Framework, so it cannot be used in .NET Core, and unless Microsoft adds support for that, this means it will also be unavailable in future versions of .NET.

I'm now ready to run my test. To do this, I open Visual Studio's Unit Test Explorer panel with the Test → Windows → Test Explorer menu item. Next, I build the project with the Build → Build Solution menu. Once I've done that, the Unit Test Explorer shows a list of all the unit tests defined in the solution. It finds my `SayHelloWorld` test, as you can see in [Figure 1-8](#). Clicking on Run All runs the test, which fails because we've only written the test so far—we've not done anything to our main program. You can see the error at the bottom of [Figure 1-8](#). It says it was expecting a "Hello, world!" message, but that actual console output was ever so slightly different. (Not by much, admittedly—Visual Studio did in fact add code to my console application that printed a message. But it has no comma, and my test demands a comma.)

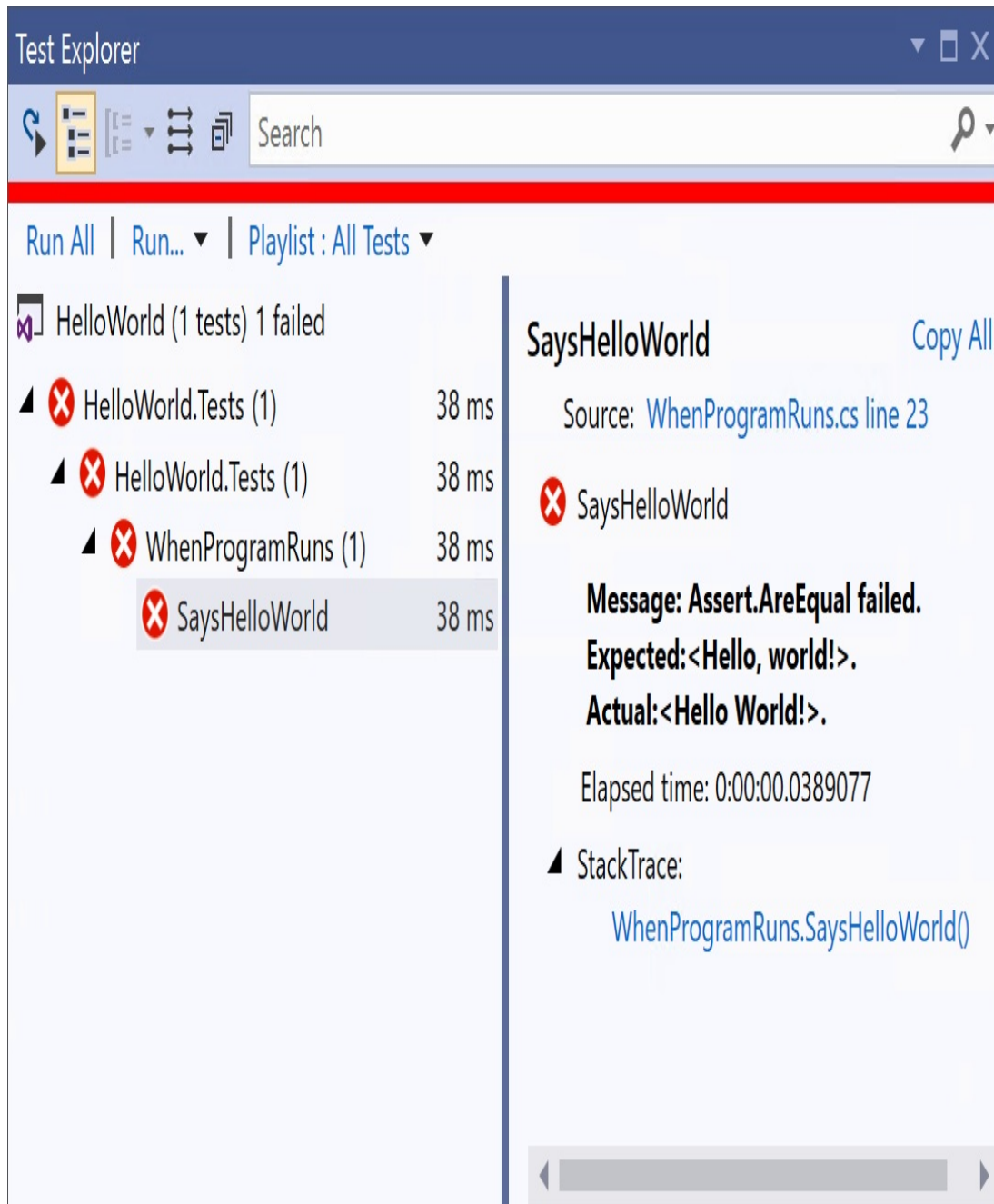


Figure 1-8. Unit Test Explorer

So it's time to look at our HelloWorld program, and to correct the code. When I created the project, Visual Studio generated various files, including *Program.cs*, which contains the program's entry point. [Example 1-4](#) shows this file, including the modifications I made in [Example 1-3](#). I will explain each element in turn, as it provides a useful

introduction to some important elements of C# syntax and structure.

Example 1-4. Program.cs

```
using System;

namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The file begins with a *using directive*. This is optional, but almost all source files contain one or more, and they tell the compiler which *namespaces* we'd like to use, raising the obvious question: what's a namespace?

Namespaces

Namespaces bring order and structure to what would otherwise be a horrible mess. The .NET class library contains a large number of classes, and there are many more classes out there in third-party libraries, not to mention the classes you will write yourself. There are two problems that can occur when dealing with this many named entities. First, it becomes hard to guarantee uniqueness unless everything either has a very long name, or the names include sections of random gibberish. Second, it can become challenging to discover the API you need; unless you know or can guess the right name, it's difficult to find what you need from an unstructured list of many thousands of things. Namespaces solve both of these problems.

Most .NET types are defined in a namespace. Microsoft-supplied types have distinctive namespaces. When the types are part of .NET, the containing namespaces start with `System`, and when they're part of some Microsoft technology that is not a core part of .NET, they usually begin with `Microsoft`. Libraries from other vendors tend to start with the company name, while open source libraries often use their project name. You are not forced to put your own types into namespaces, but it's recommended that you do. C# does not treat `System` as a special namespace, so nothing's stopping you from using that for your own types, but it's a bad idea, because it will tend to confuse other developers. You should pick something more distinctive for your own code, such as your company or project name.

The namespace usually gives a clue as to the purpose of the type. For example, all the types that relate to file handling can be found in the `System.IO` namespace, while those concerned with networking are under `System.Net`. Namespaces can form a hierarchy. So the framework's `System` namespace doesn't just contain types. It also holds other namespaces, such as `System.Net`, and these often contain yet more namespaces, such as `System.Net.Sockets` and `System.Net.Mail`. These examples show that namespaces act as a sort of description, which can help you navigate the library. If you were looking for regular expression handling, for example, you might look through the available namespaces, and notice the `System.Text` namespace. Looking in there, you'd find a `System.Text.RegularExpressions` namespace, at which point you'd be pretty confident that you were looking in the right place.

Namespaces also provide a way to ensure uniqueness. The namespace in

which a type is defined is part of that type's full name. This lets libraries use short, simple names for things. For example, the regular expression API includes a `Capture` class that represents the results from a regular expression capture. If you are working on software that deals with images, the term *capture* is more commonly used to mean the acquisition of some image data, and you might feel that `Capture` is the most descriptive name for a class in your own code. It would be annoying to have to pick a different name just because the best one is already taken, particularly if your image acquisition code has no use for regular expressions, meaning that you weren't even planning to use the existing `Capture` type.

But in fact, it's fine. Both types can be called `Capture`, and they will still have different names. The full name of the regular expression `Capture` class is effectively

`System.Text.RegularExpressions.Capture`, and likewise, your class's full name would include its containing namespace (e.g., `SpiffingSoftworks.Imaging.Capture`).

If you really want to, you can write the fully qualified name of a type every time you use it, but most developers don't want to do anything quite so tedious, which is where the `using` directives at the start of [Example 1-4](#) come in. These state the namespaces of the types this source file intends to use. You will normally edit this list to match your file's requirements. In this example, Visual Studio added `using System;` when I created the project. It chooses different sets in different contexts. If you add a class representing a user interface control, for example, Visual Studio would include various UI-related namespaces in the list.

With `using` declarations like these in place, you can just use the short, unqualified name for a class. The line of code that enables my HelloWorld

example to do its job, uses the `System.Console` class, but because of the first `using` directive, I can refer to it as just `Console`. In fact, that's the only class I'll be using, so there's no need to add any other `using` directives in my main program.

WARNING

Earlier, you saw that a project's References describe which libraries it uses. You might think that References are redundant—can't the compiler work out which external libraries we are using from the namespaces? It could if there were a direct correspondence between namespaces and either libraries or packages, but there isn't. There is sometimes an apparent connection—the popular *Newtonsoft.Json.dll* contains classes in the `Newtonsoft.Json` namespace, for example. But there often isn't—the .NET Framework's version of the class library includes *System.Core.dll*, but there is no `System.Core` namespace. So it is necessary to tell Visual Studio which libraries your project depends on, as well as saying which namespaces any particular source file uses. We will look at the nature and structure of library files in more detail later in the book.

Even with namespaces, there's potential for ambiguity. You might use two namespaces that both happen to define a class of the same name. If you want to use that class, then you will need to be explicit, referring to it by its full name. If you need to use such classes a lot in the file, you can still save yourself some typing: you only need to use the full name once because you can define an *alias*. [Example 1-5](#) uses aliases to resolve a clash that I've run into a few times: .NET's user interface framework, the Windows Presentation Foundation (WPF), defines a `Path` class for working with Bézier curves, polygons, and other shapes, but there's also a `Path` class for working with filesystem paths, and you might want to use both types together to produce a graphical representation of the contents of a file. Just adding `using` directives for both namespaces would make the

simple name `Path` ambiguous if unqualified. But as [Example 1-5](#) shows, you can define distinctive aliases for each.

Example 1-5. Resolving ambiguity with aliases

```
using System.IO;
using System.Windows.Shapes;
using IoPath = System.IO.Path;
using WpfPath = System.Windows.Shapes.Path;
```

With these aliases in place, you can use `IoPath` as a synonym for the file-related `Path` class, and `WpfPath` for the graphical one.

Going back to our HelloWorld example, directly after the `using` directives comes a *namespace declaration*. Whereas `using` directives declare which namespaces our code will consume, a namespace declaration states the namespace in which our own code lives. [Example 1-6](#) shows the relevant code from [Example 1-4](#). This is followed by an opening brace (`{`). Everything between this and the closing brace at the end of the file will be in the `HelloWorld` namespace. By the way, you can refer to types in your own namespace without qualification, without needing a `using` directive. That's why the test code in [Example 1-1](#) doesn't have a `using HelloWorld;` directive—it implicitly has access to that namespace because its code is inside a `namespace HelloWorld.Tests` declaration.

Example 1-6. Namespace declaration

```
namespace HelloWorld
```

```
{
```

Visual Studio generates a namespace declaration with the same name as your project. You're not required to keep this—a project can contain any


mixture of namespaces, and you are free to edit the namespace declaration. But if you do want to use something other than the project name consistently throughout your project, you should tell Visual Studio, because it's not just the first file, *Program.cs*, that gets this generated declaration. By default, Visual Studio adds a namespace declaration based on your project name every time you add a new file. You can tell it to use a different namespace for new files by editing the project's properties. If you double-click on the Properties node inside a project in Solution Explorer or right-click on the project node and select Properties, this opens the properties for the project, and if you go to the Application tab, there's a "Default namespace" text box. It will use whatever you put in there for namespace declarations of any new files. (It won't change the existing files, though.)

NESTED NAMESPACES

The .NET class library nests its namespaces, and sometimes quite extensively. The `System` namespace contains numerous important types, but most types are in more specific namespaces such as `System.Net` or `System.Net.Sockets`. Unless you're creating a trivial example, you will typically nest your own namespaces. There are two ways you can do this. You can nest namespace declarations, as [Example 1-7](#) shows.

Example 1-7. Nesting namespace declarations

```
namespace MyApp
{
    namespace Storage
    {
        ...
    }
}
```



Alternatively, you can just specify the full namespace in a single

declaration, as [Example 1-8](#) shows. This is the more commonly used style.

Example 1-8. Nested namespace with a single declaration

```
namespace MyApp.Storage
```

```
{
```

```
    ...
```

```
}
```



Any code you write in a nested namespace will be able to use types not just from that namespace, but also from its containing namespaces without qualification. Code in [Example 1-7](#) or [Example 1-8](#) would not need explicit qualification or `using` directives to use types either in the `MyApp.Storage` namespace or the `MyApp` namespace.

When you define nested namespaces, the convention is to create a matching folder hierarchy. If you create a project called `MyApp`, as you've seen, by default Visual Studio will put new classes in the `MyApp` namespace when you add them to the project. But if you create a new folder in the project (which you can do in Solution Explorer) called, say, *Storage*, Visual Studio will put any new classes you create in that folder into the `MyApp.Storage` namespace. Again, you're not required to keep this—Visual Studio just adds a namespace declaration when creating the file, and you're free to change it. The compiler does not care if the namespace does not match your folder hierarchy. But since the convention is supported by Visual Studio, life will be easier if you follow it.

Classes

Inside the namespace declaration, my *Program.cs* file defines a *class*.

[Example 1-9](#) shows this part of the file (which includes the `public` keywords I added earlier). The `class` keyword is followed by the name,

and of course the full name of the type is effectively `HelloWorld.Program`, because this code is inside the namespace declaration. As you can see, C# uses braces (`{}`) to delimit all sorts of things—we already saw this for namespaces, and here you can see the same thing with the class as well as the method it contains.

Example 1-9. A class with a method

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Classes are C#'s mechanism for defining entities that combine state and behavior, a common object-oriented idiom. But, as it happens, this class contains nothing more than a single method. C# does not support global methods—all code has to be written as a member of some type. So this particular class isn't very interesting—its only job is to act as the container for the program's entry point. We'll see some more interesting uses for classes in later chapters.

Program Entry Point

By default, the C# compiler will look for a method called `Main` and use that as the entry point automatically. If you really want to, you can tell the compiler to use a different method, but most programs stick with the convention. Whether you designate the entry point by configuration or convention, the method has to meet certain requirements, all of which are evident in [Example 1-9](#).

The program entry point must be a *static method*, meaning that it is not necessary to create an instance of the containing type (`Program`, in this case) in order to invoke the method. It is not required to return anything, as signified by the `void` keyword here, although if you wish you can return `int` instead, which allows the program to return an exit code that the operating system will report when the program terminates. And the method must either take no arguments at all (which would be denoted by an empty pair of parentheses after the method name) or, as in [Example 1-9](#), it can accept a single argument: an array of text strings containing the command-line arguments.

NOTE

Some C-family languages include the filename of the program itself as the first argument, on the grounds that it's part of what the user typed at the command prompt. C# does not follow this convention. If the program is launched without arguments, the array's length will be 0.

The method declaration is followed by the method body, which in this case contains code that is very nearly what we want. We've now looked at everything that Visual Studio generated for us in this file, so all that remains is to modify the code inside the braces delimiting the method body. Remember, our test is failing because our program fails to meet its one requirement: to print out a certain message to the console. This requires the single line of code shown in [Example 1-10](#), inside the method body. This is almost exactly what's already there, it just features an extra comma.

Example 1-10. Printing a message

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



With this in place, if I run the tests again, the Unit Test Explorer shows a checkmark by my test and reports that all tests have passed. So apparently the code is working. And we can verify that informally by running the program. You can do that from Visual Studio's Debug menu. The Start Debugging option runs the program in the debugger. If you run the program this way (which you can also do with the F5 keyboard shortcut), a console window will open, and you'll see it display the traditional message.



Unit Tests

Now that the program is working, I want to go back to the first code I wrote, the test, because that file illustrates some C# features that the main program does not. If you go back to [Example 1-1](#), it starts in a pretty similar way to the main program: we have some `using` directives and then a namespace declaration, the namespace being `HelloWorld.Tests` this time, matching the test project name. But the class looks different. [Example 1-11](#) shows the relevant part of [Example 1-1](#).

Example 1-11. Test class with attribute

```
[TestClass]
public class WhenProgramRuns
{

```



Immediately before the class declaration is the text `[TestClass]`. This is an *attribute*. Attributes are annotations you can apply to classes, methods, and other features of the code. Most of them do nothing on their own—the compiler records the fact that the attribute is present in the compiled output, but that is all. Attributes are useful only when something goes looking for them, so they tend to be used by frameworks. In this case,

I'm using Microsoft's unit testing framework, and it goes looking for classes annotated with this `TestClass` attribute. It will ignore classes that do not have this annotation. Attributes are typically specific to a particular framework, and you can define your own, as we'll see later on.

The two methods in the class are also annotated with attributes.

[Example 1-12](#) shows the relevant excerpts from [Example 1-1](#). The test runner will execute any methods marked with `[TestInitialize]` once for every test the class contains, and will do so before running the actual test method itself. And, as you have no doubt guessed, the `[TestMethod]` attribute tells the test runner which methods represent tests.

Example 1-12. Annotated methods

```
[TestInitialize]
public void Initialize()
...

[TestMethod]
public void SaysHelloWorld()
...
```

There's one more feature in [Example 1-1](#): the class contents begin with a field, shown again in [Example 1-13](#). Fields hold data. In this case, the `Initialize` method stores the console output that it captures while the program runs in this `_consoleOutput` field, where it is available for test methods to inspect. This particular field has been marked as `private`, indicating that it is for this particular class's own use. The C# compiler will permit only code that lives in the same class to access this data.

Example 1-13. A field

```
private string _consoleOutput;
```

And with that, we've examined every element of a program and the test project that verifies that it works as intended.

Summary

You've now seen the basic structure of C# programs. I created a solution containing two projects, one for tests and one for the program itself. This was a simple example, so each project had only one source file of interest. Both were of similar structure. Each began with `using` directives indicating which types the file uses. A namespace declaration stated the namespace that the file populates, and this contained a class containing one or more methods or other members such as fields.

We will look at types and their members in much more detail in later chapters, but first, [Chapter 2](#) will deal with the code that lives inside methods, where we express what we want our programs to do.

¹ This was true of Microsoft's previous cross-platform C# offering. In 2008, they shipped Silverlight 2.0, which enabled C# to run inside browsers on Windows and macOS. It fought a losing battle against the improving capabilities and universal reach of HTML5 and JavaScript, but its closed source nature may not have helped its cause.

² Microsoft's first set of .NET extensions for C++ resembled ordinary C++ more closely. In the end, it turned out to be less confusing to use a distinct syntax for something that is quite different from ordinary C++, so Microsoft deprecated the first system (Managed C++) in favor of the newer, more distinctive syntax, which is called C++/CLI.

³ Executables typically have an `.exe` file extension in Windows, while libraries use `.dll` (historically short for *dynamic link library*). These are almost identical, the only difference being that an `.exe` file specifies an application entry point. Both file types can export features to be consumed by other components. These are both examples of *assemblies*.

Chapter 2. Basic Coding in C#

All programming languages have to provide certain capabilities. It must be possible to express the calculations and operations that our code should perform. Programs need to be able to make decisions based on their input. Sometimes we will need to perform tasks repeatedly. These fundamental features are the very stuff of programming, and this chapter will show how these things work in C#.

Depending on your background, some of this chapter's content may seem very familiar. C# is said to be from the “C family” of languages. C is a hugely influential programming language, and numerous languages have borrowed much of its syntax. There are direct descendants such as C++ and Objective-C. There are also more distantly related languages, including Java, and JavaScript, and C# itself, that have no compatibility with C, but which still ape many aspects of its syntax. If you are familiar with any of these languages, you will recognize many of the language features we are about to explore.

We saw the basic structure of a program in [Chapter 1](#). In this chapter, I will be looking just at code inside methods. As you've seen, C# requires a certain amount of structure: code is made up of statements that live inside a method, which belongs to a type, which is typically inside a namespace, all inside a file that is part of a project, typically contained by a solution. For clarity, most of the examples in this chapter will show the code of interest in isolation, as in [Example 2-1](#).

Example 2-1. The code, and nothing but the code

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

Unless I say otherwise, this kind of extract is shorthand for showing the code in context inside a suitable program. So [Example 2-1](#) is short for [Example 2-2](#).

Example 2-2. The whole code

```
using System;

namespace Hello
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello, world!");
        }
    }
}
```

Although I'll be introducing fundamental elements of the language in this section, this book is for people who are already familiar with at least one programming language, so I'll be relatively brief with the most ordinary features of the language, and will go into more detail on those aspects that are particular to C#.

Local Variables

The inevitable “Hello, world!” example is missing a vital element: it doesn't really deal with information. Useful programs normally fetch, process, and produce information, so the ability to define and identify information is one of the most important features of a language. Like most languages, C# lets you define *local variables*, which are named elements inside a method that each hold a piece of information.

NOTE

In the C# specification, the term *variable* can refer to local variables, but also to fields in objects, and array elements. This section is concerned entirely with local variables, but it gets tiring to keep reading the *local* prefix. So, from now on in this section, *variable* means a local variable.

C# is a *statically typed* language, which is to say that any element of code that represents or produces information, such as a variable or an expression, has a data type determined at compile time. This is different than *dynamically typed* languages, such as JavaScript, in which types are determined at runtime.¹

The easiest way to see C#'s static typing in action is with simple variable declarations such as the ones in [Example 2-3](#). Each of these starts with the data type—the first two variables are of type `string`, followed by two `int` variables.

Example 2-3. Variable declarations

```
string part1 = "the ultimate question";  
string part2 = "of something";  
int theAnswer = 42;  
int something;
```

The data type is followed immediately by the variable's name. The name must begin with either a letter or an underscore, which can be followed by any combination of letters, decimal digits, and underscores. (At least, those are the options if you stick to ASCII. C# supports Unicode, so if you save your file in UTF-8 or UTF-16 format, anything after the first character in an identifier can be any of the characters described in the “Identifier and Pattern Syntax” annex of the Unicode specification. This

includes various accents, diacritics, and numerous somewhat obscure punctuation marks, but only characters intended for use *within* words—characters that Unicode identifies as being intended for *separating* words cannot be used.) These same rules determine what constitutes a legal identifier for any user-defined entity in C#, such as a class or a method.

Example 2-3 shows that there are a couple of forms of variable declaration. The first three variables include an *initializer*, which provides the variable's initial value, but as the final variable shows, this is optional. That's because you can assign new values into variables at any point.

Example 2-4 continues on from Example 2-3, and shows that you can assign a new value into a variable regardless of whether it had an initial value.

Example 2-4. Assigning values to previously declared variables

```
part2 = " of life, the universe, and everything";  
something = 123;
```

Because variables have a static type, the compiler will reject attempts to assign the wrong kind of data. So if we were to follow on from Example 2-3 with the code in Example 2-5, the compiler would complain. It knows that the variable called `theAnswer` has a type of `int`, which is a numeric type, so it will report an error if we attempt to assign a text string into it.

Example 2-5. An error: the wrong type

```
theAnswer = "The compiler will reject this";
```

You'd be allowed to do this in a dynamic language such as JavaScript, because in such a language, a variable doesn't have its own type—all that matters is the type of the value it contains, and that can change as the code

runs. It's possible to do something similar in C# by declaring a variable with type `object` (which I'll describe later in the [“Intrinsic Data Types”](#) section) or `dynamic`. However, the most common practice in C# is for variables to have a more specific type.

NOTE

The static type doesn't always provide a complete picture, thanks to inheritance. I'll be discussing this later on, but for now, it's enough to know that some types are open to extension through inheritance, and if a variable uses such a type, then it's possible for it to refer to some object of a type derived from the variable's static type. Interfaces provide a similar kind of flexibility. However, the static type always determines what operations you are allowed to perform on the variable. If you want to use additional members specific to some derived type, you won't be able to do so through a variable of the base type.

You don't have to state the variable type explicitly. You can let the compiler work it out for you by using the keyword `var` in place of the data type. [Example 2-6](#) shows the first three variable declarations from [Example 2-3](#), but using `var` instead of explicit data types.

Example 2-6. Implicit variable types with the `var` keyword

```
var part1 = "the ultimate question";  
var part2 = "of something";  
var theAnswer = 40 + 2;
```

This code often misleads people who know some JavaScript, because that also has a `var` keyword that you can use in a similar-looking way. But `var` does not work the same way in C# as in JavaScript: these variables are still all statically typed. All that's changed is that we haven't said what the type is—we're letting the compiler deduce it for us. It looks at the initializers, and can see that the first two variables are strings while the

third is an integer. (That’s why I left out the fourth variable from [Example 2-3](#), `something`. That doesn’t have an initializer, so the compiler would have no way of inferring its type. If you try to use the `var` keyword without an initializer, you’ll get a compiler error.)

You can demonstrate that variables declared with `var` are statically typed by attempting to assign something of a different type into them. We could repeat the same thing we tried in [Example 2-5](#), but this time with a `var`-style variable. [Example 2-7](#) does this, and it will produce exactly the same compiler error, because it’s the same mistake—we’re trying to assign a text string into a variable of an incompatible type. That variable, `theAnswer`, has a type of `int` here, even though we didn’t say so explicitly.

Example 2-7. An error: the wrong type (again)

```
var theAnswer = 42;  
theAnswer = "The compiler will reject this";
```



Opinion is divided on how and when to use the `var` keyword, as the sidebar [“To var, or Not to var?”](#) describes.

TO VAR, OR NOT TO VAR?

A `var`-style variable declaration is exactly equivalent to a variable declaration with an explicit type, which raises a question: which should you use? In a sense, it doesn’t matter, because they are equivalent. However, if you like your code to be consistent, you’ll want to pick one style and stick to it. Not everyone agrees on which is the “best” style.

Some developers see the extra text required for explicit variable types as unproductive “ceremony,” preferring the more succinct `var` keyword. Let the compiler deduce the type for you, instead of doing the work yourself, or so the argument goes. It also reduces visual clutter in the code.

I take a different view, because I spend more time reading code than writing it—debugging, code review, refactoring, and enhancements seem to dominate. Anything that makes those activities easier is worth the frankly minimal time it takes to write the type names explicitly. Code that uses `var` everywhere slows you down, because you have to work out what the type really is in order to understand the code. Although `var` saved you some work when you wrote the code, that gain is quickly wiped out by the additional thought

required every time you go back and look at the code. So unless you're the sort of developer who only ever writes new code, leaving others to clean up after you, the only benefit the "var everywhere" philosophy really offers is that it can look neater.

You can even use explicit types and still get the compiler to do the work: in Visual Studio, you can start with the keystroke-friendly `var`, but if you then press `Ctrl+.`, the Quick Actions menu opens, offering to replace it with the explicit type for you. (Visual Studio uses the C# compiler's API to discover the variable's type.)

That said, there are some situations in which I will use `var`. One is to avoid writing the name of the type twice as in this example:

```
List<int> numbers = new List<int>();
```

We can drop the first `List<int>` without making this harder to read, because it's still right there in the initializer. There are similar examples involving casts and generic methods. As long as the type name appears explicitly in the variable declaration, there is no downside to using `var` to avoid writing the type twice.

I also use `var` where it is necessary. As we will see in later chapters, C# supports *anonymous types*, and as the name suggests, it's not possible to write the name of such a type. In these situations, you may be compelled to use `var`. (In fact, the `var` keyword was introduced to C# only when anonymous types were added.)

One last thing worth knowing about declarations is that you can declare, and optionally initialize, multiple variables in a single line. If you want multiple variables of the same type, this may reduce clutter in your code. [Example 2-8](#) declares three variables of the same type in a single declaration.

Example 2-8. Multiple variables in a single declaration

```
double a = 1, b = 2.5, c = -3;
```

Regardless of how you declare it, a variable holds some piece of information of a particular type, and the compiler prevents us from putting data of an incompatible type into that variable. Of course, variables are useful only because we can refer back to them later in our code.

[Example 2-9](#) starts with the variable declarations we saw in earlier examples, then goes on to use the values of those variables to initialize

some more variables, and then prints out the results.

Example 2-9. Using variables

```
string part1 = "the ultimate question";
string part2 = "of something";
int theAnswer = 42;

part2 = "of life, the universe, and everything";

string questionText = "What is the answer to " + part1 + ", " +
part2 + "?";
string answerText = "The answer to " + part1 + ", " +
                    part2 + ", is: " + theAnswer;

Console.WriteLine(questionText);
Console.WriteLine(answerText);
```

By the way, this code relies on the fact that C# defines a couple of meanings for the + operator when it's used with strings. First, when you “add” two strings together, it concatenates them. Second, when you “add” something other than a string to the end of a string (as the initializer for `answerText` does—it adds `theAnswer`, which is a number), C# generates code that converts the value to a string before appending it. So [Example 2-9](#) produces this output:

```
What is the answer to the ultimate question, of life, the
universe, and everythi
ng?
The answer to the ultimate question, of life, the universe, and
everything, is:
42
```

NOTE

In this book, text longer than 80 characters is wrapped across multiple lines to fit. If you try these examples, they will look different if your console windows are configured for a different width.

When you use a variable, its value is whatever you last assigned into it. If you attempt to use a variable before you have assigned a value, as [Example 2-10](#) does, the C# compiler will report an error.

Example 2-10. Error: using an unassigned variable

```
int willNotWork;  
Console.WriteLine(willNotWork);
```

Compiling that produces this error for the second line:

```
error CS0165: Use of unassigned local variable 'willNotWork'
```

The compiler uses a slightly pessimistic system (which it calls the *definite assignment* rules) for determining whether a variable has a value yet. It's not possible to create an algorithm that can determine such things for certain in every possible situation.² Since the compiler has to err on the side of caution, there are some situations in which the variable will have a value by the time the offending code runs, and yet the compiler still complains. The solution is to write an initializer, so that the variable always contains something, perhaps using 0 for numeric values, and `false` for Boolean variables. Later in the book, I'll introduce reference types, and as the name suggests, a variable of such a type can hold a reference to an instance of the type. If you need to initialize such a variable before you've got something for it to refer to, you can use the keyword `null`, a special value signifying a reference to nothing.

The definite assignment rules determine the parts of your code in which the compiler considers a variable to contain a valid value, and will

therefore let you read from it. Writing into a variable is less restricted, but of course, any given variable is accessible only from certain parts of the code. Let's look at the rules that govern this.

Scope

A variable's *scope* is the range of code in which you can refer to that variable by its name. Variables are not the only things with scope. Methods, properties, types, and, in fact, anything with a name all have scope. These require a slightly broader definition of scope: it's the region in which you can refer to the entity by its name without needing additional qualification. When I write `Console.WriteLine` I am referring to the method by its name (`WriteLine`), but I need to qualify it with a class name (`Console`), because the method is not in scope. But with a local variable, scope is absolute: either it's accessible without qualification, or it's not accessible at all.

Broadly speaking, a variable's scope starts at its declaration, and finishes at the end of its containing *block*. A block is a region of code delimited by a pair of braces (`{}`). A method body is a block, so a variable defined in one method is not visible in another method, because it is out of scope. If you attempt to compile [Example 2-11](#), you'll get an error complaining that `The name 'thisWillNotWork' does not exist in the current context`.

Example 2-11. Error: out of scope

```
static void SomeMethod()  
{  
    int thisWillNotWork = 42;  
}  
  
static void AnUncompilableMethod()
```

```
{  
    Console.WriteLine(thisWillNotWork);  
}
```

Methods often contain nested blocks, particularly when you work with the loop and flow control constructs we'll be looking at later in this chapter. At the point where a nested block starts, everything that is in scope in the outer block continues to be in scope inside that nested block. [Example 2-12](#) declares a variable called `someValue`, and then introduces a nested block as part of an `if` statement. The code inside this block is able to access that variable declared in the containing block.

Example 2-12. Variable declared outside block, used within block

```
int someValue = GetValue();  
if (someValue > 100)  
{  
    Console.WriteLine(someValue);  
}
```

The converse is not true. If you declare a variable in a nested block, its scope does not extend outside of that block. So [Example 2-13](#) will fail to compile, because the `willNotWork` variable is only in scope within the nested block. The final line of code will produce a compiler error because it's trying to use that variable outside of that block.

Example 2-13. Error: trying to use a variable not in scope

```
int someValue = GetValue();  
if (someValue > 100)  
{  
    int willNotWork = someValue - 100;  
}  
Console.WriteLine(willNotWork);
```

This probably all seems fairly straightforward, but things get a bit more complex when it comes to potential naming collisions. C# sometimes catches people by surprise here.

VARIABLE NAME AMBIGUITY

Consider the code in [Example 2-14](#). This declares a variable called `anotherValue` inside a nested block. As you know, that variable is only in scope to the end of that nested block. After that block ends, we try to declare another variable with the same name.

Example 2-14. Error: surprising name collision

```
int someValue = GetValue();  
if (someValue > 100)  
{  
    int anotherValue = someValue - 100;  
    Console.WriteLine(anotherValue);  
}  
  
int anotherValue = 123;
```

This causes a compiler error on the first of the lines to declare `anotherValue`:

```
error CS0136: A local or parameter named 'anotherValue' cannot  
be declared in  
this scope because that name is used in an enclosing local  
scope to define a  
local or parameter
```

This seems odd. At the final line, the supposedly conflicting earlier declaration is not in scope, because we're outside of the nested block in which it was declared. Furthermore, the second declaration is not in scope within that nested block, because the declaration comes after the block. The scopes do not overlap, but despite this, we've fallen foul of C#'s rules

for avoiding name conflicts. To see why this example fails, we first need to look at a less surprising example.

C# tries to prevent ambiguity by disallowing code where one name might refer to more than one thing. [Example 2-15](#) shows the sort of problem it aims to avoid. Here we've got a variable called `errorCount`, and the code starts to modify this as it progresses, but partway through, it introduces a new variable in a nested block, also called `errorCount`. It is possible to imagine a language that allowed this—you could have a rule that says that when multiple items of the same name are in scope, you just pick the one whose declaration happened last.

Example 2-15. Error: hiding a variable

```
int errorCount = 0;
if (problem1)
{
    errorCount += 1;

    if (problem2)
    {
        errorCount += 1;
    }

    // Imagine that in a real program there was a big
    // chunk of code here before the following lines.

    int errorCount = GetErrors(); // Compiler error
    if (problem3)
    {
        errorCount += 1;
    }
}
```

C# chooses not to allow this, because code that did this would be easy to misunderstand. This is an artificially short method because it's a fake

example in a book, making it easy to see the duplicate names. If the code were a bit longer, it would be very easy to miss the nested variable declaration, and not to realize that `errorCount` refers to something different at the end of the method than it did earlier on. C# simply disallows this to avoid misunderstanding.

But why does [Example 2-14](#) fail? The scopes of the two variables don't overlap. Well, it turns out that the rule that outlaws [Example 2-15](#) is not based on scopes. It is based on a subtly different concept called a *declaration space*. A declaration space is a region of code in which a single name must not refer to two different entities. Each method introduces a declaration space for variables. Nested blocks also introduce declaration spaces, and it is illegal for a nested declaration space to declare a variable with the same name as one in its parent's declaration space. And that's the rule we've fallen foul of here—the outermost declaration space in [Example 2-15](#) contains a variable named `errorCount`, and a nested block's declaration space tries to introduce another variable of the same name.

If that all seems a bit dry, it may be helpful to know *why* there's a whole separate set of rules for name collisions instead of basing it on scopes. The intent of the declaration space rules is that it mostly shouldn't matter where you put the declaration. If you were to move all of the variable declarations in a block to the start of that block—and some organizations have coding standards that mandate this sort of layout—the idea of these rules is that this shouldn't change what the code means. Clearly this wouldn't be possible if [Example 2-15](#) were legal. And this explains why [Example 2-14](#) is illegal. Although the scopes don't overlap, they would if you moved all variable declarations to the top of their containing blocks.

LOCAL VARIABLE INSTANCES

Variables are features of the source code, so each particular variable has a distinct identity: it is declared in exactly one place in the source code, and goes out of scope at exactly one well-defined place. However, that doesn't mean that it corresponds to a single storage location in memory. It is possible for multiple invocations of a single method to be in progress simultaneously, through recursion, multithreading, or asynchronous execution.

Each time a method runs, it gets a distinct set of storage locations to hold the local variables' values. This enables multiple threads to execute the same method simultaneously without problems, because each has its own set of local variables. Likewise, in recursive code, each nested call gets its own set of locals that will not interfere with any of its callers. The same goes for multiple concurrent invocations of an asynchronous method. (To be strictly accurate, each execution of a particular *scope* gets its own set of variables. This distinction matters when you use nested methods.) C# reuses storage locations when it can, so it will only allocate new memory for each scope's execution when it really has to, but the effect is as though it allocated new space each time.

Be aware that the C# compiler does not make any particular guarantee about where variables live (except in exceptional cases). They might well live on the stack, but sometimes they don't. When we look at anonymous methods in later chapters, you'll see that variables sometimes need to outlive the method that declares them, because they remain in scope for nested methods that will run as callbacks after the containing method has returned.

By the way, before we move on, be aware that just as variables are not the

only things to have scope, they are also not the only things to which declaration space rules apply. Other language features that we'll be looking at later, including classes, methods, and properties, also have scoping and name uniqueness rules.

Statements and Expressions

Variables give us somewhere to put the information that our code works with, but to do anything with those variables, we will need to write some code. This will mean writing *statements* and *expressions*.

Statements

When we write a C# method, we are writing a sequence of statements. Informally, the statements in a method describe the actions we want the method to perform. Each line in [Example 2-16](#) is a statement. It might be tempting to think of a statement as an instruction to do one thing (e.g., initialize a variable or invoke a method). Or you might take a more lexical view, where anything ending in a semicolon is a statement. (And it's the semicolons that are significant here, not the line breaks, by the way. We could have written this as one long line of code and it would have exactly the same meaning.) However, both descriptions are simplistic, even though they happen to be true for this particular example.

Example 2-16. Some statements

```
int a = 19;  
int b = 23;  
int c;  
c = a + b;  
Console.WriteLine(c);
```



C# recognizes many different kinds of statements. The first three lines of

Example 2-16 are *declaration statements*, statements that declare and optionally initialize a variable. The fourth and fifth lines are *expression statements* (and we'll be looking at expressions shortly). But some statements have more structure than the ones in this example.

When you write a loop, that's an *iteration statement*. When you use the `if` or `switch` mechanisms described later in this chapter to choose between various possible actions, those are *selection statements*. In fact, the C# specification distinguishes between 13 categories of statement. Most fit broadly into the scheme of describing either what the code should do next, or, for features such as loops or conditional statements, describing *how* it should decide what to do next. Statements of that second kind usually contain one or more embedded statements describing the action to perform in a loop, or the action to perform when an `if` statement's condition is met.

There's one special case, though. A block is a kind of statement. This makes statements such as loops more useful than they would otherwise be, because a loop iterates over just a single embedded statement. That statement can be a block, and since a block itself is a sequence of statements (delimited by braces), this is what enables loops to contain more than one statement.

This illustrates why the two simplistic points of view stated earlier—"statements are actions" and "statements are things that end in semicolons"—are wrong. Compare Example 2-16 with Example 2-17. Both do the same thing, because the various actions we've said we want to perform remain exactly the same. However, Example 2-17 contains one extra statement. The first two statements are the same, but they are followed by a third statement, a block, which contains the final three

statements from [Example 2-16](#). The extra statement, the block, doesn't end in a semicolon, nor does it perform any action. It might seem pointless, but it can sometimes be useful to introduce a nested block like this to avoid name ambiguity errors. So statements can be structural, rather than causing anything to happen at runtime.

Example 2-17. A block

```
int a = 19;
int b = 23;
{
    int c;
    c = a + b;
    Console.WriteLine(c);
}
```

While your code will contain a mixture of statement types, it will inevitably end up containing at least a few expression statements. These are, quite simply, statements that consist of a suitable expression, followed by a semicolon. What's a suitable expression? What's an expression, for that matter? I'd better answer that second question before coming back to what constitutes a valid expression for a statement.

Expressions

The official definition of a C# *expression* is rather dry: “a sequence of operators and operands.” Admittedly, language specifications tend to be like that, but in addition to this sort of formal prose, the C# specification contains some very readable informal explanations of the more formally expressed ideas. (For example, it describes statements as the means by which “the actions of a program are expressed” before going on to pin that down with less approachable but more technically precise language.) I'm quoting from the formal definition of an expression at the start of this

paragraph, so we might hope that the informal explanation in the introduction will be more helpful. No such luck: it says that expressions “are constructed from operands and operators.” That’s certainly less precise than the other definition, but it’s no easier to understand. The problem is that there are several kinds of expressions, and they do different jobs, so there isn’t a single, general, informal description.

It’s tempting to describe an expression as some code that produces a value. That’s not true for all expressions, but the majority of expressions you’ll write will fit this description, so I’ll focus on this for now, and I’ll come to the exceptions later.

The simplest expressions with values are *literals*, where we just write the value we want, such as `"Hello, world!"` or `2`. You can also use the name of a variable as an expression. Expressions can involve operators, which describe calculations or other computations to be performed. Operators have some fixed number of inputs, or *operands*. Some take a single operand. For example, you can negate a number by putting a minus sign in front of it. Some take two: the `+` operator lets you form an expression that adds together the results of the two operands on either side of the `+` symbol.

NOTE

Some symbols have different roles depending on the context. The minus sign is not just used for negation. It acts as a two-operand subtraction operator if it appears between two expressions.

In general, operands are also expressions. So, when we write `2 + 2`, that’s an expression that contains two more expressions—the pair of ‘2’

literals on either side of the + symbol. This means that we can write arbitrarily complicated expressions by nesting expressions within expressions within expressions. Example 2-18 exploits this to evaluate the quadratic formula (the standard technique for solving quadratic equations).

Example 2-18. Expressions within expressions

```
double a = 1, b = 2.5, c = -3;  
double x = (-b + Math.Sqrt(b * b - 4 * a * c)) / (2 * a);  
Console.WriteLine(x);
```

Look at the declaration statement on the second line. Its initializer expression's overall structure is a division operation. But that division operator's two operands are also expressions. Its lefthand operand is a *parenthesized expression*, which tells the compiler that I want that whole expression `(-b + Math.Sqrt(b * b - 4 * a * c))` to be the first operand of the division. This subexpression contains an addition, whose lefthand operand is a negation expression whose single operand is the variable `b`. The addition's righthand side takes the square root of another, more complex expression. And the division's lefthand operand is another parenthesized expression, containing a multiplication. Figure 2-1 illustrates the full structure of the expression.

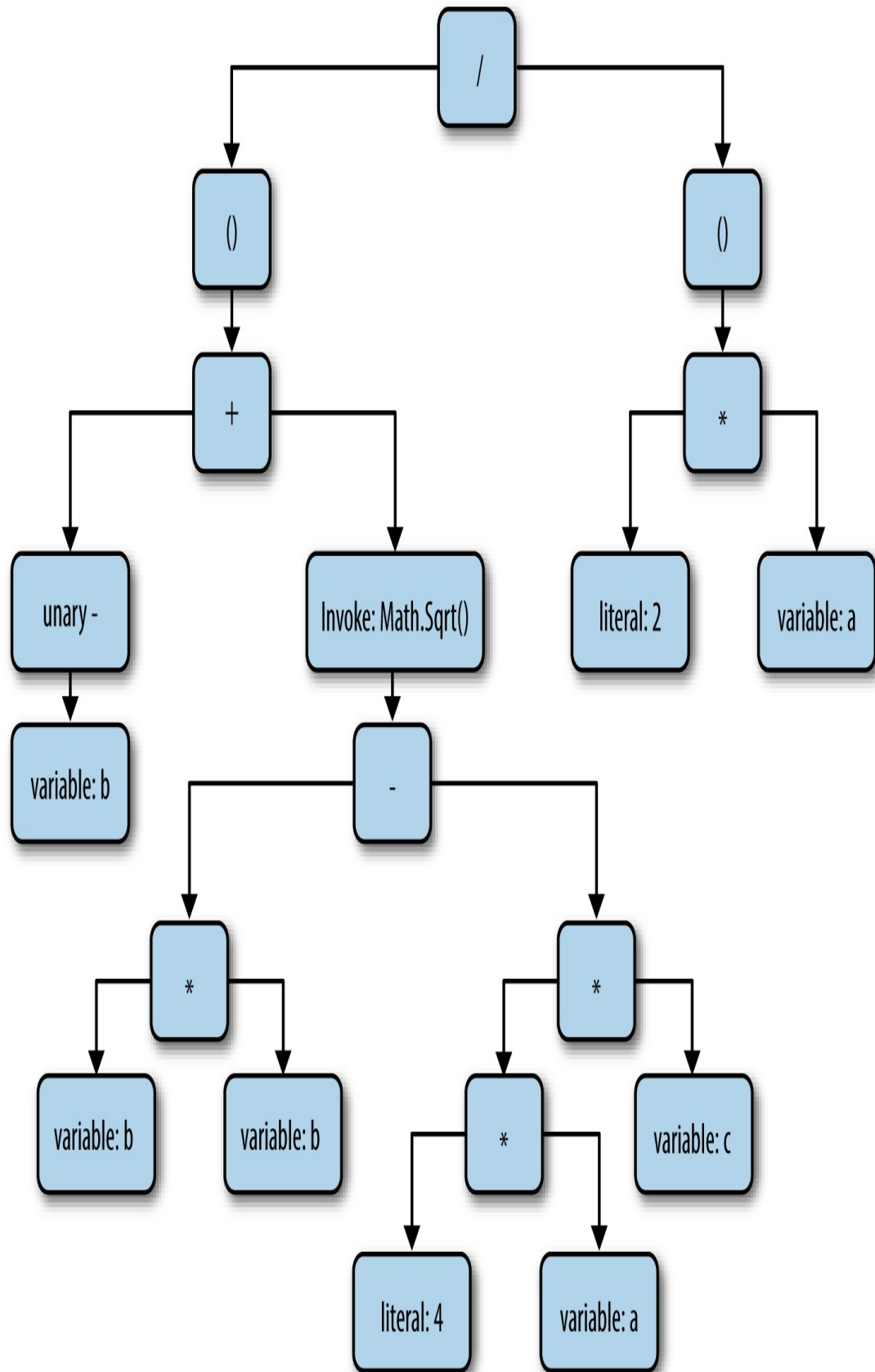


Figure 2-1. The structure of an expression

One important detail of this last example is that method invocations are a kind of expression. The `Math.Sqrt` method used in [Example 2-18](#) is a .NET Framework class library function that calculates the square root of its input and returns the result. What's perhaps more surprising is that invocations of methods that don't return a value, such as `Console.WriteLine`, are also, technically, expressions. And there are a few other constructs that don't produce values but are still considered to be expressions, including a reference to a type (e.g., the `Console` in `Console.WriteLine`) or to a namespace. These sorts of constructs take advantage of a set of common rules (e.g., scoping, how to resolve what a name refers to, etc.). However, all the non-value-producing expressions can be used only in certain specific circumstances. (You can't use one as an operand in another expression, for example.) So although it's not technically correct to define an expression as a piece of code that produces a value, the ones that do are the ones we use when describing the calculations we want our code to perform.

We can now return to the question: what can we put in an expression statement? Roughly speaking, the expression has to do something; it cannot just calculate a value. So although `2 + 2` is a valid expression, you'll get an error if you try to turn it into an expression statement by sticking a semicolon on the end. That expression calculates something but doesn't do anything with the result. To be more precise, you can use the following kinds of expressions as statements: method invocation, assignment, increment, decrement, and new object creation. We'll be looking at increment and decrement later in this chapter, and we'll be looking at objects in later chapters, so that leaves invocation and assignment.

So a method invocation is allowed to be an expression statement. This can involve nested expressions of other kinds, but the whole thing must be a method call. [Example 2-19](#) shows some valid examples. Notice that the C# compiler doesn't check whether the method call really has any lasting effect—the `Math.Sqrt` function is a pure function, in the sense that it does nothing other than returning a value determined entirely by its inputs. So invoking it and then doing nothing with the result doesn't really do anything at all—it's no more of an action than the expression `2 + 2`. But as far as the C# compiler is concerned, any method call is allowed as an expression statement.

Example 2-19. Method invocation expressions as statements

```
Console.WriteLine("Hello, world!");  
Console.WriteLine(12 + 30);  
Console.ReadKey();  
Math.Sqrt(4);
```

It seems inconsistent that C# forbids us from using an addition expression as a statement while allowing `Math.Sqrt`. Both perform a calculation that produces a result, so it makes no sense to use either in this way. Wouldn't it be more consistent if C# allowed only calls to methods that return nothing to be used for expression statements? That would rule out the final line of [Example 2-19](#), which would seem like a good idea because that code does nothing useful, and would be consistent with the fact that `2 + 2` also cannot form an expression statement. Unfortunately, sometimes you want to ignore the return value. [Example 2-19](#) calls `Console.ReadKey()`, which waits for a keypress and returns a value indicating which key was pressed. If my program's behavior depends on which particular key the user pressed, I'll need to inspect the method's return value, but if I just want to wait for any key at all, it's OK to ignore the return value. If C# didn't allow methods with return values to be used

as expression statements, I wouldn't be able to do this. The compiler has no way to distinguish between methods that make for pointless statements because they have no side effects (such as `Math.Sqrt`) and those which might be good candidates (such as `Console.ReadKey`), so it allows any method.

For an expression to be a valid expression statement, it is not enough merely to contain a method invocation. [Example 2-20](#) shows some expressions that call methods and then go on to use those as part of addition expressions. Although these are valid expressions, they're not valid expression statements, so these will cause compiler errors.

Example 2-20. Errors: some expressions that don't work as statements

```
Console.ReadKey().KeyChar + "!";  
Math.Sqrt(4) + 1;
```

Earlier I said that one kind of expression we're allowed to use as a statement is an assignment. It's not obvious that assignments should be expressions, but they are, and they do produce a value: the result of an assignment expression is the value being assigned into the variable. This means it's legal to write code like that in [Example 2-21](#). The second line here uses an assignment expression as an argument for a method invocation, which prints out the value of that expression. The first two `WriteLine` calls both display 123.

Example 2-21. Assignments are expressions

```
int number;  
Console.WriteLine(number = 123);  
Console.WriteLine(number);  
  
int x, y;  
x = y = 0;  
Console.WriteLine(x);  
Console.WriteLine(y);
```



The second part of this example assigns one value into two variables in a single step by exploiting the fact that assignments are expressions—it assigns the value of the `y = 0` expression (which evaluates to `0`) into `x`.

This shows that evaluating an expression can do more than just producing a value. Some expressions have side effects. We've just seen that an assignment is an expression, and of course it has the effect of changing what's in a variable. Method calls are expressions too, and although you can write pure functions that do nothing besides calculating their result from their input, like `Math.Sqrt`, many methods do something with lasting effects, such as printing data to the screen, updating a database, or launching a missile. This means that we might care about the order in which the operands of an expression get evaluated.

An expression's structure imposes some constraints on the order in which operators do their work. For example, I can use parentheses to enforce ordering. The expression `10 + (8 / 2)` has the value 14, while the expression `(10 + 8) / 2` has the value 9, even though both have exactly the same literal operands and arithmetic operators. The parentheses here determine whether the division is performed before or after the subtraction.³

However, while the structure of an expression imposes some ordering constraints, it still leaves some latitude: although both the operands of an addition need to be evaluated before they can be added, the addition operator doesn't care which operand we evaluate first. But if the operands do anything with side effects, the order could be important. For these simple expressions, it doesn't matter, because I've used literals, so we can't really tell when they get evaluated. But what about an expression in

which operands call some method? [Example 2-22](#) contains code of this kind.

Example 2-22. Operand evaluation order

```
class Program
{
    static int X(string label, int i)
    {
        Console.Write(label);
        return i;
    }

    static void Main(string[] args)
    {
        Console.WriteLine(X("a", 1) + X("b", 1) + X("c", 1) +
X("d", 1));
    }
}
```

This defines a method, `X`, which takes two arguments. It prints out the first, and just returns the second. I've then used this in a couple of expressions, and it lets us see exactly when the operands that call `X` are evaluated. Some languages choose not to define this order, making the behavior of such a program unpredictable, but C# does specify an order here. The rule is that within any expression, the operands are evaluated in the order in which they appear in the source. So, for the first `Console.WriteLine` in [Example 2-22](#), we see it print `abcd4`.

However, this glosses over an important subtlety: what do we mean by the order of expressions when nesting occurs? The entire argument to that first `Console.WriteLine` is one big add expression, where the first operand is `X("a", 1)`, and the second is another add expression, which in turn has a first operand of `X("b", 1)` and a second operand which is yet another add expression, whose operands are `X("c", 1)` and

`X("d", 1)`. Taking the first of those add expressions, which constitutes the entire argument to `Console.WriteLine` does it even make sense to ask whether it comes before or after its first operand? Lexically, it starts at exactly the same point that its first operand starts, and ends at the point where its second operand ends (which also happens to be at the exact same point that the final `X("d", 1)` ends). In this particular case, it doesn't really matter because the only observable effect of the order of evaluation is the output the `X` method produces when invoked. None of the expressions that invoke `X` are nested within one another, and so we can meaningfully say what order those expressions are in, and the output we see matches that order. However, in some cases, such as [Example 2-23](#) the overlapping of nested expressions can have a visible impact.

Example 2-23. Operand evaluation order with nested expressions

```
Console.WriteLine(  
    X("a", 1) +  
    X("b", (X("c", 1) + X("d", 1) + X("e", 1))) +  
    X("f", 1));
```

Here, `Console.WriteLine` adds the results of three calls to `X`; however, the second of those calls to `X` (first argument `"b"`) takes as its second argument an expression that adds the results of three more calls to `X` (with arguments of `"c"`, `"d"`, and `"e"`). With the final call to `X` (passing `"f"`) we have a total of 6 expressions invoking `X` in that statement. C#'s rule of evaluating expressions in the order in which they appear applies as always, but because there is overlap, the results are initially surprising. Although the letters appear in the source in alphabetical order, the output is `"acdeb5"`. If you're wondering how on earth that can be consistent with expressions being evaluated in order, consider that this code starts the evaluation of each expression in the order

in which the expressions start, and finishes the evaluation in the order in which the expressions finish, but that those are two different orderings. In particular, the expression that invokes X with "b" begins its evaluation before those that invoke it with "c", "d", and "e", but it finishes its evaluation *after* them. And it's that *after* ordering that we see in the output. If you find each closing parenthesis that corresponds to a call to X in this example, you'll find that the order of calls exactly matches what's printed out.

Comments and Whitespace

Most programming languages allow source files to contain text that is ignored by the compiler, and C# is no exception. As with most C-family languages, it supports two styles of *comments* for this purpose. There are *single-line comments*, as shown in [Example 2-24](#), in which you write two / characters in a row, and everything from there to the end of the line will be ignored by the compiler.

Example 2-24. Single line comments

```
Console.WriteLine("Say");           // This text will be ignored but
the code on
Console.WriteLine("Anything");      // the left is still compiled as
usual.
```



C# also supports *delimited comments*. You start a comment of this kind with /*, and the compiler will ignore everything that follows until it encounters the first */ character sequence. This can be useful if you don't want the comment to go all the way to the end of the line, as the first line of [Example 2-25](#) illustrates. This example also shows that delimited comments can span multiple lines.

Example 2-25. Delimited comments

```
Console.WriteLine(/* Has side effects */ GetLog());

/* Some developers like to use delimited comments for big blocks
   * of text,
   * where they need to explain something particularly complex or
   * odd in the
   * code. The column of asterisks on the left is for decoration -
   * asterisks
   * are necessary only at the start and end of the comment.
   */
```

There's a slight snag you can run into with delimited comments; it can happen even when the comment is within a single line, but it more often occurs with multiline comments. [Example 2-26](#) shows the problem with a comment that begins in the middle of the first line, and ends at the end of the fourth.

Example 2-26. Multiline comments

```
Console.WriteLine("This will run"); /* This comment includes not
just the
Console.WriteLine("This won't");      * text on the right, but
also the text
Console.WriteLine("Nor will this");    * on the left except the
first and last
Console.WriteLine("Nor this");         * lines. */
Console.WriteLine("This will also run");
```

Notice that the `/*` character sequence appears twice in this example. When this sequence appears in the middle of a comment, it does nothing special—comments don't nest. Even though we've seen two `/*` sequences, the first `*/` is enough to end the comment. This is occasionally frustrating, but it's the norm for C-family languages.

Occasionally, it's useful to take a chunk of code out of action temporarily, in a way that's easy to put back. Turning the code into a comment is a

common way to do this, and although a delimited comment might seem like the obvious thing to use, it becomes awkward if the region you commented out happens to include a delimited comment. Since there's no support for nesting, you would need to add a `/*` after the inner comment's closing `*/` to ensure that you've commented out the whole range. So it is common to use single-line comments for this purpose. (You can also use the `#if` directive described in [Example 2-28](#).)

NOTE

Visual Studio can comment out regions of code for you. If you select several lines of text, and type Ctrl-K followed immediately by Ctrl-C, it will add `//` to the start of every line in the selection. And you can uncomment a region with Ctrl-K, Ctrl-U. If you chose something other than C# as your preferred language when you first ran Visual Studio, these actions may be bound to different key sequences, but they are also available on the Edit → Advanced menu, as well as on the Text Editor toolbar, one of the standard toolbars that Visual Studio shows by default.

Speaking of ignored text, C# ignores extra whitespace for the most part. Not all whitespace is insignificant, because you need at least some space to separate tokens that consist entirely of alphanumeric symbols. For example, you can't write `staticvoid` as the start of a method declaration—you'd need at least one space (or tab, newline, or other space-like character) between `static` and `void`. But with nonalphanumeric tokens, spaces are optional, and in most cases, a single space is equivalent to any amount of whitespace and new lines. This means that the three statements in [Example 2-27](#) are all equivalent.

Example 2-27. Insignificant whitespace

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



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

There are a couple of cases where C# is more sensitive to whitespace. Inside a string literal, space is significant, because whatever spaces you write will be present in the string value. Also, while C# mostly doesn't care whether you put each element on its own line, or put all your code in one massive line, or (as seems more likely) something in between, there is an exception: preprocessing directives are required to appear on their own lines.

Preprocessing Directives

If you're familiar with the C language or its direct descendants, you may have been wondering if C# has a preprocessor. It doesn't have a separate preprocessing stage, and it does not offer macros. However, it does have a handful of directives similar to those offered by the C preprocessor, although it is only a very limited selection.

Compilation Symbols

C# offers a `#define` directive that lets you define a *compilation symbol*. These symbols are commonly used to compile code in different ways for different situations. For example, you might want some code to be present only in debug builds, or perhaps you need to use different code on different platforms to achieve a particular effect. Often, you won't use the `#define` directive, though—it's more common to define compilation symbols through the compiler build settings. Visual Studio lets you configure different symbol values for each build configuration. To control this, right-click the project's node in Solution Explorer, select Properties,

and in the property page that this opens, go to the Build tab. Or you can just open up the `.csproj` file and define the values you want in a `<DefineConstants>` element of a `<PropertyGroup>`.

NOTE

The .NET SDK defines certain symbols by default in newly created projects. It supports two configurations, Debug and Release. It defines a `DEBUG` compilation symbol in the Debug configuration whereas Release will define `RELEASE` instead. It defines a symbol called `TRACE` in both Debug and Release builds. Certain project types get additional symbols. A library targeting .NET Standard 2.0 will have both `NETSTANDARD` and `NETSTANDARD2_0` defined, for example.

Compilation symbols are typically used in conjunction with the `#if`, `#else`, `#elif`, and `#endif` directives. [Example 2-28](#) uses some of these directives to ensure that certain lines of code get compiled only in Debug builds. (You can also write `#if false` to prevent sections of code from being compiled at all. This is typically done only as a temporary measure, and is an alternative to commenting out that sidesteps some of the lexical pitfalls of nested comments.)

Example 2-28. Conditional compilation

```
#if DEBUG
    Console.WriteLine("Starting work");
#endif
    DoWork();
#if DEBUG
    Console.WriteLine("Finished work");
#endif
```



C# provides a more subtle mechanism to support this sort of thing, called a *conditional method*. The compiler recognizes an attribute defined by the

.NET Framework class libraries, called `ConditionalAttribute`, for which it provides special compile-time behaviors. You can annotate any method with this attribute. [Example 2-29](#) uses it to indicate that the annotated method should be used only when the `DEBUG` compilation symbol is defined.

Example 2-29. Conditional method

```
[System.Diagnostics.Conditional("DEBUG")]  
static void ShowDebugInfo(object o)  
{  
    Console.WriteLine(o);  
}
```

If you call a method that has been annotated in this way, the C# compiler will omit that call in builds that do not have the relevant symbol defined. So if you write code that calls this `ShowDebugInfo` method, the compiler strips out all those calls in non-Debug builds. This means you can get the same effect as [Example 2-28](#), but without cluttering up your code with directives.

The .NET Framework's `Debug` and `Trace` classes in the `System.Diagnostics` namespace use this feature. The `Debug` class offers various methods that are conditional on the `DEBUG` compilation symbol, while the `Trace` class has methods conditional on `TRACE`. If you leave the default settings for a new C# project in place, any diagnostic output produced through the `Trace` class will be available in both Debug and Release builds, but any code that calls a method on the `Debug` class will not get compiled into Release builds.

WARNING

The `Debug` class's `Assert` method is conditional on `DEBUG`, which sometimes

catches developers out. `Assert` lets you specify a condition that must be true at runtime, and it throws an exception if the condition is false. There are two things developers new to C# often mistakenly put in a `Debug.Assert`: checks that should in fact occur in all builds, and expressions with side effects that the rest of the code depends on. This leads to bugs, because the compiler will strip this code out in non-Debug builds.

#error and #warning

C# lets you choose to generate compiler errors or warnings with the `#error` and `#warning` directives. These are typically used inside of conditional regions, as [Example 2-30](#) shows, although an unconditional `#warning` could be useful as a way to remind yourself that you've not written some particularly important bit of the code yet.

Example 2-30. Generating a compiler error

```
#if NETSTANDARD
    #error .NET Standard is not a supported platform for this source
    file
#endif
```



#line

The `\#line` directive is useful in generated code. When the compiler produces an error or a warning, it normally states where the problem occurred, providing the filename, a line number, and an offset within that line. But if the code in question was generated automatically using some other file as input, and if that other file contains the root cause of the problem, it may be more useful to report an error in the input file, rather than the generated file. A `\#line` directive can instruct the C# compiler to act as though the error occurred at the line number specified, and optionally, as if the error were in an entirely different file. [Example 2-31](#)

shows how to use it. The error after the directive will be reported as though it came from line 123 of a file called *Foo.cs*.

Example 2-31. The #line directive and a deliberate mistake

```
#line 123 "Foo.cs"
    intt x;
```

The filename part is optional, enabling you to fake just line numbers. You can tell the compiler to revert to reporting warnings and errors without fakery by writing `#line default`.

There's another use for this directive. Instead of a line number (and optional filename) you can write just `#line hidden`. This affects only the debugger behavior: when single stepping, Visual Studio will run straight through all the code after such a directive without stopping until it encounters a non-hidden `#line` directive (typically `#line default`).

#pragma

The `#pragma` directive provides two features: it can be used to disable selected compiler warnings, and it can also be used to override the checksum values the compiler puts into the *.pdb* file it generates containing debug information. Both of these are designed primarily for code generation scenarios, although it can very occasionally be useful to disable warnings in ordinary code. [Example 2-32](#) shows how to use a `#pragma` to prevent the compiler from issuing the warning that would normally occur if you declare a variable that you do not then go on to use.

Example 2-32. Disabling a compiler warning

```
#pragma warning disable CS0168
    int a;
```



You should generally avoid disabling warnings. This feature is useful in generated code because code generation can often end up creating items that are not always used, and pragmas may offer the only way to get a clean compilation. But when you're writing code by hand, it should usually be possible to avoid warnings in the first place.

It's possible that future versions of C# may add other features based on this directive. In fact, when the compiler encounters a pragma it does not understand, it generates a warning, not an error, on the grounds that an unrecognized pragma might be valid for some future compiler version or some other vendor's compiler.

#region and #endregion

Finally, we have two preprocessing directives that do nothing. If you write **#region** directives, the only thing the compiler does is ensure that they have corresponding **#endregion** directives. Mismatches cause compiler errors, but the compiler ignores correctly paired **#region** and **#endregion** directives. Regions can be nested.

These directives exist entirely for the benefit of text editors that choose to recognize them. Visual Studio uses them to provide the ability to collapse sections of the code down to a single line on screen. The C# editor automatically allows certain features to be expanded and collapsed, such as methods and class definitions, but if you define regions with these two directives, it will also allow those to be expanded and collapsed. If you hover the mouse over a collapsed region, Visual Studio displays a tool tip showing the region's contents.

You can put text after the **#region** token. When Visual Studio displays a

collapsed region, it shows this text on the single line that remains. Although you're allowed to omit this, it's usually a good idea to include some descriptive text so that people can have a rough idea of what they'll see if they expand it.

Some people like to put the entire contents of a class into regions, because by collapsing all regions, you can see a file's structure at a glance. It may all fit on the screen at once, thanks to the regions being reduced to a single line. On the other hand, some people hate collapsed regions, because they present speed bumps on the way to being able to look at the code, and can also encourage people to put too much source code into one file.

Intrinsic Data Types

The .NET Framework defines thousands of types in its class library, and you can write your own, so C# can work with an unlimited number of data types. However, a handful of types get special treatment from the compiler. You saw earlier in [Example 2-9](#) that if you have a string, and you try to add a number to it, the compiler will generate code that converts the number to a string and appends it to the first string. In fact, the behavior is more general than that—it's not limited to numbers. If you have a string, and you add to it some value of any type that's not a string, the compiler calls the `Tostring` method on whatever you're trying to add, and then calls the `String.Concat` method to combine the string with the result. All types offer a `Tostring` method, so this means you can append values of any type to a string.

That's handy, but it only works because the C# compiler knows about strings, and provides special services for them. (There's a part of the C# specification that defines the unique string handling for the `+` operator.) C#

provides various special services not just for strings, but also certain numeric data types, and Booleans. It defines a family of types called tuples. And it has special handling for type types called `dynamic` and `object`.

Numeric Types

C# supports integer and floating-point arithmetic. There are signed and unsigned versions of the integer types, and they come in various sizes, as [Table 2-1](#) shows. The most commonly used integer type is `int`, not least because it is large enough to represent a usefully wide range of values, without being too large to work efficiently on all CPUs that support .NET. (Larger data types might not be handled natively by the CPU, and can also have undesirable characteristics in multithreaded code: reads and writes are atomic for 32-bit types,⁴ but may not be for larger ones.)

Table 2-1. Integer types

C# type	CLR name	Signed	Size in bits	Inclusive range
byte	System.Byte	No	8	0 to 255
sbyte	System.SByte	Yes	8	−128 to 127
ushort	System.UInt16	No	16	0 to 65535
short	System.Int16	Yes	16	−32768 to 32767
uint	System.UInt32	No	32	0 to 4294967295
int	System.Int32	Yes	32	−2147483648 to 2147483647
ulong	System.UInt64	No	64	0 to 18446744073709551615
long	System.Int64	Yes	64	−9223372036854775808 to 9223372036854775807

The second column in [Table 2-1](#) shows the name of the type in the CLR. Different languages have different naming conventions, and C# uses names from its C-family roots for numeric types, but those don't fit with the naming conventions that .NET has for its data types. As far as the runtime is concerned, the names in the second column are the real names—there are various APIs that can report information about types at runtime, and they report these CLR names, not the C# ones. The names are synonymous in C# source code, so you're free to use the runtime names if you want to, but the C# names are a better stylistic fit—keywords in C-family languages are all lowercase. Since the compiler handles these types differently than most, it's arguably good to have them stand out.

WARNING

Not all .NET languages support unsigned numbers, so the .NET Framework class library tends to avoid them. A runtime that supports multiple languages (such as the CLR) faces a trade-off between offering a type system rich enough to cover most languages' needs, and forcing an overcomplicated type system on simple languages. To resolve this, .NET's type system, the CTS, is reasonably comprehensive, but languages don't have to support all of it. The Common Language Specification (CLS) identifies a relatively small subset of the CTS that all languages should support. Signed integers are in the CLS, but unsigned ones are not. This is why you will sometimes see surprising-looking type choices, such as the `Length` property of an array being `int` despite the fact that it will never return a negative value.

C# also supports floating-point numbers. There are two types: `float` and `double`, which are 32-bit and 64-bit numbers in the standard IEEE 754⁵ formats, and as the CLR names in [Table 2-2](#) suggest, these correspond to what are commonly called *single-precision* and *double-precision numbers*. Floating-point values do not work in the same way as integers, so this table is a little different than the integer types table. Floating point

numbers store a value and an exponent (similar in concept to scientific notation, but working in binary instead of decimal). The precision column shows how many bits are dedicated to the value part, and then the range is expressed the smallest nonzero value and the largest value that can be represented. (These can be either positive or negative.)

Table 2-2. Floating-point types

C# type	CLR name	Size in bits	Precision	Range (magnitude)
float	System.Single	32	23 bits (~7 decimal digits)	1.5×10^{-45} to 3.4×10^{38}
double	System.Double	64	52 bits (~15 decimal digits)	5.0×10^{-324} to 1.7×10^{308}

C# recognizes a third numeric representation called `decimal` (or `System.Decimal` in the CLR). This is a 128-bit value, so it can offer greater precision than the other formats, but it is not just a bigger version of `double`. It is designed for calculations that require predictable handling of decimal fractions, something neither `float` nor `double` can offer. If you write code that initializes a variable of type `float` to 0 and then adds 0.1 to it nine times in a row, you might expect to get a value of 0.9, but in fact you'll get 0.9000001. That's because IEEE 754 stores numbers in binary, which cannot represent all decimal fractions. Some are fine, such as the decimal 0.5; written in base 2, that's 0.1. But the decimal 0.1 turns into a recurring number in binary. (Specifically, it's 0.0 followed by the recurring sequence 0011.) This means `float` and `double` can represent only an approximation of the decimal value 0.1, and more generally, only a few decimals can be represented completely accurately. This isn't always instantly obvious, because when floating-point numbers

are converted to text, they are rounded to a decimal approximation that can mask the discrepancy. But over multiple calculations, the inaccuracies tend to add up, and eventually produce surprising-looking results.

For some kinds of calculations, this doesn't really matter; in simulations or signal processing, for example, some noise and error is expected. But accountants tend to be less forgiving—little discrepancies like this can make it look like money has magically vanished or appeared. We need calculations that involve money to be absolutely accurate, which makes floating point a terrible choice for such work. This is why C# offers the `decimal` type, which provides a well-defined level of decimal precision.

NOTE

Most of the integer types can be handled natively by the CPU. (All of them can when running in a 64-bit process.) Likewise, many CPUs can work directly with `float` and `double` representations. However, they do not have intrinsic support for `decimal`, meaning that even simple operations, such as addition, require multiple CPU instructions. This means that arithmetic is significantly slower with `decimal` than with the other numeric types shown so far.

A `decimal` stores numbers as a sign bit (positive or negative) and a pair of integers. There's a 96-bit integer, and the value of the decimal is this first integer (negated if the sign bit says so) multiplied by 10 raised to the power of the second integer, which is a number in the range of 0 to -28 .⁶ 96 bits is enough to represent any 28-digit decimal integer (and some, but not all, 29-digit ones), so the second integer—the one representing the power of 10 by which the first is multiplied—effectively says where the decimal point goes. This format makes it possible to represent any decimal with 28 or fewer digits accurately.

When you write a literal numeric value, you can choose the type, or you can let the compiler pick a suitable type for you. If you write a plain integer, such as `123`, its type will be `int`, `uint`, `long`, or `ulong`—the compiler picks the first type from that list with a range that contains the value. (So `123` would be an `int`, `3000000000` would be a `uint`, `5000000000` would be a `long`, etc.) If you write a number with a decimal point, such as `1.23`, its type is `double`.

If you're dealing with large numbers, it's very easy to get the number of zeros wrong. This is usually bad, and possibly very expensive or dangerous, depending on your application area. C# provides some mitigation by allowing you to add underscores anywhere in numeric literals, to break the numbers up however you please. This is analogous to the common practice in most English-speaking countries of using a comma to separate zeros into groups of 3. For example, instead of writing `5000000000`, most native English speakers would write `5,000,000,000`, instantly making it much easier to see that this is 5 billion, and not, say, 50 billion, or 500 million. (What many native English speakers don't know is that several countries around the world use a period for this, and would write `5.000.000.000` instead, using the comma where purpose most native English speakers would use a decimal point. This leads to the fun fact that interpreting a value such as `€100.000` requires you to know what country it came from, if you don't want to make a disastrous financial miscalculation. But I digress.) In C# we can do something similar by writing the numeric literal as `5_000_000_000`.

You can tell the compiler that you want a specific type by adding a suffix. So `123U` is a `uint`, `123L` is a `long`, and `123UL` is a `ulong`. Suffix letters are case- and order-independent, so instead of `123UL`, you could write `123Lu`, `123uL`, or any other permutation. For `double`, `float`,

and `decimal`, use the `D`, `F`, and `M` suffixes, respectively.

These last three types all support a decimal exponential literal format for large numbers, where you put the letter `E` in the constant followed by the power. For example, the literal value `1.5E-20` is the value 1.5 multiplied by 10^{-20} . (This happens to be of type `double`, because that's the default for a number with a decimal point, regardless of whether it's in exponential format. You could write `1.5E-20F` and `1.5E-20M` for `float` and `decimal` constants with equivalent values.)

It's often useful to be able to write integer literals in hexadecimal, because the digits map better onto the binary representation used at runtime. This is particularly important when different bit ranges of a number represent different things. For example, you may need to deal with a numeric error code that originated from a Windows system call—these occasionally crop up in exceptions. In some cases, these codes use the topmost bit to indicate success or failure, and the next few bits to indicate the origin of the error, and the remaining bits to identify the specific error. For example, the COM error code `E_ACCESSDENIED` has the value `-2,147,024,891`. It's hard to see the structure in decimal, but in hexadecimal, it's easier: `80070005`. The `007` part indicates that this was originally a plain Win32 error that has been translated into a COM error, and then the remaining bits indicate that the Win32 error code was `5` (`ERROR_ACCESS_DENIED`). C# lets you write integer literals in hexadecimal for scenarios like these, where the hex representation is more readable. You just prefix the number with `0x`, so in this case, you would write `0x80070005`.

You can also write binary literals by using the `0b` prefix. Digit separators can be used in hex and binary just as they can in decimals, although it's

more common to group the digits by fours in these cases, e.g.
`0b_0010_1010`.

NUMERIC CONVERSIONS

Each of the built-in numeric types uses a different representation for storing numbers in memory. Converting from one form to another requires some work—even the number 1 looks quite different if you inspect its binary representations as a `float`, an `int`, and a `decimal`. However, C# is able to generate code that converts between formats, and it will often do so automatically. [Example 2-33](#) shows some cases in which this will happen.

Example 2-33. Implicit conversions

```
int i = 42;  
double di = i;  
Console.WriteLine(i / 5);  
Console.WriteLine(di / 5);  
Console.WriteLine(i / 5.0);
```

The second line assigns the value of an `int` variable into a `double` variable. The C# compiler will generate the necessary code to convert the integer value into its equivalent (or nearest approximately equivalent) floating-point value. More subtly, the last two lines will perform similar conversions, as we can see from the output of that code:

```
8  
8.4  
8.4
```

This shows that the first division produced an integer result—dividing the integer variable `i` by the integer literal 5 caused the compiler to generate code that performs integer division, so the result is 8. But the other two

divisions produced a floating-point result. In the second case, we've divided the `double` variable `di` by an integer literal 5. C# converts that 5 to floating point before performing the division. And in the final line, we're dividing an integer variable by a floating-point literal. This time, it's the variable's value that gets turned from an integer into a floating-point value before the division takes place.

In general, when you perform arithmetic calculations that involve a mixture of numeric types, C# will pick the type with the largest range, and *promote* values of types with a narrower range into that larger one before performing the calculations. (Arithmetic operators generally require all their operands to have the same type, so if you supply operands with different types, one type has to “win” for any particular operator.) For example, `double` can represent any value that `int` can, and many that it cannot, so `double` is the more expressive type.⁷

C# will perform numeric conversions implicitly whenever the conversion is a promotion (i.e., the target type has a wider range than the source), because there is no possibility of the conversion failing. However, it will not implicitly convert in the other direction. The second and third lines of [Example 2-34](#) will fail to compile, because they attempt to assign expressions of type `double` into an `int`, which is a *narrowing* conversion, meaning that the source might contain values that are out of the target's range.

Example 2-34. Errors: implicit conversions not available

```
int i = 42;  
int willFail = 42.0;  
int willAlsoFail = i / 1.0;
```



It is possible to convert in this direction, just not implicitly. You can use a

cast, where you specify the name of the type to which you'd like to convert in parentheses. [Example 2-35](#) shows a modified version of [Example 2-34](#), where we've stated explicitly that we want a conversion to `int`, and we either don't mind that this conversion might not work correctly, or we have reason to believe that, in this specific case, the value will be in range. Note that I've used parentheses on the final line. That makes the cast apply to the parenthesized expression; otherwise, it would apply just to the `i` variable, and since that's already an `int`, it would have no effect.

Example 2-35. Explicit conversions with casts

```
int i = 42;  
int i2 = (int) 42.0;  
int i3 = (int) (i / 1.0);
```

So narrowing conversions require explicit casts, and conversions that cannot lose information occur implicitly. However, with some combinations of types, neither is strictly more expressive than the other. What should happen if you try to add an `int` to a `uint`? Or an `int` to a `float`? These types are all 32 bits in size, so none of them can possibly offer more than 2^{32} distinct values, but they have different ranges, which means that each has values it can represent that the other types cannot. For example, you can represent the value 3,000,000,001 in a `uint`, but it's too large for an `int`, and can only be approximated in a `float`. As floating-point numbers get larger, the values that can be represented get farther apart—a `float` can represent 3,000,000,000 and also 3,000,001,024, but nothing in between. So for the value 3,000,000,001, `uint` seems better than `float`. But what about -1? That's a negative number, so `uint` can't cope with that. Then there are very large numbers that `float` can represent that are out of range for both `int` and `uint`. Each of these

types has its strengths and weaknesses, and it makes no sense to say that one of them is generally better than the rest.

Perhaps surprisingly, C# allows some implicit conversions even in these potentially lossy scenarios. It cares only about range, not precision: implicit conversions are allowed if the target type has a wider range than the source type. So you can convert from either `int` or `uint` to `float`, because although `float` is unable to represent some values exactly, there are no `int` or `uint` values that it cannot at least approximate. But implicit conversions are not allowed in the other direction, because there are some values that are simply too big—unlike `float`, the integer types can't offer approximations for bigger numbers.

You might be wondering what happens if you force a narrowing conversion to `int` with a cast, as [Example 2-35](#) does, in situations where the number is out of range. The answer depends on the type from which you are casting. Conversion from one integer type to another works differently than conversion from floating point to integer. In fact, the C# specification does not define what you get when floating-point numbers that are too big get cast to an integer type—the result could be anything. But when casting between integer types, the outcome is well defined. If the two types are of different sizes, the binary will be either truncated or padded with zeros to make it the right size for the target type, and then the bits are just treated as if they are of the target type. This is occasionally useful, but can more often produce surprising results, so you can choose an alternative behavior for any out-of-range cast by making it a *checked* conversion.

CHECKED CONTEXTS

C# defines the `checked` keyword, which you can put in front of either a

statement or an expression, making it a *checked context*. This means that certain arithmetic operations, including casts, are checked for range overflow at runtime. If you cast a value to an integer type in a checked context, and the value is too high or low to fit, an error will occur—the code will throw a `System.OverflowException`.

As well as checking casts, a checked context will detect range overflows in ordinary arithmetic. Addition, subtraction, and other operations can take a value beyond the range of its data type. For integers, this typically causes the value to “roll over,” so adding 1 to the maximum value produces the minimum value, and vice versa for subtraction. Occasionally, this wrapping can be useful. For example, if you want to determine how much time has elapsed between two points in the code, one way to do this is to use the `Environment.TickCount` property.⁸ (This is more reliable than using the current date and time, because that can change as a result of the clock being adjusted, or when moving between time zones. The tick count just keeps increasing at a steady rate. That said, in real code you’d probably use the class library’s `Stopwatch` class.) [Example 2-36](#) shows one way to do this.

Example 2-36. Exploiting unchecked integer overflow

```
int start = Environment.TickCount;
DoSomeWork();
int end = Environment.TickCount;

int totalTicks = end - start;
Console.WriteLine(totalTicks);
```



The tricky thing about `Environment.TickCount` is that it occasionally “wraps around.” It counts the number of milliseconds since the system last rebooted, and since its type is `int`, it will eventually run

out of range. A span of 25 days is 2.16 billion milliseconds—too large a number to fit in an `int`. Imagine the tick count is 2,147,483,637, which is 10 short of the maximum value for `int`. What would you expect it to be 100 ms later? It can't be 100 higher (2,147,483,727), because that's too big a value for an `int`. We'd expect it to get to the highest possible value after 10 ms, so after 11 ms, it'll roll round to the minimum value; thus, after 100 ms, we'd expect the tick count to be 89 above the minimum value (which would be -2,147,483,559).

WARNING

The tick count is not necessarily precise to the nearest millisecond in practice. It often stands still for milliseconds at a time before leaping forward in increments of 10 ms, 15 ms, or even more. However, the value still rolls around—you just might not be able to observe every possible tick value as it does so.

Interestingly, [Example 2-36](#) handles this perfectly. If the tick count in `start` was obtained just before the count wrapped, and the one in `end` was obtained just after, `end` will contain a much lower value than `start`, which seems upside down, and the difference between them will be large—larger than the range of an `int`. However, when we subtract `start` from `end`, the overflow rolls over in a way that exactly matches the way the tick count rolls over, meaning we end up getting the correct result regardless. For example, if the `start` contains a tick count from 10 ms before rollover, and `end` is from 90 ms afterward, subtracting the relevant tick counts (i.e., subtracting -2,147,483,558 from 2,147,483,627) seems like it should produce a result of 4,294,967,185. But because of the way the subtraction overflows, we actually get a result of 100, which corresponds to the elapsed time of 100 ms.

But in most cases, this sort of integer overflow is undesirable. It means that when dealing with large numbers, you can get results that are completely incorrect. A lot of the time, this is not a big risk, because you'll be dealing with fairly small numbers, but if there's any possibility that your calculations might encounter overflow, you might want to use a checked context. Any arithmetic performed in a checked context will throw an exception when overflow occurs. You can request this in an expression with the `checked` operator, as [Example 2-37](#) shows. Everything inside the parentheses will be evaluated in a checked context, so you'll see an `OverflowException` if the addition of `a` and `b` overflows. The `checked` keyword does not apply to the whole statement here, so if an overflow happens as a result of adding `c`, that will not cause an exception.

Example 2-37. Checked expression

```
int result = checked(a + b) + c;
```

You can also turn on checking for an entire block of code with a `checked` statement, which is a block preceded by the `checked` keyword, as [Example 2-38](#) shows. Checked statements always involve a block—you cannot just add the `checked` keyword in front of the `int` keyword in [Example 2-37](#) to turn that into a checked statement. You'd also need to wrap the code in braces.

Example 2-38. Checked statement

```
checked
{
    int r1 = a + b;
    int r2 = r1 - (int) c;
}
```

WARNING

A checked block only affects the code inside the block. If the code invokes any methods, those will be unaffected by the presence of the `checked` keyword—there isn't some *checked* bit in the CPU that gets enabled on the current thread inside a checked block. (In other words, this keyword's scope is lexical, not dynamic.)

C# also has an `unchecked` keyword. You can use this inside a checked block to indicate that a particular expression or nested block should not be a checked context. This makes life easier if you want everything except for one particular expression to be checked—rather than having to label everything except the chosen part as checked, you can put all the code into a checked block, and then exclude the one piece that wants to allow overflow without errors.

You can configure the C# compiler to put everything into a checked context by default, so that only explicitly `unchecked` expressions and statements will be able to overflow silently. In Visual Studio, you can configure this by opening the project properties, going to the Build tab, and clicking the Advanced button. Or you can edit the `.csproj` file, adding `<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>` inside a `<PropertyGroup>`. Be aware that there's a significant cost—checking can make individual integer operations several times slower. The impact on your application as a whole will be smaller, because programs don't spend their whole time performing arithmetic, but the cost may still be nontrivial. Of course, as with any performance matter, you should measure the practical impact. You may find that the performance cost is an acceptable price to pay for the guarantee that you will find out about unexpected overflows.

BIGINTEGER

There's one last numeric type worth being aware of: `BigInteger`. It's part of the .NET Framework class library, and gets no special recognition from the C# compiler. However, it defines arithmetic operators and conversions, meaning that you can use it just like the built-in data types. It will compile to slightly less compact code—the compiled format for .NET programs can represent integers and floating-point values natively, but `BigInteger` has to rely on the more general-purpose mechanisms used by ordinary class library types. In theory it is likely to be significantly slower too, although in an awful lot of code, the speed at which you can perform basic arithmetic on small integers is not a limiting factor, so it's quite possible that you won't notice. And as far as the programming model goes, it looks and feels like a normal numeric type in your code.

As the name suggests, a `BigInteger` represents an integer. Its unique selling point is that it will grow as large as is necessary to accommodate values. So unlike the built-in numeric types, it has no theoretical limit on its range. [Example 2-39](#) uses it to calculate values in the Fibonacci sequence, printing out every 100,000th value. This quickly produces numbers far too large to fit into any of the other integer types. I've shown the full source of this example to illustrate that this type is defined in the `System.Numerics` namespace.

Example 2-39. Using BigInteger

```
using System;
using System.Numerics;

class Program
{
    static void Main(string[] args)
    {
        BigInteger i1 = 1;
```

```
BigInteger i2 = 1;
Console.WriteLine(i1);
int count = 0;
while (true)
{
    if (count++ % 1000000 == 0)
    {
        Console.WriteLine(i2);
    }
    BigInteger next = i1 + i2;
    i1 = i2;
    i2 = next;
}
}
```

Although `BigInteger` imposes no fixed limit, there are practical limits. You might produce a number that's too big to fit in the available memory, for example. Or more likely, the numbers may grow large enough that the amount of CPU time required to perform even basic arithmetic becomes prohibitive. But until you run out of either memory or patience, `BigInteger` will grow to accommodate numbers as large as you like.

Booleans

C# defines a type called `bool`, or as the runtime calls it, `System.Boolean`. This offers only two values: `true` and `false`. Whereas some C-family languages allow numeric types to stand in for Boolean values, with conventions such as 0 meaning false and anything else meaning true, C# will not accept a number. It demands that values indicating truth or falsehood be represented by a `bool`, and none of the numeric types is convertible to `bool`. For example, in an `if` statement, you cannot write `if (someNumber)` to get some code to run only when `someNumber` is nonzero. If that's what you want, you need to say

so explicitly by writing `if (someNumber != 0)`.

Strings and Characters

The `string` type (synonymous with the CLR `System.String` type) represents text. A string is a sequence of values of type `char` (or `System.Char`, as the CLR calls it), and each `char` is a 16-bit value representing a single UTF-16 *code unit*.

A common mistake is to think that each `char` represents a character. (The type's name has to share some of the blame for this.) It's often true, but not always. There are two factors to bear in mind: first, something that we might think of as a single character can be made up from multiple Unicode *code points*. (The code point is Unicode's central concept, and in English at least, each character is represented by a single code point.) [Example 2-40](#) uses Unicode's 0301 “COMBINING ACUTE ACCENT” to add an accent to a letter to form the text `café`s.

Example 2-40. Characters vs char

```
char[] chars = { 'c', 'a', 'f', 'e', (char) 0x301, 's' };  
string text = new string(chars);
```

So this string is a sequence of 6 `char` values, but it represents text that seems to contain just 5 characters. There are other ways to achieve this—I could have used code point 00E9 “LATIN SMALL LETTER E WITH ACUTE” to represent that accented character as a single code point. But either approach is valid, and there are plenty of scenarios in which the only way to create the exact character required is to use this combining character mechanism. This means that certain operations on the `char` values in a string can have surprising results—if you were to reverse the order of the values, the resulting string would not look like a reversed

version of the text—the acute accent would now apply to the s, resulting in `śefac`! (If I had used 00E9 instead of combining e with 0301, reversing the characters would have resulted in the less surprising `séfac`.)

Unicode’s combining marks notwithstanding, there is a second factor to consider. The Unicode standard defines more code points than can be represented in a single 16-bit value. (We passed that point back in 2001, when Unicode 3.1 defined 94,205 code points.) UTF-16 represents any code point with a value higher than 65,535 as a pair of UTF-16 code units, referred to as a *surrogate pair*. The Unicode standard defines rules for mapping code points to surrogate pairs in a way that the resulting code units have values in the range 0xD800 to 0xDFFF, a reserved range for which no code points will ever be defined. (E.g., code point 10C48, “OLD TURKIC LETTER ORKHON BASH”, which looks like `𐱈` would become 0xD801, followed by 0xDCC8.)

In summary, items that users perceive as single characters might be represented with multiple Unicode code points, and some single code points might be represented as multiple code units. Manipulating the individual `char` values that make up a `string` is therefore a job you should approach with caution.

IMMUTABILITY OF STRINGS

.NET strings are immutable. There are many operations that sound as though they will modify a string, such as concatenation, or the `ToUpper` and `ToLower` methods offered by instances of the `string` type, but all of these generate a new string, leaving the original one unmodified. This means that if you pass strings as arguments to code you didn’t write, you can be certain that it cannot change your strings.

The downside of immutability is that string processing can be inefficient. If you need to do work that performs a series of modifications to a string, such as building it up character by character, you will end up allocating a lot of memory, because you'll get a separate string for each modification. This creates a lot of extra work for .NET's garbage collector, causing your program to use more CPU time than necessary. In these situations, you can use a type called `StringBuilder`. (This type gets no special recognition from the C# compiler, unlike `string`.) This is conceptually similar to a `string`—it is a sequence of `char` values and offers various useful string manipulation methods—but it is modifiable.

FORMATTING DATA IN STRINGS

C# provides a syntax that makes it easy to produce strings that contain a mixture of fixed text and information determined at runtime. (The official name for this feature is *string interpolation*.) For example, if you have local variables called `name` and `age`, you could use them in a string as [Example 2-41](#) shows.

Example 2-41. Expressions in strings

```
string message = $"{name} is {age} years old";
```

When you put a `$` symbols in front of a string constant, the C# compiler looks for embedded expressions, delimited by braces, and produces code that will insert a textual representation of the expression at that point in the string. (So if `name` and `age` were `Ian` and `46` respectively, the string's value would be `"Ian is 46 years old"`.)

Embedded expressions can be more complex than just variable names, as [Example 2-42](#) shows.

Example 2-42. More complex expressions in strings

```
double width = 3, height = 4;  
string info = $"Hypotenuse: {Math.Sqrt(3 * 3 + 4 * 4)}";
```

This new syntax compiles into code that uses the `string` class's `Format` method (which is how this sort of data formatting was usually done in previous versions of C#). [Example 2-43](#) shows code that is roughly equivalent to what the compiler will produce for [Example 2-41](#) and [Example 2-42](#).

Example 2-43. The effect of string interpolation

```
string message = string.Format("{0} is {1} years old", name, age);  
string info = string.Format("Hypotenuse: {0}", Math.Sqrt(3 * 3 + 4 * 4));
```

This shows that the new string interpolation syntax doesn't enable anything you couldn't have done before. However, it is much less error prone—`string.Format` uses position-based placeholders, and it's all too easy to get an expression appearing in the wrong place. It's also tedious for anyone reading the code to try and work out how the numbered placeholders relate to the arguments that follow, particularly as the number of expressions increases. The new syntax, by contrast, is much less error-prone, and usually much easier to read.

With some data types, there are choices to be made about their textual representation. For example, with floating point numbers, you might want to limit the number of decimal places, or force the use of exponential notation (e.g., `1e6` instead of `1000000`). In .NET, we control this with a *format specifier*, which is a string describing how to convert some data to a string. Some data types have only one reasonable string representation, so they don't support this, but with types that do, you pass the format

specifier as an argument to the `ToString` method. For example, `System.Math.PI.ToString("f4")` formats the PI constant (which is of type `double`) to 4 decimal places ("3.1416"). There are 9 built-in formats for numbers, and if none of those suits your requirements there's also a mini language for defining custom formats. Moreover, different types use different format strings—as you'd expect, dates work quite differently from numbers—so the full range of available formats is too large to list here. Microsoft supplies extensive documentation of the details.

When using `string.Format`, you can include a format specifier in the placeholder, e.g. `{0:f3}` indicates that the first expression is to be formatted with three digits after the decimal point. You can include a format specifier in a similar way with string interpolation. [Example 2-44](#) shows the age with one digit after the decimal point.

Example 2-44. Format specifiers

```
string message = $"{name} is {age:f1} years old";
```

There's one wrinkle with this: with many data types, the process of converting to a string is culture-specific. For example, in the USA and the UK decimals are typically written with a period between the whole number part and the fractional part and you might use commas to group digits for readability. But as mentioned earlier, in some European countries it works the other way around: they use periods to group digits, while the comma denotes the start of the fractional part. So what might be written as 1,000.2 in one country could be written as 1.000,2 in another. (And there may be ambiguity—if you see 1.000, you need to know which country's conventions are in use to know whether that's one, or one thousand.)

As far as numeric literals in source code are concerned, this is a non-issue: C# uses underscores for digit grouping, and always uses a period as the decimal point. But what about formatting numbers at runtime? By default, you will get conventions determined by the current thread's culture, and unless you've changed that, it will use the regional settings of the computer. Sometimes this is useful—it can mean that numbers, dates, and so on are correctly formatted for whatever locale a program runs in. However, it can sometimes be problematic: if your code relies on strings being formatted in a particular way (e.g., to serialize data that will be transmitted over a network) you may need to force a particular set of conventions. For this reason, you can pass the `string.Format` method a *format provider*, an object that controls formatting conventions. Likewise, data types with culture-dependent representations accept an optional format provider argument in their `ToString` methods. But how do you control this when using string interpolation? There's nowhere to put the format provider.

You can solve this by using string interpolation in a slightly different way: you can assign an interpolated string into a variable of type `IFormattable` (or pass it to a method that requires an argument of that type). In this case, the C# compiler will generate slightly different code—instead of creating a string directly, it produces an object with a method that accepts a format provider. [Example 2-45](#) uses this technique to show the same data as in [Example 2-44](#), but using a format provider that .NET Framework supplies for the express purpose of guaranteeing consistent formatting regardless of the locale in which the code runs.

Example 2-45. Format specifiers with invariant culture

```
IFormattable fmt = $"{name} is {age:f1} years old";  
string message = fmt.ToString(null, CultureInfo.InvariantCulture);
```



That odd-looking `null` first argument to `ToString` requires some explanation. `IFormattable` was originally designed for a different purpose. (It's for types that support multiple textual formats, such as `double` and `int`. You'd normally pass a format specifier as the first argument.) In an ideal world, we could assign interpolated strings into a type with a method that takes just one argument, the format provider. However, there isn't any such type already in the framework, and Microsoft didn't want to introduce a new interface type just to support this feature back when they added string interpolation to the language. (That would prevent you from using string interpolation when targeting old versions of .NET; because it repurposes an existing interface, you can use this language feature even if you need to be able to target old versions of .NET.)

VERBATIM STRING LITERALS

C# supports one more way of expressing a string value: you can prefix a string literal with the `@` symbol. Strings of this form, e.g. `@"Hello"` are called *verbatim string literals*. They are useful for two reasons: they can improve the readability of strings containing backslashes, and they make it possible to write multi-line string literals.

In a normal string literal, the compiler treats a backslash as an escape character, enabling various special values to be included. For example, in the literal `"Hello\tworld"` the `\t` denotes a single tab character (code point 9). This is a common way to express control characters in C family languages. You can also use the backslash to include a double quote in a string—the backslash prevents the compiler from interpreting the character as the end of the string. Useful though this is, it makes including a backslash in a string slightly awkward: you have to write two of them.

Since Windows uses backslashes in paths, this can get ugly, e.g., `"C:\\Windows\\System32\\"`. A verbatim string literal can be useful here, because it treats backslashes literally, enabling you to write just `@ "C:\\Windows\\System32"`. (You can still include double quotes in a verbatim literal: just write two double quotes in a row. E.g., `@ "Hello ""world"""` produces the string value `Hello "world"`.)

Verbatim string literals also allow values to span multiple lines. With a normal string literal, the compiler will report an error if the closing double quote is not on the same line as the opening one. But with a verbatim string literal, the string can cover as many lines of source as you like. The resulting string will use whichever line ending convention your source code uses.

Just in case you've not encountered this, one of the unfortunate accidents of computing history is that different systems use different character sequences to denote line endings. The predominant system in internet protocols is to use a pair of control codes for each line end: in either Unicode or ASCII we use code points 13 and 10, denoting a *carriage return*, and a *line feed* respectively, often abbreviated to CR LF. The HTTP specification requires this representation, as do the various popular email standards, SMTP, POP3 and IMAP. It is also the standard convention on Windows. Unfortunately, the UNIX operating system does things differently, as do most of its derivatives and lookalikes such as macOS and Linux—the convention on these systems is to use just a single line feed character. The C# compiler accepts either, and will not complain even if a single source file contains a mixture of both conventions. This introduces a potential problem for multi-line string literals if you are using a source control system which converts line endings for you. For example, 'git' is a very popular source control system, and thanks to its origins (it was

created by Linus Torvalds, who also created Linux) there is a widespread convention of using Unix-style line endings in its repositories. However, it can be configured to convert working copies of files to a CR LF representation, automatically converting them back when committing changes. This means that files will appear to use different line ending conventions depending on whether you're looking at them on a Windows system or a UNIX one. (And it might even vary from one Windows system to another, because the default line handling ending is configurable. Individual users can configure the machine-wide default setting and can also set the configuration for their local clone of any repository if the repository does not specify the setting itself.) This in turn means compiling a file containing a multi-line verbatim string literal on a Windows system could produce subtly different behavior than you'd see with the exact same file on a UNIX system, if automatic line end conversion is enabled (which is it by default on most Windows installations of git). That might be fine—you typically want CR LF when running on Windows and LF on UNIX, but it could cause surprises if you deploy code to a machine running a different OS than the one you built it on. So it's important to provide a `.gitattributes` file in your repositories so that they can specify the required behavior, instead of relying on changeable local settings. If you need to rely on a particular line ending in a string literal, it's best to make your `.gitattributes` disable line end conversions.

Tuples

C# 7.0 introduced a new language feature, called *tuples*. These let you combine multiple values into a single value. The name tuple (which C# shares with many other programming languages that provide a similar feature) is meant to be a generalized version of words like double, triple,

quadruple, and so on, but we generally call them tuples even in cases where we don't need the generality—e.g., even if we're talking about a tuple with two items in it, we still call it a tuple, not a double. [Example 2-46](#) creates a tuple containing two `int` values, and then prints them out.

Example 2-46. Creating and using a tuple

```
(int X, int Y) point = (10, 5);  
Console.WriteLine($"X: {point.X}, Y: {point.Y}");
```

That first line is a variable declaration with an initializer. It's worth breaking this down, because the syntax for tuples makes for a slightly more complex declaration than we've seen so far. Remember, the general pattern for statements of this form is:

```
type identifier = initial-value;
```

So in [Example 2-46](#), the type is `(int X, int Y)`. So we're saying that our variable, `point`, is a tuple containing two values, both of type `int`, and we want to refer to those as `X` and `Y`. The initializer here is `(10, 5)`. So when we run the example, it produces this output:

```
X: 10, Y: 5
```

If you're a fan of `var`, you'll be pleased to know that you can specify the names in the initializer using the syntax shown in [Example 2-47](#), enabling you to use `var` instead of the explicit type. This is equivalent to [Example 2-46](#).

Example 2-47. Naming tuple members in the initializer

```
var point = (X: 10, Y: 5);  
Console.WriteLine($"X: {point.X}, Y: {point.Y}");
```

If you initialize a tuple from existing variables, and you do not specify names, the compiler will presume that you want to use the names of those variables, as [Example 2-48](#) shows.

Example 2-48. Inferring tuple member names from variables

```
int x = 10, y = 5;  
var point = (x, y);  
Console.WriteLine($"X: {point.x}, Y: {point.y}");
```

This raises a stylistic question: should tuple member names start with lowercase or uppercase letters? The members are similar in nature to properties, which we'll be discussing later in the book, and conventionally those start with an uppercase letter. For this reason, many people believe that tuple member names should also be uppercase. To a seasoned .NET developer, that `point.x` just looks weird. However, another .NET convention is that local variables usually start with a lowercase name. If you stick to both of these conventions, tuple name inference doesn't look very useful. Many developers choose to accept lowercase tuple member names for tuples used purely in local variables, because it enables the use of convenient features such as name inference, using uppercase first letters only for tuples that are exposed outside of a method.

Arguably it doesn't matter much, because tuple member names turn out to exist only in the eye of the beholder. Firstly, they're optional. As [Example 2-49](#) shows, it's perfectly legal to omit them. The names just default to `Item1`, `Item2` etc.

Example 2-49.

```
(int, int) point = (10, 5);  
Console.WriteLine($"X: {point.Item1}, Y: {point.Item2}");
```

Secondly, the names are purely for the convenience of the code using the tuples, and are completely ignored by the runtime. You'll have noticed that I've used the same initializer expression, `(10, 5)` as I did earlier—I've not had to specify the field names here. Because it doesn't specify names, the expression's type is `(int, int)` and yet I've been able to assign it straight into an `(int X, int Y)`. That's because the names are essentially irrelevant—these are all the same thing under the covers. (As we'll see later on, these are all instances of a type called `ValueTuple<int, int>`.) C# keeps track of the names we've chosen to use, but as far as the CLR is concerned, all these tuples just have members called `Item1` and `Item2`. An upshot of this is that we can assign any tuple into any variable with the same shape as [Example 2-50](#) shows.

Example 2-50. Structural equivalence of tuples

```
(int X, int Y) point = (40, 6);  
(int Width, int Height) dimensions = point;  
(int Age, int NumberOfChildren) person = point;
```



This flexibility is a double-edged sword. The assignments in [Example 2-50](#) seem rather sketchy. It might conceivably be OK to assign something that represents a location into something that represents a size—there are certainly some situations in which that would be valid. But to then assign that same value into something apparently representing someone's age and the number of children they have looks likely to be wrong. But the compiler won't stop us, because it considers all tuples comprising a pair of `int` values to have the same type. (It's not really any different from the fact that the compiler won't stop you assigning an `int` variable named `age` into an `int` variable named `height`. They're both of type `int`.)

If you want to enforce a semantic distinction, you would be better off defining custom types as described elsewhere. Tuples are really designed as a convenient way to package together a few values into one thing in cases where defining a whole new type wouldn't really be justified.

Of course, C# does require tuples to have an appropriate shape. You cannot assign an `(int, int)` into a `(int, string)`, nor into an `(int, int, int)`. However, all of the implicit conversions in "Numeric conversions" work, so you can assign anything with an `(int, int)` shape into an `(int, double)`, or a `(double, long)`. So a tuple is really just like having a handful of variables neatly packaged inside another variable.

Tuples support comparison, so you can use the `==` and `!=` relational operators described later in this chapter. To be considered equal, two tuples must have the same shape, and each value in the first tuple must be equal to its counterpart in the second tuple.

DECONSTRUCTION

Sometimes you will want to split a tuple back into its component parts. Obviously you can just access each item in turn by its name (or as `Item1`, `Item2`, etc. if you didn't specify names), but C# provides another mechanism, called deconstruction. Example 2-51 declares and initializes two tuples, and then shows two different ways to deconstruct them.

Example 2-51. Constructing then deconstructing tuples

```
(int X, int Y) point1 = (40, 6);  
(int X, int Y) point2 = (12, 34);  
  
(int x, int y) = point1;  
Console.WriteLine($"1: {x}, {y}");
```

```
(x, y) = point2;  
Console.WriteLine($"2: {x}, {y}");
```

This deconstructs `point1` into two variables, `x` and `y`. This particular form of deconstruction also declares the variables into which the tuple is being deconstructed. The alternative form is shown when we deconstruct `point2`—here, we’re deconstructing it into two variables that already exist, so there’s no need to declare them.

Until you become accustomed to this syntax, the first deconstruction example can seem confusingly similar to the first couple of lines, in which we declare and initialize new tuples. In those first couple of lines, the `(int X, int Y)` text signifies a tuple type with two `int` values named `X` and `Y`, but in the deconstruction line when we write `(int x, int y)` we’re actually declaring two variables, each of type `int`. The only significant difference is that in the lines where we’re constructing new tuples, there’s a variable name before the `=` sign. (Also, we’re using uppercase names there, but that’s just a matter of convention. It would be entirely legal to write `(int x, int y) point3 = point1;` That would declare a new tuple with two `int` values named `x` and `y`, stored in a variable named `point3`, initialized with the same values as are in `point1`. Equally, we could write `(int X, int Y) = point1;` That would deconstruct `point` into two local variables called `X` and `Y`.)

Dynamic

C# defines a special type called `dynamic`. This doesn’t directly correspond to any CLR type—when we use `dynamic` in C#, the compiler presents it to the runtime as `object`, which is described in the next section. However, from the perspective of C# code, `dynamic` is a distinct

type, and it enables some special behavior.

With `dynamic`, the compiler makes no attempt at compile time to check whether operations performed by code are likely to succeed. In other words, it effectively disables the statically typed behavior that we normally get with C#. You are free to attempt almost any operation on a `dynamic` variable—you can use arithmetic operators, you can attempt to invoke methods on it, you can try to assign it into variables of some other type, you can try to invoke properties on it. When you do this, the compiler generates code that attempts to make sense of what you've asked it to do at runtime.

If you have come to C# from a language in which this sort of behavior is the norm (e.g., JavaScript), you might be tempted to use `dynamic` because it works in a way you are used to. However, you should be aware that there are a couple of issues with it. First, it was designed with a particular scenario in mind: interoperability with certain pre-.NET Windows components. The Component Object Model (COM) system in Windows is the basis for automatability of the Microsoft Office Suite, and many other applications, and the scripting language built into Office was dynamic in nature. An upshot of this is that a lot of Office's automation APIs used to be extremely hard work to use from C#. One of the big drivers behind adding `dynamic` to the language was a desire to improve this. Now as with all C# features, it was designed with broader applicability in mind, and not simply as an Office interop feature. But since that was the primary goal, you may find that its ability to support idioms you are familiar with from dynamic languages is disappointing. And the second issue to be aware of is that it is not an area of the language that is getting a lot of new work. When it was introduced, Microsoft went to considerable lengths to ensure that all dynamic behavior was as

consistent as possible as the behavior you would have seen if the compiler had known at compile time what types you were going to be using. This means that the infrastructure supporting `dynamic`, the Dynamic Language Runtime (DLR) has to replicate significant portions of C# behavior. However, the DLR has not been updated much since `dynamic` was added. It appeared in C# 4.0, back in 2010. The language has seen many new features since then. Of course, `dynamic` still works, but its capabilities represent how the language looked around a decade ago.

Even when it first appeared, it had some limitations. There are some aspects of C# that depend on the availability of static type information. So it has some problems working with delegates, and also with LINQ. So even from the start, it was at something of a disadvantage compared to using C# as intended, i.e., as a statically typed language.

Object

The last intrinsic data type recognized by the C# compiler is `object` (or `System.Object`, as the CLR calls it). This is the base class of almost⁹ all C# types. A variable of type `object` is able to refer to a value of any type that derives from `object`. This includes all numeric types, the `bool` and `string` types, and any custom types you can define using the keywords we'll look at in the next chapter, such as `class` and `struct`. And it also includes all the types defined by the .NET Framework class library, with the exception of certain types that can only be stored on the stack.

So `object` is the ultimate general-purpose container. You can refer to almost anything with an `object` variable. We will return to this in a later chapter where we look at inheritance.

Operators

Earlier you saw that expressions are sequences of operators and operands. I've shown some of the types that can be used as operands, so now it's time to see what operators C# offers. [Table 2-2](#) shows the operators that support common arithmetic operations.

Name	Example
Identity (unary plus)	<code>+x</code>
Negation (unary minus)	<code>-x</code>
Post-increment	<code>x++</code>
Post-decrement	<code>x--</code>
Pre-increment	<code>++x</code>
Pre-decrement	<code>--x</code>
Multiplication	<code>x * y</code>
Division	<code>x / y</code>
Remainder	<code>x % y</code>
Addition	<code>x + y</code>
Subtraction	<code>x - y</code>

If you're familiar with any other C-family language, all of these should seem familiar. If you are not, the most peculiar ones will probably be the increment and decrement operators. These all have side effects: they add or subtract one from the variable to which they are applied (meaning they can be applied only to variables). With the post-increment and post-decrement, although the variable gets modified, the containing expression

ends up getting the original value. So if `x` is a variable containing the value 5, the value of `x\++` is also 5, even though the `x` variable will have a value of 6 after evaluating the `x++` expression. The pre- forms evaluate to the modified value, so if `x` is initially 5, `++x` evaluates to 6, which is also the value of `x` after evaluating the expression.

Although the operators in [Table 2-2](#) are used in arithmetic, some are available on certain nonnumeric types. As you saw earlier, the `+` symbol represents concatenation when working with strings. And, as you'll see later on, the addition and subtraction operators are also used for combining and removing delegates. C# also offers some operators that perform certain binary operations on the bits that make up a value, shown in [Table 2-2](#). These are not available on floating-point types.

Name	Example
Bitwise negation	<code>~x</code>
Bitwise AND	<code>x & y</code>
Bitwise OR	<code>x y</code>
Bitwise XOR	<code>x ^ y</code>
Shift left	<code>x << y</code>
Shift right	<code>x >> y</code>

The bitwise negation operator inverts all bits in an integer—any binary digit with a value of 1 becomes 0, and vice versa. The shift operators move all the binary digits left or right by one position. A left shift sets the bottom digit to 0. Right shifts of unsigned integers set the top digit with 0, and right shifts of signed integers leave the top digit as it is (i.e., negative numbers remain negative because they keep their top bit set, while

positive numbers keep their top bit as 0, so they also retain their sign).

The bitwise AND, OR, and XOR (exclusive OR) operators perform Boolean logic operations on each bit of the two operands, when applied to integers. These three operators are also available when the operands are of type `bool`. (In effect, these operators treat a `bool` as a one-digit binary number.) There are some additional operators available for `bool` values, shown in [Table 2-2](#). The `!` operator does to a `bool` what the `~` operator does to each bit in an integer.

Name	Example
Logical negation (also known as NOT)	<code>!x</code>
Conditional AND	<code>x && y</code>
Conditional OR	<code>x y</code>

If you have not used other C-family languages, the conditional versions of the AND and OR operators may be new to you. These evaluate their second operand only if necessary. For example, when evaluating `(a && b)`, if the expression `a` is `false`, the code generated by the compiler will not even attempt to evaluate `b`, because the result will be `false` no matter what value `b` has. Conversely, the conditional OR operator does not bother to evaluate its second operand if the first is `true`, because the result will be `true` regardless of the second operand's value. This is significant if the second operand's expression either has side effects (e.g., it includes a method invocation) or might produce an error. For example, you often see code of the form shown in [Example 2-52](#).

Example 2-52. The conditional AND operator

```
if (s != null && s.Length > 10)
```



This checks to see if the variable `s` contains the special value `null`, meaning that it doesn't currently refer to any value. The use of the `&&` operator here is important, because if `s` is `null`, evaluating the expression `s.Length` would cause a runtime error. If we had used the `&` operator, the compiler would have generated code that always evaluates both operands, meaning that we would see a `NullReferenceException` at runtime if `s` is `null`; however, by using the conditional AND operator, we avoid that, because the second operand, `s.Length > 10`, will be evaluated only if `s` is not `null`.

NOTE

Although code of the kind shown in [Example 2-52](#) was once common, it has gradually become much rarer thanks to a feature introduced back in C# 6, *null conditional operators*. If you write `s?.Length` instead of just `s.Length`, the compiler generates code that checks `s` for `null` first, avoiding the `NullReferenceException`. This means the check can become just `if (s?.Length > 10)`. Furthermore, C# 8 introduces a new feature in which the compiler can, if you choose, impose restrictions to prevent certain values from ever being `null`.

[Example 2-52](#) tests to see if a property is greater than 10 by using the `>` operator. This is one of several *relational operators*, which allow us to compare values. They all take two operands and produce a `bool` result. [Table 2-2](#) shows these, and they are supported for all numeric types. Some operators are available on some other types too. For example, you can compare string values with the `==` and `!=` operators. (There is no built-in meaning for the other relational operators with `string` because different

countries have different ideas about the order in which to sort strings. If you want to compare strings, .NET offers the `StringComparer` class, which requires you to select the rules by which you'd like your strings ordered.)

Name	Example
Less than	<code>x < y</code>
Greater than	<code>x > y</code>
Less than or equal	<code>x <= y</code>
Greater than or equal	<code>x >= y</code>
Equal	<code>x == y</code>
Not equal	<code>x != y</code>

As with most C-family languages, the equality operator is a pair of equals signs. This is because a single equals sign also produces a valid expression, and it means something else: it's an assignment, and assignments are expressions too. This can lead to an unfortunate problem in C-family languages: it's all too easy to write `if (x = y)` when you meant `if (x == y)`. Fortunately, this will usually produce a compiler error in C#, because C# has a special type to represent Boolean values. In languages that allow numbers to stand in for Booleans, both pieces of code are legal even if `x` and `y` are numbers. (The first means to assign the value of `y` into `x`, and then to execute the body of the `if` statement if that value is nonzero. That's very different than the second one, which doesn't change the value of anything, and executes the body of the `if` statement only if `x` and `y` are equal.) But in C#, the first example would be meaningful only if `x` and `y` were both of type `bool`.¹⁰

Another feature that's common to the C family is the conditional operator. (This is sometimes also called the ternary operator, because it's the only operator in the language that takes three operands.) It chooses between two expressions. More precisely, it evaluates its first operand, which must be a Boolean expression, and then returns the value of either the second or third operand, depending on whether the value of the first was `true` or `false`, respectively. [Example 2-53](#) uses this to pick the larger of two values. (This is just for illustration. In practice, you'd normally use `.NET`'s `Math.Max` method, which has the same effect but is rather more readable. `Math.Max` also has the benefit that if you use expressions with side effects, it will only evaluate each one once, something you can't do with the approach shown in [Example 2-53](#).)

Example 2-53. The conditional operator

```
int max = (x > y) ? x : y;
```

This illustrates why C and its successors have a reputation for terse syntax. If you are familiar with any language from this family, [Example 2-53](#) will be easy to read, but if you're not, its meaning might not be instantly clear. This will evaluate the expression before the `?` symbol, which is `(x > y)` in this case, and that's required to be an expression that produces a `bool`. If that is `true`, the expression between the `?` and `:` symbols is used (`x`, in this case); otherwise, the expression after the `:` symbol (`y` here) is used.

NOTE

The parentheses in [Example 2-53](#) are optional. I put them in because I think they make the code easier to read.

The conditional operator is similar to the conditional AND and OR operators in that it will evaluate only the operands it has to. It always evaluates its first operand, but it will never evaluate both the second and third operands. That means you can handle `null` values by writing something like [Example 2-54](#). This does not risk causing a `NullReferenceException`, because it will evaluate the third operand only if `s` is not `null`.

Example 2-54. Exploiting conditional evaluation

```
int characterCount = s == null ? 0 : s.Length;
```

However, in some cases, there are simpler ways of dealing with `null` values. Suppose you have a `string` variable, and if it's `null`, you'd like to use the empty string instead. You could write `(s == null ? "" : s)`. But you could just use the *null coalescing* operator instead, because it's designed for precisely this job. This operator, shown in [Example 2-55](#) (it's the `??` symbol), evaluates its first operand, and if that's non-`null`, that's the result of the expression. If the first operand is `null`, it evaluates its second operand and uses that instead.

Example 2-55. The null coalescing operator

```
string neverNull = s ?? "";
```

We could combine a conditional null operator with the null coalescing operator to provide a more succinct alternative to [Example 2-54](#), shown in [Example 2-56](#).

Example 2-56. Conditional null and null coalescing operators

```
int characterCount = s?.Length ?? 0;
```

One of the main benefits offered by the conditional, conditional null, and null coalescing operators is that they allow you to write a single expression in cases where you would otherwise have needed to write considerably more code. This can be particularly useful if you're using the expression as an argument to a method, as in [Example 2-53](#).

Example 2-57. Conditional expression as method argument

```
FadeVolume(gateOpen ? MaxVolume : 0.0, FadeDuration,  
FadeCurve.Linear);
```



Compare this with what you'd need to write if the conditional operator did not exist. You would need an `if` statement. (I'll get to `if` statements in the next section, but since this book is not for novices, I'm assuming you're familiar with the rough idea.) And you'd either need to introduce a local variable, as [Example 2-58](#) does, or you'd need to duplicate the method call in the two branches of the `if/else`, changing just the first argument. So, terse though the conditional and null coalescing operators are, they can remove a lot of clutter from your code.

Example 2-58. Life without the conditional operator

```
double targetVolume;  
if (gateOpen)  
{  
    targetVolume = MaxVolume;  
}  
else  
{  
    targetVolume = 0.0;  
}  
FadeVolume(targetVolume, FadeDuration, FadeCurve.Linear);
```



There is one last set of operators to look at: the *compound assignment* operators. These combine assignment with some other operation, and they

are available for the `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, `|`, and `??` operators. So you don't have to write the sort of code shown in [Example 2-59](#).

Example 2-59. Assignment and addition

```
x = x + 1;
```

We can write this assignment statement more compactly as the code in [Example 2-60](#). All the compound assignment operators take this form—you just stick an `=` on the end of the original operator.

Example 2-60. Compound assignment (addition)

```
x += 1;
```

As well as being more succinct, this can be less offensive to those of a sensitive mathematical disposition. [Example 2-59](#) looks like a mathematical equation, but one that is complete nonsense. (This doesn't stop it being perfectly legal as C#, of course—we are requesting an operation with side effects, rather than stating a mathematical truth. It looks weird only because C-family languages use the `=` symbol to denote assignment rather than equality.) [Example 2-60](#) doesn't even resemble any common basic mathematical notation. More usefully, it is a distinctive syntax that makes it very clear that we are modifying the value of a variable in some particular way. So, although those two snippets perform identical work, many developers find the second idiomatically preferable.

That's not quite a comprehensive list of operators. There are a few more specialized ones that I'll get to once we've looked at the areas of the language for which they were defined. (Some relate to classes and other types, some to inheritance, some to collections, and some to delegates. There are chapters coming up on all of these.) By the way, although I've

been describing which operators are available on which types (e.g., numeric versus Boolean), it's possible to write a custom type that defines its own meanings for most of these. That's how the .NET Framework's `BigInteger` type can support the same arithmetic operations as the built-in numeric types.

Flow Control

Most of the code we have examined so far executes statements in the order they are written, and stops when it reaches the end. If that were the only possible way in which execution could flow through our code, C# would not be very useful. So, as you'd expect, it has a variety of constructs for writing loops, and for deciding which code to execute based on input conditions.

Boolean Decisions with `if` Statements

An `if` statement decides whether or not to run some particular statement depending on the value of a `bool` expression. For example, the `if` statement in [Example 2-61](#) will execute the block statement that prints a message only if the `age` variable's value is less than 18.

Example 2-61. Simple if statement

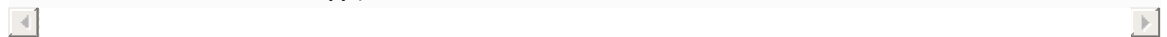
```
if (age < 18)
{
    Console.WriteLine("You are too young to buy alcohol in a bar
in the UK.");
}
```

You don't have to use a block statement with an `if` statement. You can use any statement type as the body. A block is necessary only if you want

the `if` statement to govern the execution of multiple statements. However, many coding style guidelines recommend using a block in all cases. This is partly for consistency, but also because it avoids a possible error when modifying the code at a later date: if you have a nonblock statement as the body of an `if`, and then you add another statement after that, intending it to be part of the same body, it can be easy to forget to wrap it in a block, leading to code like that in [Example 2-62](#). The indentation suggests that the developer meant for the final statement to be part of the `if` statement's body, but C# ignores indentation, so that final statement will always run. If you are in the habit of always using a block, you won't make this mistake.

Example 2-62. Probably not what was intended


```
if (launchCodesCorrect)
    TurnOnMissileLaunchedIndicator();
    LaunchMissiles();
```



An `if` statement can optionally include an `else` part, which is followed by another statement that runs only if the `if` statement's expression evaluates to `false`. So [Example 2-63](#) will print either the first or the second message, depending on whether the `optimistic` variable is `true` or `false`.

Example 2-63. If and else

```
if (optimistic)
{
    Console.WriteLine("Glass half full");
}
else
{
    Console.WriteLine("Glass half empty");
}
```



The `else` keyword can be followed by any statement, and again, this is

typically a block. However, there's one scenario in which most developers do not use a block for the body of the `else` part, and that's when they use another `if` statement. [Example 2-64](#) shows this—its first `if` statement has an `else` part, which has another `if` statement as its body.

Example 2-64. Picking one of several possibilities

```
if (temperatureInCelsius < 52)
{
    Console.WriteLine("Too cold");
}
else if (temperatureInCelsius > 58)
{
    Console.WriteLine("Too hot");
}
else
{
    Console.WriteLine("Just right");
}
```

This code still looks like it uses a block for that first `else`, but that block is actually the statement that forms the body of a second `if` statement. It's that second `if` statement that is the body of the `else`. If we were to stick rigidly to the rule of giving each `if` and `else` body its own block, we'd rewrite [Example 2-64](#) as [Example 2-65](#). This seems unnecessarily fussy, because the main risk that we're trying to avert by using blocks doesn't really apply in [Example 2-64](#).

Example 2-65. Overdoing the blocks

```
if (temperatureInCelsius < 52)
{
    Console.WriteLine("Too cold");
}
else
{
    if (temperatureInCelsius > 58)
```

```
{  
    Console.WriteLine("Too hot");  
}  
else  
{  
    Console.WriteLine("Just right");  
}  
}
```

Although we can chain `if` statements together as shown in [Example 2-64](#), C# offers a more specialized statement that can sometimes be easier to read.

Multiple Choice with `switch` Statements

A `switch` statement defines multiple groups of statements and either runs one group or does nothing at all, depending on the value of an expression. The expression can be any integer type, a `string`, a `char`, or any enumeration type. As [Example 2-66](#) shows, you put the expression inside parentheses after the `switch` keyword, and after that, there's a region delimited by braces containing a series of `case` sections, defining the behavior for each anticipated value for the expression.

Example 2-66. A `switch` statement with strings

```
switch (workStatus)  
{  
    case "ManagerInRoom":  
        WorkDiligently();  
        break;  
  
    case "HaveNonUrgentDeadline":  
    case "HaveImminentDeadline":  
        CheckTwitter();  
        CheckEmail();  
        CheckTwitter();  
        ContemplateGettingOnWithSomeWork();  
}
```

```
    CheckTwitter();  
    CheckTwitter();  
    break;  
  
case "DeadlineOvershot":  
    WorkFuriously();  
    break;  
  
default:  
    CheckTwitter();  
    CheckEmail();  
    break;  
}
```

As you can see, a single section can serve multiple possibilities—you can put several different `case` lines at the start of a section, and the statements in that section will run if any of those cases apply. You can also write a `default` section, which will run if none of the cases apply. By the way, you're not required to provide a `default` section. A `switch` statement does not have to be comprehensive, so if there is no `case` that matches the expression's value, and there is no `default` section, the `switch` statement simply does nothing.

Unlike `if` statements, which take exactly one statement for the body, a `case` may be followed by multiple statements without needing to wrap them in a block. The sections in [Example 2-66](#) are delimited by `break` statements, which causes execution to jump to the end of the `switch` statement. This is not the only way to finish a section—strictly speaking, the rule imposed by the C# compiler is that the end point of the statement list for each `case` must not be reachable, so anything that causes execution to leave the `switch` statement is acceptable. You could use a `return` statement instead, or throw an exception, or you could even use a `goto` statement.

Some C-family languages (C, for example) allow *fall-through*, meaning that if execution is allowed to reach the end of the statements in a **case** section, it will continue with the next one. [Example 2-67](#) shows this style, and it is not allowed in C# because of the rule that requires the end of a **case** statement list not to be reachable.

Example 2-67. C-style fall-through, illegal in C#

```
switch (x)
{
case "One":
    Console.WriteLine("One");
case "Two": // This line will not compile
    Console.WriteLine("One or two");
    break;
}
```

C# outlaws this, because the vast majority of **case** sections do not fall through, and when they do in languages that allow it, it's often a mistake caused by the developer forgetting to write a **break** statement (or some other statement to break out of the **switch**). Accidental fall-through is likely to produce unwanted behavior, so C# requires more than the mere omission of a **break**: if you want fall-through, you must ask for it explicitly. As [Example 2-68](#) shows, we use the unloved **goto** keyword to express that we really do want one case to fall through into the next one.

Example 2-68. Fall-through in C#

```
switch (x)
{
case "One":
    Console.WriteLine("One");
    goto case "Two";
case "Two":
    Console.WriteLine("One or two");
    break;
}
```

This is not technically a `goto` statement. It is a `goto case` statement, and can be used only to jump within a `switch` block. C# does also support more general `goto` statements—you can add labels to your code and jump around within your methods. However, `goto` is heavily frowned upon, so the fall-through form offered by `goto case` statements seems to be the only use for this keyword that is considered respectable in modern society.

Loops: while and do

C# supports the usual C-family loop mechanisms. [Example 2-69](#) shows a `while` loop. This takes a `bool` expression. It evaluates that expression, and if the result is `true`, it will execute the statement that follows. So far, this is just like an `if` statement, but the difference is that once the loop's embedded statement is complete, it then evaluates the expression again, and if it's `true` again, it will execute the embedded statement a second time. It will keep doing this until the expression evaluates to `false`. As with `if` statements, the body of the loop does not need to be a block, but it usually is.

Example 2-69. A while loop

```
while (!reader.EndOfStream)
{
    Console.WriteLine(reader.ReadLine());
}
```

The body of the loop may decide to finish the loop early. A `break` statement will break out of the loop. It does not matter whether the `while` expression is `true` or `false`—executing a `break` statement will always terminate the loop.

C# also offers the `continue` statement. Like a `break` statement, this terminates the current iteration, but unlike `break`, it will then reevaluate the `while` expression, so iteration may continue. Both `continue` and `break` jump straight to the end of the loop, but you could think of `continue` as jumping directly to the point just before the loop's closing `}`, while `break` jumps to the point just after. By the way, `continue` and `break` are also available for all of the other loop styles I'm about to show.

Because a `while` statement evaluates its expression before each iteration, it's possible for a `while` loop not to run its body at all. Sometimes, you may want to write a loop that runs at least once, only evaluating the `bool` expression after the first iteration. This is the purpose of a `do` loop, as shown in [Example 2-70](#).

Example 2-70. A do loop

```
char k;  
do  
{  
    Console.WriteLine("Press x to exit");  
    k = Console.ReadKey().KeyChar;  
}  
while (k != 'x');
```

Notice that [Example 2-70](#) ends in a semicolon, denoting the end of the statement. Compare this with the line containing the `while` keyword [Example 2-69](#), which does not, despite otherwise looking very similar. That may look inconsistent, but it's not a typo. Putting a semicolon at the end of the line with the `while` keyword in [Example 2-69](#) would be legal, but it would change the meaning—it would indicate that we want the body of the `while` loop to be an empty statement. The block that followed

would then be treated as a brand-new statement to execute after the loop completes. The code would get stuck in an infinite loop unless the reader were already at the end of the stream. (The compiler will issue a warning about a “Possible mistaken empty statement” if you do that, by the way.)

C-Style for Loops

Another style of loop that C# inherits from C is the `for` loop. This is similar to `while`, but it adds two features to that loop’s `bool` expression: it provides a place to declare and/or initialize one or more variables that will remain in scope for as long as the loop runs, and it provides a place to perform some operation each time around the loop (in addition to the statement that forms the body of the loop). So the structure of a `for` loop looks like this:

```
for (initializer; condition; iterator) body
```

A very common application of this is to do something to all the elements in an array. [Example 2-71](#) shows a `for` loop that multiplies every element in an array by 2. The condition part works in exactly the same way as in a `while` loop—it determines whether the embedded statement forming the loop’s body runs, and it will be evaluated before each iteration. Again, the body doesn’t strictly have to be a block, but usually is.

Example 2-71. Modifying array elements with a for loop

```
for (int i = 0; i < myArray.Length; i++)  
{  
    myArray[i] *= 2;  
}
```

The initializer in this example declares a variable called `i` and initializes it

to 0. This initialization happens just once, of course—this wouldn't be very useful if it reset the variable to 0 every time around the loop, because the loop would never end. This variable's lifetime effectively begins just before the loop starts, and finishes when the loop finishes. The initializer does not need to be a variable declaration—you can use any expression statement.

The iterator in [Example 2-71](#) just adds 1 to the loop counter. It runs at the end of each loop iteration, after the body runs, and before the condition is reevaluated. (So if the condition is initially false, not only does the body not run, the iterator will never be evaluated.) C# does nothing with the result of the iterator expression—it is useful only for its side effects. So it doesn't matter whether you write `i\++`, `++i`, `i += 1`, or even `i = i + 1`.

The iterator is a redundant construct, because it doesn't let you do anything that you couldn't have achieved by putting the exact same code in a statement at the end of the loop's body instead.¹¹ However, there may be readability benefits. A `for` statement puts the code that defines how we loop in one place, separate from the code that defines what we do each time around the loop, which might help those reading the code to understand what it does. They don't have to scan down to the end of a long loop to find the iterator statement (although a long loop body that trails over pages of code is generally considered to be bad practice, so this last benefit is a little dubious).

Both the initializer and the iterator can contain lists, as [Example 2-72](#) shows, although in this particular case it isn't terribly useful—since all the iterators run every time around, `i` and `j` will have the same value as each other throughout.

Example 2-72. Multiple initializers and iterators

```
for (int i = 0, j = 0; i < myArray.Length; i++, j++)  
...
```

You can't write a single `for` loop that performs a multidimensional iteration. If you want that, you would nest one loop inside another, as [Example 2-73](#) illustrates.

Example 2-73. Nested for loops

```
for (int j = 0; j < height; ++j)  
{  
    for (int i = 0; i < width; ++i)  
    {  
        ...  
    }  
}
```

Although [Example 2-71](#) shows a common enough idiom for iterating through arrays, you will often use a different, more specialized construct.

Collection Iteration with `foreach` Loops

C# offers a style of loop that is not universal in C-family languages. The `foreach` loop is designed for iterating through collections. A `foreach` loop fits this pattern:

```
foreach (item-type iteration-variable in collection) body
```

The `_collection_` is an expression whose type must match a particular pattern recognized by the compiler. The .NET Framework's `IEnumerable<T>` interface matches this pattern, although the compiler doesn't actually require an implementation of that interface—it just requires the collection to implement a `GetEnumerator` method that

resembles the one defined by that interface. [Example 2-74](#) uses `foreach` to print all the strings in an array. (All arrays provide the method that `foreach` requires.)

Example 2-74. Iterating over a collection with `foreach`

```
string[] messages = GetMessagesFromSomewhere();  
foreach (string message in messages)  
{  
    Console.WriteLine(message);  
}
```

This loop will run the body once for each item in the array. The *iteration variable* (`message`, in this example) is different each time around the loop, and will refer to the item for the current iteration.

In one way, this is less flexible than the `for`-based loop shown in [Example 2-71](#): a `foreach` loop cannot modify the collection it iterates over. That's because not all collections support modification.

`IEnumerable<T>` demands very little of its collections—it does not require modifiability, random access, or even the ability to know up front how many items the collection provides. (In fact, `IEnumerable<T>` is able to support never-ending collections. For example, it's perfectly legal to write an implementation that returns random numbers for as long as you care to keep fetching values.)

But `foreach` offers two advantages over `for`. One advantage is subjective, and therefore debatable: it's slightly more readable. But significantly, it's also more general. If you're writing methods that do things to collections, those methods will be more broadly applicable if they use `foreach` rather than `for`, because you'll be able to accept an `IEnumerable<T>`. [Example 2-75](#) can work with any collection that

contains strings, rather than being limited to arrays.

Example 2-75. General-purpose collection iteration

```
public static void ShowMessages(IEnumerable<string> messages)
{
    foreach (string message in messages)
    {
        Console.WriteLine(message);
    }
}
```

This code can work with collection types that do not support random access, such as the `LinkedList<T>` class described later on. It can also process lazy collections that decide what items to produce on demand, including those produced by iterator functions and by certain LINQ queries.

Patterns

There's one last essential mechanism to look at in C#: *patterns*. A pattern describes some criterion or criteria that a value can be tested against. You've already seen some simple patterns in action: each `case` in a `switch` specifies a pattern. But as we'll now see, there are many kinds of patterns, and they aren't just for `switch` statements.

NOTE

Most of the pattern functionality was added to the C# relatively recently. The very simplest kind have been available in `case` labels from the start, but broader support first appeared in C# 7, and most of the pattern types now available were added in C# 8.

The `switch` examples earlier, such as [Example 2-66](#), all used one of the simplest pattern types: the *constant pattern*. With this pattern, you simply specify a constant value, and an expression matches this pattern if it has that value. In a similar vein, [Example 2-76](#) shows *tuple patterns*, which match tuples with specific values. These are conceptually very similar to constant patterns—the values for the individual tuple elements must all be constants with this pattern. (The distinction is largely a matter of history: before patterns were introduced, `case` labels only supported the limited set of types for which the CLR offered intrinsic support for constant values, and that list did not include tuples.)

Example 2-76. Tuple patterns

```
switch (p)
{
    case (0, 0):
        Console.WriteLine("How original");
        break;

    case (0, 1):
    case (1, 0):
        Console.WriteLine("What an absolute unit");
        break;

    case (1, 1):
        Console.WriteLine("Be there and be square");
        break;
}
```

[Example 2-77](#) shows a slightly more interesting kind of pattern: *type patterns*. An expression matches a type pattern if it has the specified type. As you saw earlier in [“Object”](#), some variables are capable of holding a variety of different types. (Variables of type `object` are an extreme case of this, since they can hold more or less anything. Language features such as *interfaces*, inheritance, and generics can lead to scenarios where the

static type of a variable provides more information than the anything-goes `object` type, but with some latitude for a range of possible types at runtime.) Type patterns can be useful in these cases.

Example 2-77. Type patterns

```
switch (o)
{
    case string s:
        Console.WriteLine($"A piece of string is {s.Length} long");
        break;

    case int i:
        Console.WriteLine($"That's numberwang! {i}");
        break;
}
```

Type patterns have an interesting characteristic that constant ones do not: as well as the Boolean match/no-match common to all patterns, a type pattern also produces an output. Each `case` in [Example 2-77](#) introduces a variable, which the code for that `case` then goes on to use. With a type pattern, this output is just the input, but in a variable with the specified static type. So that first `case` will match if `o` turns out to be a `string`, in which case we can access it through the `s` variable (which is why that `s.Length` expression compiles correctly; `o.Length` would not if `o` is of type `object`).

TIP

Sometimes, you won't actually need the output of a pattern—it might be enough just to know that the input matched a pattern. In these cases you can use a *discard*: if you put an underscore (`_`) in the place where the output variable name would normally go, that tells the C# compiler that you are only interested in whether there was a match, and if there is, that it should discard the result.

Some patterns do a little more work to produce their output. For example, [Example 2-78](#) shows a *positional pattern* that matches any tuple containing a pair of `int` values, and extracts those values into two variables, `x` and `y`.

Example 2-78. Positional pattern

```
case (int x, int y):  
    Console.WriteLine($"I know where it's at: {x}, {y}");  
    break;
```

Positional patterns are an example of a *recursive pattern*: they are patterns that contain patterns. In this case, this positional pattern contains a type pattern as each of its children. But we can mix things up, because positional patterns can contain any pattern type (including another recursive pattern, if that's what you need to do). In fact, that's exactly what was going on in [Example 2-76](#)—a tuple pattern is really just a special case of a positional pattern, where the children are all constant patterns. [Example 2-79](#) shows a positional pattern with a constant pattern in the first position, and a type pattern in the second.

Example 2-79. Positional pattern with constant and type patterns

```
case (0, int y):  
    Console.WriteLine($"This is on the X axis, at height {y}");  
    break;
```

If you are a fan of `var` you might be wondering if you can write something like [Example 2-80](#). This will work, and the static types of the `x` and `y` variables here will depend on the type of the pattern's input expression. If the compiler can determine how the expression deconstructs (e.g., if the input's static type is an `(int, int)` tuple), then it will use this information to determine the output variables' static types. In cases

where this is unknown, but it's still conceivable that this pattern could match (e.g., the input is `object`) then `x` and `y` here will also have type `object`.

NOTE

The compiler will reject patterns in cases where it can determine that a match is impossible. For example, if it knows the input type is a `(string, int, bool)` tuple, it cannot possibly match a positional pattern with only 2 child patterns, so C# won't let you try.

Example 2-80. Positional pattern with var

```
case (var x, var y):  
    Console.WriteLine($"I know where it's at: {x}, {y}");  
    break;
```

Example 2-80 shows an unusual case where using `var` instead of an explicit type can introduce a significant change of behavior. These *var patterns* differ in one important respect from the *type patterns* in Example 2-78. A *var pattern* always matches its input, whereas a *type pattern* inspects its input's type to determine at runtime whether it matches. This check might be optimised away in practice—there are cases where a type pattern will always match because its input type is known. But the only way to express in your code that you definitely don't want the child patterns in a positional pattern to perform a runtime check is to use `var`. So although a positional pattern containing type patterns strongly resembles the deconstruction syntax shown in Example 2-51, the behavior is quite different: Example 2-78 is in effect performing three runtime tests: is the value a 2-tuple, is the first value an `int` and is the second value an `int`? (So it would work for tuples of type `(object, object)`, as

long as each value is an `int`). This shouldn't really be surprising: the point of patterns is to test at runtime whether a value has certain characteristics. However, with some recursive patterns you may find yourself wanting to express a mixture of runtime matching (e.g., is this thing a `string`?) combined with statically-typed deconstruction (e.g., if this is a `string` I'd like to extract its `Length` property which I believe to be of type `int`, and I want a compiler error if that belief turns out to be wrong). Patterns are not designed to do this.

What if we don't need to use all of the items in the tuple? You already know one way to handle that. Since we can use any pattern in each position, we could use a type pattern with a discard in, say, the second position: `(int x, int _)`. However, [Example 2-81](#) shows a shorter alternative: instead of a discarding type pattern, we can use just a lone underscore. This is a *discard pattern*. You can use it a recursive pattern any place a pattern is required, but where you want to indicate that anything will do in that particular position.

Example 2-81. Positional pattern with discard pattern

```
case (int x, _):  
    Console.WriteLine($"At X: {x}. As for Y, who knows?");  
    break;
```

This has subtly different semantics though: a type pattern with a discard will check at runtime that the value to be discarded has the specified type, and the pattern will only match if this check succeeds. A discard pattern always matches.

Positional patterns are not the only recursive ones: you can also write a *property pattern*. We'll look at properties in detail in the next chapter, but for now it's enough to know that they are members of a type that provide

some sort of information, such as the `string` type's `Length` property, which provides an `int` telling you how many code units the string contains. [Example 2-82](#) shows a *property pattern* that inspects this `Length` property.

Example 2-82. Property pattern

```
case string { Length: 0 } s:
    Console.WriteLine($"How long is a piece of string? Not
very!");
    break;
```

A property pattern starts with a type name, so it effectively incorporates the behavior of a type pattern in addition to its property-based tests. (You can omit this in cases where the pattern's input's type is sufficiently specific to identify the property. E.g., if the input in this case were already of type `string`, we could omit this.) This is then followed by a section in braces listing each of the properties that the pattern wants to inspect, and the pattern to apply for that property. (These child patterns are what make this another recursive pattern.) So this example first checks to see if the input is a `string`, and if it is, it then applies a constant pattern to the string's `Length`, so this pattern matches only if the input is a `string` with `Length` of 0.

Summary

In this chapter, I showed the nuts and bolts of C# code—variables, statements, expressions, basic data types, operators, and flow control. Now it's time to take a look at the broader structure of a program. All code in C# programs must belong to a type, and types are the topic of the next chapter.

-
- 1 C# does in fact offer dynamic typing as an option with its `dynamic` keyword, but it takes the slightly unusual step of fitting that into a statically typed point of view: dynamic variables have a static type of `dynamic`.
 - 2 See Alan Turing's seminal work on computation for details. Charles Petzold's *The Annotated Turing* (John Wiley & Sons) is an excellent guide to the relevant paper.
 - 3 In the absence of parentheses, C# has rules of *precedence* that determine the order in which operators are evaluated. For the full (and not very interesting) details, consult the C# specification, but in this case, division has higher precedence than addition, so without parentheses, the expression would evaluate to 14.
 - 4 Strictly speaking, this is guaranteed only for correctly aligned 32-bit types. However, C# aligns them correctly by default, and you'd normally encounter misaligned data only if your code needs to call out into unmanaged code.
 - 5 http://en.wikipedia.org/wiki/IEEE_floating_point
 - 6 A decimal, therefore, doesn't use all of its 128 bits. Making it smaller would cause alignment difficulties, and using the additional bits for extra precision would have a significant performance impact, because integers whose length is a multiple of 32 bits are easier for a most CPUs to deal with than the alternatives.
 - 7 Promotions are not in fact a feature of C#. C# has a more general mechanism: conversion operators. Promotions are built on top of this—C# defines intrinsic implicit conversion operators for the built-in data types. The promotions discussed here occur as a result of the compiler following its usual rules for conversions.
 - 8 A *property* is a member of a type that represents a value that can be either read or modified or both.
 - 9 There are some specialized exceptions, such as pointer types.
 - 10 Language pedants will note that this is not exactly right. It will also be meaningful in certain situations where custom implicit conversions to `bool` are available. We'll be getting to custom conversions in a later chapter.
 - 11 A `continue` statement complicates matters, because it provides a way to move to the next iteration without getting all the way to the end of the loop body. Even so, you could still reproduce the effect of the iterator when using `continue` statements—it would just require more work.

About the Author

Ian Griffiths is a C# and WPF instructor with Pluralsight; the coauthor of *Programming WPF*, Second Edition (O'Reilly); and a widely recognized expert on the subject of WPF. Ian is a technical fellow at Endjin, a technology consultant firm. He holds a degree in computer science from Cambridge University.