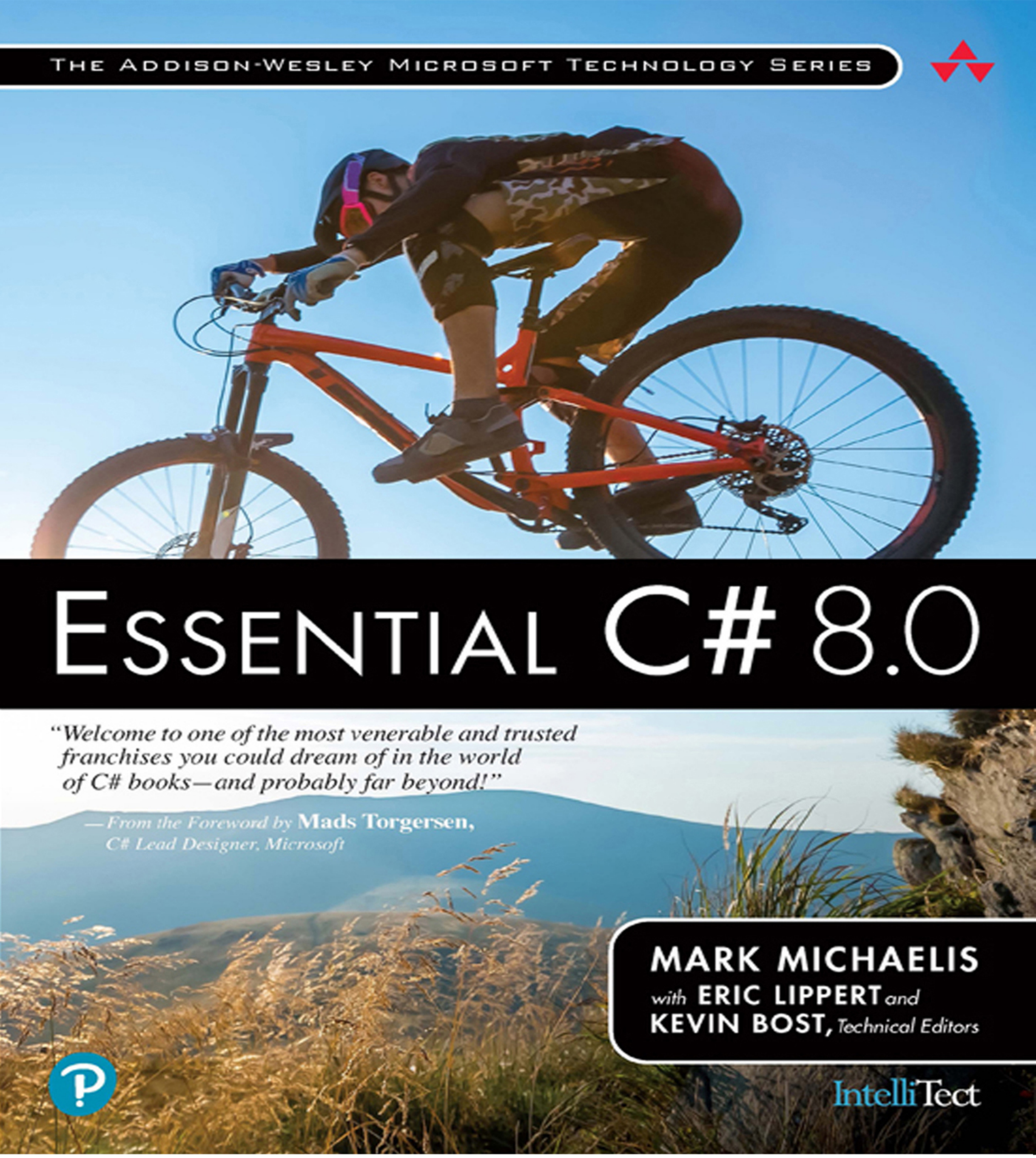


THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES

ESSENTIAL C# 8.0

"Welcome to one of the most venerable and trusted franchises you could dream of in the world of C# books—and probably far beyond!"

—From the Foreword by **Mads Torgersen,** *C# Lead Designer, Microsoft*

MARK MICHAELIS

with **ERIC LIPPERT** *and* **KEVIN BOST,** *Technical Editors*

IntelliTect

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Essential C# 8.0

This page intentionally left blank

Essential C# 8.0

Mark Michaelis
with Eric Lippert & Kevin Bost, Technical Editors

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town • Dubai
London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2020940544

Cover image: Taras Hipp/Shutterstock

ISBN-13: 978-0-13-597226-7
ISBN-10: 0-13-597226-4
ScoutAutomatedPrintCode

To my family: Elisabeth, Benjamin, Hanna, and Abigail.
You have sacrificed a husband and daddy for countless hours of writing,
frequently at times when he was needed most.

Thanks!

Also, to my friends and colleagues at IntelliTect. Thanks for filling in for me when I was writing rather than doing my job and for helping with the myriad of details in trying to improve the content and devops processes that help keep a code base like this running smoothly.

This page intentionally left blank

Contents at a Glance

Contents

This page intentionally left blank

Figures

Tables

Foreword

Welcome to one of the most venerable and trusted franchises you could dream of in the world of C# books—and probably far beyond! Mark Michaelis's *Essential C#* book has been a classic for years, but it was yet to see the light of day when I first got to know Mark.

In 2005, when LINQ (Language Integrated Query) was disclosed, I had only just joined Microsoft, and I got to tag along to the PDC conference for the big reveal. Despite my almost total lack of contribution to the technology, I thoroughly enjoyed the hype. The talks were overflowing, the printed leaflets were flying off the tables like hotcakes: It was a big day for C# and .NET, and I was having a great time.

It was pretty quiet in the hands-on labs area, though, where people could try out the technology preview themselves with nice scripted walk-throughs. That's where I ran into Mark. Needless to say, he wasn't following the script. He was doing his own experiments, combing through the docs, talking to other folks, busily pulling together his own picture.

As a newcomer to the C# community, I may have met a lot of people for the first time at that conference—people with whom I have since formed great relationships. But to be honest, I don't remember them—it's all a blur. The only one I remember is Mark. Here is why: When I asked him if he was liking the new stuff, he didn't just join the rave. He was totally level-headed: "*I don't know yet. I haven't made up my mind about it.*" He wanted to absorb and understand the full package, and until then he wasn't going to let anyone tell him what to think.

So instead of the quick sugar rush of affirmation I might have expected, I got to have a frank and wholesome conversation, the first of many over the years, about details, consequences, and concerns with this new technology. And so it remains: Mark is an incredibly valuable community member for us language designers to have, because he is super smart, insists on understanding everything to the core, and has phenomenal insight into how things affect real developers. But perhaps most of all, he is forthright and never afraid to speak his mind. If something passes the Mark Test, then we know we can start feeling pretty good about it!

These are the same qualities that make Mark such a great writer. He goes right to the essence and communicates with great integrity, no sugarcoating, and a keen eye for practical value and real-world problems. Mark has a great gift of providing clarity and elucidation, and no one will help you get C# 8.0 like he does.

Enjoy!

—Mads Torgersen,
C# Lead Designer,
Microsoft

Preface

Throughout the history of software engineering, the methodology used to write computer programs has undergone several paradigm shifts, each building on the foundation of the former by increasing code organization and decreasing complexity. This book takes you through these same paradigm shifts.

The beginning chapters take you through **sequential programming structure**, in which statements are executed in the order in which they are written. The problem with this model is that complexity increases exponentially as the requirements increase. To reduce this complexity, code blocks are moved into methods, creating a **structured programming model**. This allows you to call the same code block from multiple locations within a program, without duplicating code. Even with this construct, however, programs quickly become unwieldy and require further abstraction. Object-oriented programming, introduced in Chapter 6, was the response. In subsequent chapters, you will learn about additional methodologies, such as interface-based programming, LINQ (and the transformation it makes to the collection API), and eventually rudimentary forms of declarative programming (in Chapter 18) via attributes.

This book has three main functions.

- It provides comprehensive coverage of the C# language, going beyond a tutorial and offering a foundation upon which you can begin effective software development projects.
- For readers already familiar with C#, this book provides insight into some of the more complex programming paradigms and provides

in-depth coverage of the features introduced in the latest version of the language, C# 8.0 and .NET Framework 4.8/.NET Core 3.1.

- It serves as a timeless reference even after you gain proficiency with the language.

The key to successfully learning C# is to start coding as soon as possible. Don't wait until you are an "expert" in theory; start writing software immediately. As a believer in iterative development, I hope this book enables even a novice programmer to begin writing basic C# code by the end of Chapter 2.

Many topics are not covered in this book. You won't find coverage of topics such as ASP.NET, Entity Framework, Xamarin, smart client development, distributed programming, and so on. Although these topics are relevant to .NET, to do them justice requires books of their own. Fortunately, Addison-Wesley's Microsoft Windows Development Series provides a wealth of writing on these topics. *Essential C# 8.0* focuses on C# and the types within the Base Class Library. Reading this book will prepare you to focus on and develop expertise in any of the areas covered by the rest of the series.

Target Audience for This Book

My challenge with this book was to keep advanced developers awake while not abandoning beginners by using words such as *assembly*, *link*, *chain*, *thread*, and *fusion* as though the topic was more appropriate for blacksmiths than for programmers. This book's primary audience is experienced developers looking to add another language to their quiver. However, I have carefully assembled this book to provide significant value to developers at all levels.

- *Beginners:* If you are new to programming, this book serves as a resource to help transition you from an entry-level programmer to a C# developer who is comfortable with any C# programming task that's thrown your way. This book not only teaches you syntax but also trains you in good programming practices that will serve you throughout your programming career.
- *Structured programmers:* Just as it's best to learn a foreign language through immersion, learning a computer language is most effective

when you begin using it before you know all the intricacies. In this vein, this book begins with a tutorial that will be comfortable for those familiar with structured programming, and by the end of Chapter 5, developers in this category should feel at home writing basic control flow programs. However, the key to excellence for C# developers is not memorizing syntax. To transition from simple programs to enterprise development, the C# developer must think natively in terms of objects and their relationships. To this end, Chapter 6's Beginner Topics introduce classes and object-oriented development. The role of historically structured programming languages such as C, COBOL, and FORTRAN is still significant but shrinking, so it behooves software engineers to become familiar with object-oriented development. C# is an ideal language for making this transition because it was designed with object-oriented development as one of its core tenets.

- *Object-based and object-oriented developers:* C++, Java, Python, TypeScript, Visual Basic, and Java programmers fall into this category. Many of you are already completely comfortable with semicolons and curly braces. A brief glance at the code in Chapter 1 reveals that, at its core, C# is like other C- and C++-style languages that you already know.
- *C# professionals:* For those already versed in C#, this book provides a convenient reference for less frequently encountered syntax. Furthermore, it provides insight into language details and subtleties that are seldom addressed. Most important, it presents the guidelines and patterns for programming robust and maintainable code. This book also aids in the task of teaching C# to others. With the emergence of C# 3.0 through 8.0, some of the most prominent enhancements are
  - String interpolation (see Chapter 2)
  - Implicitly typed variables (see Chapter 3)
  - Tuples (see Chapter 3)
  - Nullable reference types (see Chapter 3)
  - Pattern matching (see Chapter 4)
  - Extension methods (see Chapter 6)
  - Partial methods (see Chapter 6)
  - Default interface members (see Chapter 8)
  - Anonymous types (see Chapter 12)
  - Generics (see Chapter 12)
  - Lambda statements and expressions (see Chapter 13)

- Expression trees (see Chapter 13)
- Standard query operators (see Chapter 15)
- Query expressions (see Chapter 16)
- Dynamic programming (Chapter 18)
- Multithreaded programming with the Task Programming Library and `async` (Chapter 20)
- Parallel query processing with PLINQ (Chapter 21)
- Concurrent collections (Chapter 22)

These topics are covered in detail for those not already familiar with them. Also pertinent to advanced C# development is the subject of pointers, covered in Chapter 23. Even experienced C# developers often do not understand this topic well.

Features of This Book

Essential C# 8.0 is a language book that adheres to the core C# Language Specification. To help you understand the various C# constructs, it provides numerous examples demonstrating each feature. Accompanying each concept are guidelines and best practices, ensuring that code compiles, avoids likely pitfalls, and achieves maximum maintainability.

To improve readability, code is specially formatted and chapters are outlined using mind maps.

C# Coding Guidelines

One of the more significant enhancements included in *Essential C# 8.0* is the C# coding guidelines, as shown in the following example taken from Chapter 17:

Guidelines

DO ensure that equal objects have equal hash codes.

DO ensure that the hash code of an object never changes while it is in a hash table.

DO ensure that the hashing algorithm quickly produces a well-distributed hash.

DO ensure that the hashing algorithm is robust in any possible object state.

These guidelines are the key to differentiating a programmer who knows the syntax from an expert who can discern the most effective code to write based on the circumstances. Such an expert not only gets the code to compile but does so while following best practices that minimize bugs and enable maintenance well into the future. The coding guidelines highlight some of the key principles that readers will want to be sure to incorporate into their development. Visit https://IntelliTect.com/Guidelines for a current list of all the guidelines.

Code Samples

The code snippets in most of this text can run on most implementations of the Common Language Infrastructure (CLI), but the focus is on the Microsoft .NET Framework and the .NET Core implementation. Platform- or vendor-specific libraries are seldom used except when communicating important concepts relevant only to those platforms (e.g., appropriately handling the single-threaded user interface of Windows). Any code that specifically relates to C# 5.0, 6.0, 7.0, or 8.0 is called out in the C# version indexes at the end of the book.

Here is a sample code listing.

Begin 2.0

Listing 1.19: Commenting Your Code

```
class CommentSamples
{
  static void Main()
  {
                          single-line comment
      string firstName; //Variable for storing the first name
      string lastName;  //Variable for storing the last name

      System.Console.WriteLine("Hey you!");
                         delimited comment inside statement
      System.Console.Write /* No new line */ (
          "Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write /* No new line */ (
          "Enter your last name: ");
      lastName = System.Console.ReadLine();

      /* Display a greeting to the console   } delimited comment
         using composite formatting. */

      System.Console.WriteLine("Your full name is {0} {1}.",
          firstName, lastName);
      // This is the end
      // of the program listing
  }
}
```

The formatting is as follows.

- Comments are shown in italics.

```
/* Display a greeting to the console
   using composite formatting */
```

- Keywords are shown in bold.

```
static void Main()
```

- Highlighted code calls out specific code snippets that may have changed from an earlier listing, or demonstrates the concept described in the text.

```
System.Console.WriteLine(valerie);
miracleMax = "It would take a miracle.";
System.Console.WriteLine(miracleMax);
```

  Highlighting can appear on an entire line or on just a few characters within a line.

```
System.Console.WriteLine(
    $"The palindrome \"{palindrome}\" is"
    + $" {palindrome.Length} characters.");
```

- Incomplete listings contain an ellipsis to denote irrelevant code that has been omitted.

```
// ...
```

- Console output is the output from a particular listing that appears following the listing. User input for the program appears in **boldface**.

Output 1.7

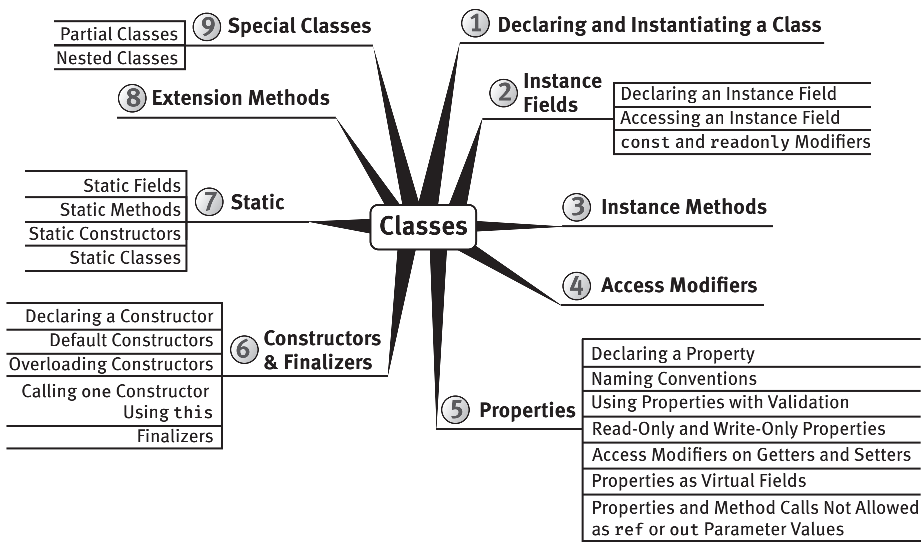
```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya

Your full name is Inigo Montoya.
```

Although it might have been convenient to provide full code samples that you could copy into your own programs, doing so would detract from your learning a particular topic. Therefore, you need to modify the code samples before you can incorporate them into your programs. The core omission is error checking, such as exception handling. Also, code samples do not explicitly include using System statements. You need to assume the statement throughout all samples.

You can find sample code at https://IntelliTect.com/EssentialCSharp.

Mind Maps

Each chapter's introduction includes a **mind map**, which serves as an outline that provides an at-a-glance reference to each chapter's content. Here is an example (taken from Chapter 6).



The theme of each chapter appears in the mind map's center. High-level topics spread out from the core. Mind maps allow you to absorb the flow from high-level to more detailed concepts easily, with less chance of encountering very specific knowledge that you might not be looking for.

Helpful Notes

Depending on your level of experience, special features will help you navigate through the text.

- Beginner Topics provide definitions or explanations targeted specifically toward entry-level programmers.
- Advanced Topics enable experienced developers to focus on the material that is most relevant to them.

- Callout notes highlight key principles in boxes so that readers easily recognize their significance.
- Language Contrast sidebars identify key differences between C# and its predecessors to aid those familiar with other languages.

How This Book Is Organized

At a high level, software engineering is about managing complexity, and it is toward this end that I have organized *Essential C# 8.0.* Chapters 1–5 introduce structured programming, which enable you to start writing simple functioning code immediately. Chapters 6–10 present the object-oriented constructs of C#. Novice readers should focus on fully understanding this section before they proceed to the more advanced topics found in the remainder of this book. Chapters 12–14 introduce additional complexity-reducing constructs, handling common patterns needed by virtually all modern programs. This leads to dynamic programming with reflection and attributes, which is used extensively for threading and interoperability in the chapters that follow.

The book ends with Chapter 24 on the Common Language Infrastructure, which describes C# within the context of the development platform in which it operates. This chapter appears at the end because it is not C# specific and it departs from the syntax and programming style in the rest of the book. However, this chapter is suitable for reading at any time, perhaps most appropriately immediately following Chapter 1.

Here is a description of each chapter (in this list, chapter numbers shown in **bold** indicate the presence of C# 7.0-8.0 material).

- *Chapter 1, Introducing C#:* After presenting the C# `HelloWorld` program, this chapter proceeds to dissect it. This should familiarize readers with the look and feel of a C# program and provide details on how to compile and debug their own programs. It also touches on the context of a C# program's execution and its intermediate language.
- *Chapter 2, Data Types:* Functioning programs manipulate data, and this chapter introduces the primitive data types of C#.
- ***Chapter 3**, More with Data Types:* This chapter includes coverage of two type categories, value types and reference types. From there, it delves into implicitly typed local variables, tuples, the nullable

modifier, and the C# 8.0–introduced feature, nullable reference types. It concludes with an in-depth look at a primitive array structure.

- *Chapter 4, Operators and Control Flow:* To take advantage of the iterative capabilities in a computer, you need to know how to include loops and conditional logic within your program. This chapter also covers the C# operators, data conversion, and preprocessor directives.
- *Chapter 5, Methods and Parameters:* This chapter investigates the details of methods and their parameters. It includes passing by value, passing by reference, and returning data via an `out` parameter. In C# 4.0, default parameter support was added, and this chapter explains how to use default parameters.
- ***Chapter 6**, Classes:* Given the basic building blocks of a class, this chapter combines these constructs to form fully functional types. Classes form the core of object-oriented technology by defining the template for an object. This chapter also includes the nullable attributes newly introduced in C# 8.0.
- *Chapter 7, Inheritance:* Although inheritance is a programming fundamental to many developers, C# provides some unique constructs, such as the `new` modifier. This chapter discusses the details of the inheritance syntax, including overriding.
- ***Chapter 8**, Interfaces:* This chapter demonstrates how interfaces are used to define the versionable interaction contract between classes. C# includes both explicit and implicit interface member implementation, enabling an additional encapsulation level not supported by most other languages. With the introduction of default interface members, there is a new section on interface versioning in C# 8.0.
- *Chapter 9, Value Types:* Although not as prevalent as defining reference types, it is sometimes necessary to define value types that behave in a fashion similar to the primitive types built into C#. This chapter describes how to define structures while exposing the idiosyncrasies they may introduce.
- ***Chapter 10**, Well-Formed Types:* This chapter discusses more advanced type definition. It explains how to implement operators, such as `+` and casts, and describes how to encapsulate multiple classes into a single library. In addition, the chapter demonstrates defining namespaces

and XML comments and discusses how to design classes for garbage collection.

- *Chapter 11, Exception Handling:* This chapter expands on the exception-handling introduction from Chapter 5 and describes how exceptions follow a hierarchy that enables creating custom exceptions. It also includes some best practices on exception handling.
- *Chapter 12, Generics:* Generics are perhaps the core feature missing from C# 1.0. This chapter fully covers this 2.0 feature. In addition, C# 4.0 added support for covariance and contravariance—something covered in the context of generics in this chapter.
- *Chapter 13, Delegates and Lambda Expressions:* Delegates begin clearly distinguishing C# from its predecessors by defining patterns for handling events within code. This virtually eliminates the need for writing routines that poll. Lambda expressions are the key concept that make C# 3.0's LINQ possible. This chapter explains how lambda expressions build on the delegate construct by providing a more elegant and succinct syntax. This chapter forms the foundation for the collection API discussed next.
- *Chapter 14, Events:* Encapsulated delegates, known as events, are a core construct of the Common Language Runtime. Anonymous methods, another C# 2.0 feature, are also presented here.
- *Chapter 15, Collection Interfaces with Standard Query Operators:* The simple and yet elegantly powerful changes introduced in C# 3.0 begin to shine in this chapter as we take a look at the extension methods of the `Enumerable` class. This class makes available a collection API known as the standard query operators, which is discussed in detail here.
- ***Chapter 16**, LINQ with Query Expressions:* Using standard query operators alone results in some long statements that are hard to decipher. However, query expressions provide an alternative syntax that matches closely with SQL, as described in this chapter.
- ***Chapter 17**, Building Custom Collections:* In building custom APIs that work against business objects, it is sometimes necessary to create custom collections. This chapter details how to do this and in the process introduces contextual keywords that make custom collection building easier.

- ***Chapter 18**, Reflection, Attributes, and Dynamic Programming:* Object-oriented programming formed the basis for a paradigm shift in program structure in the late 1980s. In a similar way, attributes facilitate declarative programming and embedded metadata, ushering in a new paradigm. This chapter looks at attributes and discusses how to retrieve them via reflection. It also covers file input and output via the serialization framework within the Base Class Library. In C# 4.0, a new keyword, `dynamic`, was added to the language. This removed all type checking until runtime, a significant expansion of what can be done with C#.
- *Chapter 19, Introducing Multithreading:* Most modern programs require the use of threads to execute long-running tasks while ensuring active response to simultaneous events. As programs become more sophisticated, they must take additional precautions to protect data in these advanced environments. Programming multithreaded applications is complex. This chapter introduces how to work with tasks, including canceling them, and how to handle exceptions executing in the task context.
- ***Chapter 20**, Programming the Task-Based Asynchronous Pattern:* This chapter delves into the task-based asynchronous pattern with its accompanying `async`/`await` syntax. It provides a significantly simplified approach to multithreaded programming. In addition, the C# 8.0 concept of asynchronous streams is included.
- *Chapter 21, Iterating in Parallel:* One easy way to introduce performance improvements is by iterating through data in parallel using a `Parallel` object or with the Parallel LINQ library.
- *Chapter 22, Thread Synchronization:* Building on the preceding chapter, this chapter demonstrates some of the built-in threading pattern support that can simplify the explicit control of multithreaded code.
- *Chapter 23, Platform Interoperability and Unsafe Code:* Given that C# is a relatively young language, far more code is written in other languages than in C#. To take advantage of this preexisting code, C# supports interoperability—the calling of unmanaged code—through P/Invoke. In addition, C# provides for the use of pointers and direct memory manipulation. Although code with pointers requires special

privileges to run, it provides the power to interoperate fully with traditional C-based application programming interfaces.

- *Chapter 24, The Common Language Infrastructure:* Fundamentally, C# is the syntax that was designed as the most effective programming language on top of the underlying Common Language Infrastructure. This chapter delves into how C# programs relate to the underlying runtime and its specifications.
- *Indexes of C# 6.0, 7.0, and 8.0:* These indexes provide quick references for the features added in C# 6.0 through 8.0. They are specifically designed to help programmers quickly update their language skills to a more recent version.

I hope you find this book to be a great resource in establishing your C# expertise and that you continue to reference it for those areas that you use less frequently well after you are proficient in C#.

—Mark Michaelis
IntelliTect.com/mark
Twitter: @Intellitect, @MarkMichaelis

Acknowledgments

NO BOOK CAN BE PUBLISHED by the author alone, and I am extremely grateful for the multitude of people who helped me with this one. The order in which I thank people is not significant, except for those who come first. Given that this is now the seventh edition of the book, you can only imagine how much my family has sacrificed to allow me to write over the last 14 years (not to mention the books before that). Benjamin, Hanna, and Abigail often had a Daddy distracted by this book, but Elisabeth suffered even more so. She was often left to take care of things, holding the family's world together on her own. (While on vacation in 2017, I spent days indoors writing while they would much have preferred to go to the beach.)

The difference when I was writing *Essential C# 8.0* is that instead of sacrificing as much home life, my work is really what suffered. I am so grateful to be surrounded by a team of amazing software engineers who produce such excellence autonomously from me. If that wasn't enough, several engineers stepped in to help me with myriad details, from errata, devops, and listing numbering to technical edits. Special thanks to Cameron Osborn, Phil Spokas (who helped with portions of the writing in Chapter 24), Andres Scott, and, more recently, Austen Frostad.

I have worked with Kevin Bost at IntelliTect since 2013, and he continues to surprise me with his incredible aptitude for software development. Not only is the depth of his C# knowledge phenomenal, but he is a level-10 expert in so many additional development technologies. For all this and more, I asked Kevin Bost to review the book as an official technical editor this year, and I am truly grateful. He brought insights and improvements to content that has been in the book since the early editions, which

no one else thought to mention. This attention to detail, combined with his unswerving demand for excellence, truly establishes *Essential C# 8.0* as a quintessential C# book for those looking to focus on the language.

Of course, Eric Lippert is no less than amazing as well. His grasp of C# is truly astounding, and I am very appreciative of his edits, especially when he pushed for perfection in terminology. His improvements to the C# 3.0 chapters were incredibly significant, and in the second edition my only regret was that I didn't have him review all the chapters. However, that regret has since been mitigated: Eric painstakingly reviewed every *Essential C# 4.0* chapter and even served as a contributing author for *Essential C# 5.0* and *Essential C# 6.0*. I am extremely grateful for his role as a technical editor for *Essential C# 8.0*. Thanks, Eric! I can't imagine anyone better for the job. You deserve all the credit for raising the bar from good to great.

As is the case with Eric and C#, there are fewer than a handful of people who know .NET multithreading as well as Stephen Toub. Accordingly, Stephen concentrated on the two rewritten (for a third time) multithreading chapters and their new focus on `async` support in C# 5.0. Thanks, Stephen!

Over the years, many technical editors have reviewed each chapter in minute detail to ensure technical accuracy. I was often amazed by the subtle errors these folks still managed to catch: Paul Bramsman, Kody Brown, Andrew Comb, Ian Davis, Doug Dechow, Gerard Frantz, Dan Haley, Thomas Heavey, Anson Horton, Brian Jones, Shane Kercheval, Angelika Langer, Neal Lundby, John Michaelis, Jason Morse, Nicholas Paldino, Jason Peterson, Jon Skeet, Michael Stokesbary, Robert Stokesbary, and John Timney.

Thanks to everyone at Pearson/Addison-Wesley for their patience in working with me in spite of my frequent focus on everything else except the manuscript. Thanks to Chris Zahn for his work to format the content and make it readable. Thanks to Jill Hobbs! People like you and your attention to detail and knowledge of the English language astound me. Thanks to the production team, Rob Mauhar and Viola Jasko, who laid out the pages. Your exceptional abilities meant my comments were essentially limited to things that I didn't get right in the manuscript. Thanks to Rachel Paul and her constant follow-up whenever a jot or tittle was found out of place, in addition to all the management she did behind the scenes. Thanks also to Malobika Chakraborty for helping me through the entire process from proposal to production.

About the Author

Mark Michaelis is the founder of IntelliTect, an innovative software architecture and development firm where he serves as the chief technical architect and trainer. Mark leads his successful company while flying around the world delivering conference sessions on leadership or technology and conducting speaking engagements on behalf of Microsoft or other clients. He has also written numerous articles and books, and is an adjunct professor at Eastern Washington University, founder of the Spokane .NET Users Group, and co-organizer of the annual TEDx Coeur d'Alene events.

A world-class C# expert, Mark has been a Microsoft Regional Director since 2007 and a Microsoft MVP for more than 25 years.

Mark holds a bachelor of arts in philosophy from the University of Illinois and a master's degree in computer science from the Illinois Institute of Technology.

When not bonding with his computer, Mark is busy showing his kids real life in other countries or playing racquetball (having suspended competing in Ironman back in 2016). Mark lives in Spokane, Washington, with his wife, Elisabeth, and three children, Benjamin, Hanna, and Abigail.

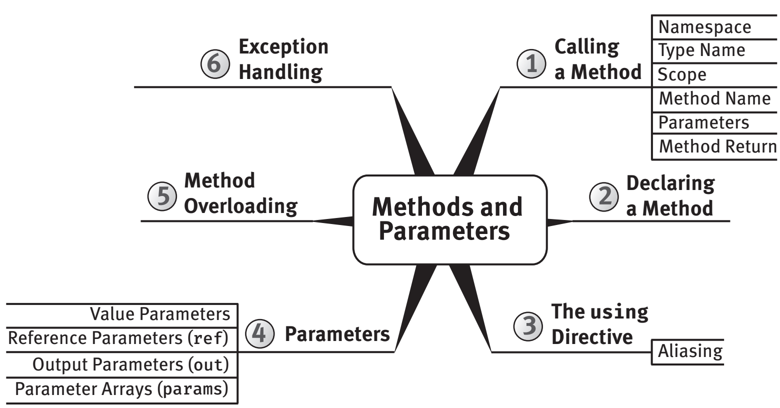
About the Technical Editors

Kevin Bost is a successful Microsoft MVP and senior software architect at IntelliTect. He has been instrumental in the building of several innovative products, including System.CommandLine, Moq.AutoMocker, and ShowMeTheXAML. When not at work, Kevin can be found online mentoring other developers on YouTube (youtube.keboo.dev) and maintaining the popular Material Design in XAML toolkit (http://materialdesigninxaml.net/). He also enjoys board games, Ultimate Frisbee, and riding his motorcycle.

Eric Lippert works on developer tools at Facebook; he is a former member of the C# language design team at Microsoft. When not answering C# questions on StackOverflow or editing programming books, Eric does his best to keep his tiny sailboat upright. He lives in Seattle, Washington, with his wife, Leah.

5
Methods and Parameters

FROM WHAT YOU HAVE LEARNED about C# programming so far, you should be able to write straightforward programs consisting of a list of statements, similar to the way programs were created in the 1970s. Programming has come a long way since the 1970s, however; as programs have become more complex, new paradigms have emerged to manage that complexity. *Procedural* or *structured* programming provides constructs by which statements are grouped together to form units. Furthermore, with structured programming, it is possible to pass data to a group of statements and then have data returned once the statements have executed.



Besides the basics of calling and defining methods, this chapter covers some slightly more advanced concepts—namely, recursion, method overloading, optional parameters, and named arguments. All method calls discussed so far and through the end of this chapter are static (a concept that Chapter 6 explores in detail).

Even as early as the `HelloWorld` program in Chapter 1, you learned how to define a method. In that example, you defined the `Main()` method. In this chapter, you will learn about method creation in more detail, including the special C# syntaxes (`ref` and `out`) for parameters that pass variables rather than values to methods. Lastly, we will touch on some rudimentary error handling.

Calling a Method

BEGINNER TOPIC

What Is a Method?

Up to this point, all of the statements in the programs you have written have appeared together in one grouping called a `Main()` method. When programs become any more complex than those we have seen thus far, a single method implementation quickly becomes difficult to maintain and complex to read through and understand.

A **method** is a means of grouping together a sequence of statements to perform a particular action or compute a particular result. This provides greater structure and organization for the statements that compose a program. Consider, for example, a `Main()` method that counts the lines of source code in a directory. Instead of having one large `Main()` method, you can provide a shorter version that allows you to hone in on the details of each method implementation as necessary. Listing 5.1 shows an example.

LISTING 5.1: Grouping Statements into Methods

```
class LineCount
{
  static void Main()
  {
      int lineCount;
      string files;
      DisplayHelpText();
```

```
            files = GetFiles();
            lineCount = CountLines(files);
            DisplayLineCount(lineCount);
        }
        // ...
    }
```

Instead of placing all of the statements into `Main()`, the listing breaks them into groups called methods. The `System.Console.WriteLine()` statements that display the help text have been moved to the `DisplayHelpText()` method. All of the statements used to determine which files to count appear in the `GetFiles()` method. To actually count the lines, the code calls the `CountLines()` method before displaying the results using the `DisplayLineCount()` method. With a quick glance, it is easy to review the code and gain an overview, because the method name describes the purpose of the method.

Guidelines

DO give methods names that are verbs or verb phrases.

A method is always associated with a type—usually a **class**—that provides a means of grouping related methods together.

Methods can receive data via **arguments** that are supplied for their **parameters**. Parameters are variables used for passing data from the **caller** (the code containing the method call) to the invoked method (`Write()`, `WriteLine()`, `GetFiles()`, `CountLines()`, and so on). In Listing 5.1, `files` and `lineCount` are examples of arguments passed to the `CountLines()` and `DisplayLineCount()` methods via their parameters. Methods can also return data to the caller via a **return value** (in Listing 5.1, the `GetFiles()` method call has a return value that is assigned to `files`).

To begin, we reexamine `System.Console.Write()`, `System.Console.WriteLine()`, and `System.Console.ReadLine()` from Chapter 1. This time we look at them as examples of method calls in general instead of looking at the specifics of printing and retrieving data from the console. Listing 5.2 shows each of the three methods in use.

Listing 5.2: A Simple Method Call

```
class HeyYou
{
  static void Main()
  {
      string firstName;
      string lastName;

      System.Console.WriteLine("Hey you!");

      System.Console.Write("Enter your first name: ");

      firstName = System.Console.ReadLine();
      System.Console.Write("Enter your last name: ");
      lastName = System.Console.ReadLine();
      System.Console.WriteLine(
          $"Your full name is { firstName } { lastName }.");
  }
}
```

The parts of the method call include the method name, argument list, and returned value. A fully qualified method name includes a namespace, type name, and method name; a period separates each part of a fully qualified method name. As we will see, methods are often called with only a part of their fully qualified name.

Namespaces

Namespaces are a categorization mechanism for grouping all types related to a particular area of functionality. Namespaces are hierarchical and can have arbitrarily many levels in the hierarchy, though namespaces with more than half a dozen levels are rare. Typically the hierarchy begins with a company name, and then a product name, and then the functional area. For example, in `Microsoft.Win32.Networking`, the outermost namespace is `Microsoft`, which contains an inner namespace `Win32`, which in turn contains an even more deeply nested `Networking` namespace.

Namespaces are primarily used to organize types by area of functionality so that they can be more easily found and understood. However, they can also be used to avoid type name collisions. For example, the compiler can distinguish between two types with the name `Button` as long as each type has a different namespace. Thus you can disambiguate types `System.Web.UI.WebControls.Button` and `System.Windows.Controls.Button`.

In Listing 5.2, the `Console` type is found within the `System` namespace. The `System` namespace contains the types that enable the programmer to perform many fundamental programming activities. Almost all C# programs use types within the `System` namespace. Table 5.1 provides a listing of other common namespaces.

Begin 4.0

Table 5.1: Common Namespaces

Namespace	Description
`System`	Contains the fundamental types and types for conversion between types, mathematics, program invocation, and environment management.
`System.Collections.Generics`	Contains strongly typed collections that use generics.
`System.Data`	Contains types used for working with databases.
`System.Drawing`	Contains types for drawing to the display device and working with images.
`System.IO`	Contains types for working with directories and manipulating, loading, and saving files.
`System.Linq`	Contains classes and interfaces for querying data in collections using a Language Integrated Query.
`System.Text`	Contains types for working with strings and various text encodings, and for converting between those encodings.
`System.Text.RegularExpressions`	Contains types for working with regular expressions.
`System.Threading`	Contains types for multithreaded programming.
`System.Threading.Tasks`	Contains types for task-based asynchrony.
`System.Web`	Contains types that enable browser-to-server communication, generally over HTTP. The functionality within this namespace is used to support ASP.NET.

continues

TABLE 5.1: Common Namespaces (*continued*)

Namespace	Description
`System.Windows`	Contains types for creating rich user interfaces starting with .NET 3.0 using a UI technology called Windows Presentation Framework (WPF) that leverages Extensible Application Markup Language (XAML) for declarative design of the UI.
`System.Xml`	Contains standards-based support for XML processing.

End 4.0

It is not always necessary to provide the namespace when calling a method. For example, if the call expression appears in a type in the same namespace as the called method, the compiler can infer the namespace to be the namespace that contains the type. Later in this chapter, you will see how the `using` directive eliminates the need for a namespace qualifier as well.

Guidelines

DO use PascalCasing for namespace names.

CONSIDER organizing the directory hierarchy for source code files to match the namespace hierarchy.

Type Name

Calls to static methods require the type name qualifier as long as the target method is not within the same type.[1] (As discussed later in the chapter, a `using static` directive allows you to omit the type name.) For example, a call expression of `Console.WriteLine()` found in the method `HelloWorld.Main()` requires the type, `Console`, to be specified. However, just as with the namespace, C# allows the omission of the type name from a method call whenever the method is a member of the type containing the call expression. (Examples of method calls such as this appear in Listing 5.4.) The type name is unnecessary in such cases because the compiler

1. Or base class.

infers the type from the location of the call. If the compiler can make no such inference, the name must be provided as part of the method call.

At their core, types are a means of grouping together methods and their associated data. For example, `Console` is the type that contains the `Write()`, `WriteLine()`, and `ReadLine()` methods (among others). All of these methods are in the same *group* because they belong to the `Console` type.

Scope

In Chapter 4, you learned that the *scope* of a program element is the region of text in which it can be referred to by its unqualified name. A call that appears inside a type declaration to a method declared in that type does not require the type qualifier because the method is in scope throughout its containing type. Similarly, a type is in scope throughout the namespace that declares it; therefore, a method call that appears in a type in a particular namespace need not specify that namespace in the method call name.

Method Name

Every method call contains a method name, which might or might not be qualified with a namespace and type name, as we have discussed. After the method name comes the argument list, which is a parenthesized, comma-separated list of the values that correspond to the parameters of the method.

Parameters and Arguments

A method can take any number of parameters, and each parameter is of a specific data type. The values that the caller supplies for parameters are called the **arguments**; every argument must correspond to a particular parameter. For example, the following method call has three arguments:

```
System.IO.File.Copy(
    oldFileName, newFileName, false)
```

The method is found on the class `File`, which is located in the namespace `System.IO`. It is declared to have three parameters, with the first and second being of type `string` and the third being of type `bool`. In this example, we use variables (`oldFileName` and `newFileName`) of type `string` for the old and new filenames, and then specify `false` to indicate that the copy should fail if the new filename already exists.

Method Return Values

In contrast to System.Console.WriteLine(), the method call System.Console.ReadLine() in Listing 5.2 does not have any arguments because the method is declared to take no parameters. However, this method happens to have a **method return value**. The method return value is a means of transferring results from a called method back to the caller. Because System.Console.ReadLine() has a return value, it is possible to assign the return value to the variable firstName. In addition, it is possible to pass this method return value itself as an argument to another method call, as shown in Listing 5.3.

Listing 5.3: Passing a Method Return Value as an Argument to Another Method Call

```
class Program
{
  static void Main()
  {
      System.Console.Write("Enter your first name: ");
      System.Console.WriteLine("Hello {0}!",
          System.Console.ReadLine());
  }
}
```

Instead of assigning the returned value to a variable and then using that variable as an argument to the call to System.Console.WriteLine(), Listing 5.3 calls the System.Console.ReadLine() method within the call to System.Console.WriteLine(). At execution time, the System.Console.ReadLine() method executes first, and its return value is passed directly into the System.Console.WriteLine() method, rather than into a variable.

Not all methods return data. Both versions of System.Console.Write() and System.Console.WriteLine() are examples of such methods. As you will see shortly, these methods specify a return type of void, just as the HelloWorld declaration of Main returned void.

Statement versus Method Call

Listing 5.3 provides a demonstration of the difference between a statement and a method call. Although System.Console.WriteLine("Hello {0}!", System.Console.ReadLine()); is a single statement, it contains two method calls. A statement often contains one or more expressions, and in this example, two of those expressions are method calls. Therefore, method calls form parts of statements.

Although coding multiple method calls in a single statement often reduces the amount of code, it does not necessarily increase the readability and seldom offers a significant performance advantage. Developers should favor readability over brevity.

> **NOTE**
>
> In general, developers should favor readability over brevity. Readability is critical to writing code that is self-documenting and therefore more maintainable over time.

Declaring a Method

Begin 6.0

This section expands on the explanation of declaring a method to include parameters or a return type. Listing 5.4 contains examples of these concepts, and Output 5.1 shows the results.

Listing 5.4: Declaring a Method

```
class IntroducingMethods
{
  public static void Main()
  {
      string firstName;
      string lastName;
      string fullName;
      string initials;

      System.Console.WriteLine("Hey you!");

      firstName = GetUserInput("Enter your first name: ");
      lastName = GetUserInput("Enter your last name: ");

      fullName = GetFullName(firstName, lastName);
      initials = GetInitials(firstName, lastName);
      DisplayGreeting(fullName, initials);
  }

  static string GetUserInput(string prompt)
  {
      System.Console.Write(prompt);
      return System.Console.ReadLine();
  }

  static string GetFullName(  // C# 6.0 expression bodied method
      string firstName, string lastName) =>
          $"{ firstName } { lastName }";
```

```
    static void DisplayGreeting(string fullName, string initials)
    {
        System.Console.WriteLine(
            $"Hello { fullName }! Your initials are { initials }");
        return;
    }

    static string GetInitials(string firstName, string lastName)
    {
        return $"{ firstName[0] }. { lastName[0] }.";
    }
}
```

Output 5.1

End 6.0

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Hello Inigo Montoya! Your initials are I. M.
```

Five methods are declared in Listing 5.4. From `Main()` the code calls `GetUserInput()`, followed by a call to `GetFullName()` and `GetInitials()`. All of the last three methods return a value and take arguments. In addition, the listing calls `DisplayGreeting()`, which doesn't return any data. No method in C# can exist outside the confines of an enclosing type; in this case, the enclosing type is the `IntroducingMethods` class. Even the `Main` method examined in Chapter 1 must be within a type.

Language Contrast: C++/Visual Basic—Global Methods

C# provides no global method support; everything must appear within a type declaration. This is why the `Main()` method was marked as `static`—the C# equivalent of a C++ global and Visual Basic "shared" method.

BEGINNER TOPIC

Refactoring into Methods

Moving a set of statements into a method instead of leaving them inline within a larger method is a form of **refactoring**. Refactoring reduces code duplication, because you can call the method from multiple places instead of duplicating the code. Refactoring also increases code readability. As part of the coding process, it is a best practice to continually review your code and look for opportunities to refactor. This involves looking for blocks of code that are difficult to understand at a glance and moving them into a method with a name that clearly defines the code's behavior. This practice is often preferred over commenting a block of code, because the method name serves to describe what the implementation does.

For example, the `Main()` method that is shown in Listing 5.4 results in the same behavior as does the `Main()` method that is shown in Listing 1.16 in Chapter 1. Perhaps even more noteworthy is that although both listings are trivial to follow, Listing 5.4 is easier to grasp at a glance by just viewing the `Main()` method and not worrying about the details of each called method's implementation.

Visual Studio allows you to right-click on a block of code within a method and click **Quick Actions and Refactorings…** (Ctrl+.) to extract the block into its own method, automatically inserting code to call the new method from the original location.

Formal Parameter Declaration

Consider the declarations of the `DisplayGreeting()`, `GetFullName()`, and the `GetInitials()` methods. The text that appears between the parentheses of a method declaration is the **formal parameter list**. (As we will see when we discuss generics, methods may also have a **type parameter list**. When it is clear from the context which kind of parameters we are discussing, we simply refer to them as *parameters* in a *parameter list*.) Each parameter in the parameter list includes the type of the parameter along with the parameter name. A comma separates each parameter in the list.

Behaviorally, most parameters are virtually identical to local variables, and the naming convention of parameters follows accordingly. Therefore, parameter names use camelCase. Also, it is not possible to declare a local variable (a variable declared inside a method) with the same name as a parameter of the containing method, because this would create two *local variables* of the same name.

Guidelines

DO use camelCasing for parameter names.

Method Return Type Declaration

In addition to `GetUserInput()`, `GetFullName()`, and the `GetInitials()` methods requiring parameters to be specified, each of these methods includes a **method return type**. You can tell that a method returns a value because a data type appears immediately before the method name in the method declaration. Each of these method examples specifies a `string` return type. Unlike with parameters, of which there can be any number, only one method return type is allowable.

As with `GetUserInput()` and `GetInitials()`, methods with a return type almost always contain one or more `return` statements that return control to the caller. A `return` statement consists of the `return` keyword followed by an expression that computes the value the method is returning. For example, the `GetInitials()` method's `return` statement is `return $"{ firstName[0] }. { lastName[0] }.";`. The expression (an interpolated string in this case) following the `return` keyword must be compatible with the stated return type of the method.

If a method has a return type, the block of statements that makes up the body of the method must not have an *unreachable end point*. That is, there must be no way for control to "fall off the end" of a method without returning a value. Often the easiest way to ensure that this condition is met is to make the last statement of the method a `return` statement. However, `return` statements can appear in locations other than at the end of a method implementation. For example, an `if` or `switch` statement in a method implementation could include a `return` statement within it; see Listing 5.5 for an example.

LISTING 5.5: A return Statement before the End of a Method

```
class Program
{
  static bool MyMethod()
  {
      string command = ObtainCommand();
      switch(command)
      {
          case "quit":
              return false;
          // ... omitted, other cases
          default:
              return true;
      }
  }
}
```

(Note that a `return` statement transfers control out of the `switch`, so no `break` statement is required to prevent illegal fall-through in a switch section that ends with a `return` statement.)

In Listing 5.5, the last statement in the method is not a `return` statement; it is a `switch` statement. However, the compiler can deduce that every possible code path through the method results in a `return`, so that the end point of the method is not reachable. Thus this method is legal even though it does not end with a `return` statement.

If particular code paths include unreachable statements following the `return`, the compiler will issue a warning that indicates the additional statements will never execute.

Though C# allows a method to have multiple `return` statements, code is generally more readable and easier to maintain if there is a single exit location rather than having multiple returns sprinkled through various code paths of the method.

Specifying `void` as a return type indicates that there is no return value from the method. As a result, a call to the method may not be assigned to a variable or used as a parameter type at the call site. A `void` method call may be used only as a statement. Furthermore, within the body of the method the `return` statement becomes optional, and when it is specified, there must be no value following the `return` keyword. For example, the return of `Main()` in Listing 5.4 is `void`, and there is no `return` statement within the method. However, `DisplayGreeting()` includes an (optional) `return` statement that is not followed by any returned result.

Begin 7.0

Although, technically, a method can have only one return type, the return type could be a tuple. As a result, starting with C# 7.0, it is possible to return multiple values packaged as a tuple using C# tuple syntax. For example, you could declare a `GetName()` method, as shown in Listing 5.6.

LISTING 5.6: Returning Multiple Values Using a Tuple

```
class Program
{
  static string GetUserInput(string prompt)
  {
      System.Console.Write(prompt);
      return System.Console.ReadLine();
  }
  static (string First, string Last) GetName()
  {
      string firstName, lastName;
      firstName = GetUserInput("Enter your first name: ");
```

```
        lastName = GetUserInput("Enter your last name: ");
        return (firstName, lastName);
    }
    static public void Main()
    {
        (string First, string Last) name = GetName();
        System.Console.WriteLine($"Hello { name.First } { name.Last }!");
    }
}
```

End 7.0

Technically, we are still returning only one data type, a `ValueTuple<string, string>`. However, effectively, you can return any (preferably reasonable) number you like.

Begin 6.0

Expression Bodied Methods

To support the simplest of method declarations without the formality of a method body, C# 6.0 introduced **expression bodied methods**, which are declared using an expression rather than a full method body. Listing 5.4's `GetFullName()` method provides an example of the expression bodied method:

```
static string GetFullName( string firstName, string lastName) =>
```

In place of the curly brackets typical of a method body, an expression bodied method uses the "goes to" operator (fully introduced in Chapter 13), for which the resulting data type must match the return type of the method. In other words, even though there is no explicit `return` statement in the expression bodied method implementation, it is still necessary that the return type from the expression match the method declaration's return type.

Expression bodied methods are syntactic shortcuts to the fuller method body declaration. As such, their use should be limited to the simplest of method implementations—generally expressible on a single line.

End 6.0

Language Contrast: C++—Header Files

Unlike in C++, C# classes never separate the implementation from the declaration. In C#, there is no header (`.h`) file or implementation (`.cpp`) file. Instead, declaration and implementation appear together in the same file. (C# does support an advanced feature called *partial methods*, in which the method's defining declaration is separate from its implementation, but for the purposes of this chapter, we consider only nonpartial methods.) The lack of a separate declaration and implementation in C# removes the requirement to maintain redundant declaration information in two places found in languages that have separate header and implementation files, such as C++.

BEGINNER TOPIC

Namespaces

As described earlier, **namespaces** are an organizational mechanism for categorizing and grouping together related types. Developers can discover related types by examining other types within the same namespace as a familiar type. Additionally, through namespaces, two or more types may have the same name as long as they are disambiguated by different namespaces.

The using Directive

Fully qualified namespace names can become quite long and unwieldy. It is possible, however, to import all the types from one or more namespaces into a file so that they can be used without full qualification. To achieve this, the C# programmer includes a `using` directive, generally at the top of the file. For example, in Listing 5.7, `Console` is not prefixed with `System`. The namespace may be omitted because of the `using System` directive that appears at the top of the listing.

Listing 5.7: using Directive Example

```
// The using directive imports all types from the
// specified namespace into the entire file
using System;

class HelloWorld
{
  static void Main()
  {
      // No need to qualify Console with System
      // because of the using directive above
      Console.WriteLine("Hello, my name is Inigo Montoya");
  }
}
```

The results of Listing 5.7 appear in Output 5.2.

Output 5.2

```
Hello, my name is Inigo Montoya
```

A using directive such as using System does not enable you to omit System from a type declared within a child namespace of System. For example, if your code accessed the StringBuilder type from the System.Text namespace, you would have to either include an additional using System.Text; directive or fully qualify the type as System.Text.StringBuilder, not just Text.StringBuilder. In short, a using directive does not import types from any **nested namespaces**. Nested namespaces, which are identified by the period in the namespace, always need to be imported explicitly.

Language Contrast: Java—Wildcards in the import Directive

Java enables importing namespaces using a wildcard such as the following:

```
import javax.swing.*;
```

In contrast, C# does not support a wildcard using directive but instead requires each namespace to be imported explicitly.

Language Contrast: Visual Basic .NET—Project Scope Imports Directive

Unlike C#, Visual Basic .NET supports the ability to specify a using directive equivalent, Imports, for an entire project rather than for just a specific file. In other words, Visual Basic .NET provides a command-line version of the using directive that will span an entire compilation.

Frequent use of types within a particular namespace implies that the addition of a using directive for that namespace is a good idea, instead of fully qualifying all types within the namespace. Accordingly, almost all C# files include the using System directive at the top. Throughout the remainder of this book, code listings often omit the using System directive. Other namespace directives are included explicitly, however.

One interesting effect of the using System directive is that the string data type can be identified with varying case: String or string. The former version relies on the using System directive, and the latter uses the string keyword. Both are valid C# references to the System.String data type, and the resultant Common Intermediate Language (CIL) code is unaffected by which version is chosen.[2]

ADVANCED TOPIC

Nested using Directives

Not only can you have `using` directives at the top of a file, but you can also include them at the top of a namespace declaration. For example, if a new namespace, `EssentialCSharp`, were declared, it would be possible to add a `using` declarative at the top of the namespace declaration (see Listing 5.8).

Listing 5.8: Specifying the using Directive inside a Namespace Declaration

```
namespace EssentialCSharp
{
  using System;

  class HelloWorld
  {
    static void Main()
    {
        // No need to qualify Console with System
        // because of the using directive above
        Console.WriteLine("Hello, my name is Inigo Montoya");
    }
  }
}
```

The results of Listing 5.8 appear in Output 5.3.

Output 5.3

```
Hello, my name is Inigo Montoya
```

The difference between placing the `using` directive at the top of a file and placing it at the top of a namespace declaration is that the directive is active only within the namespace declaration. If the code includes a new namespace declaration above or below the `EssentialCSharp` declaration,

2. I prefer the `string` keyword, but whichever representation a programmer selects, the code within a project ideally should be consistent.

the `using System` directive within a different namespace would not be active. Code seldom is written this way, especially given the standard practice of providing a single type declaration per file.

Begin 6.0

using static Directive

The `using` directive allows you to abbreviate a type name by omitting the namespace portion of the name—such that just the type name can be specified for any type within the stated namespace. In contrast, the `using static` directive allows you to omit both the namespace and the type name from any member of the stated type. A `using static System.Console` directive, for example, allows you to specify `WriteLine()` rather than the fully qualified method name of `System.Console.WriteLine()`. Continuing with this example, we can update Listing 5.2 to leverage the `using static System.Console` directive to create Listing 5.9.

Listing 5.9: using static Directive

```
using static System.Console;

class HeyYou
{
  static void Main()
  {
      string firstName;
      string lastName;

      WriteLine("Hey you!");

      Write("Enter your first name: ");

      firstName = ReadLine();
      Write("Enter your last name: ");
      lastName = ReadLine();
      WriteLine(
          $"Your full name is { firstName } { lastName }.");
  }
}
```

In this case, there is no loss of readability of the code: `WriteLine()`, `Write()`, and `ReadLine()` all clearly relate to a console directive. In fact, one could argue that the resulting code is simpler and therefore clearer than before.

However, sometimes this is not the case. For example, if your code uses classes that have overlapping behavior names, such as an `Exists()` method on a file and an `Exists()` method on a directory, then perhaps a `using static` directive would reduce clarity when you invoke `Exists()`. Similarly, if the class you were writing had its own members with overlapping behavior names—for example, `Display()` and `Write()`—then perhaps clarity would be lost to the reader.

This ambiguity would not be allowed by the compiler. If two members with the same signature were available (through either `using static` directives or separately declared members), any invocation of them that was ambiguous would result in a compile error.

End 6.0

Aliasing

The `using` directive also allows **aliasing** a namespace or type. An alias is an alternative name that you can use within the text to which the `using` directive applies. The two most common reasons for aliasing are to disambiguate two types that have the same name and to abbreviate a long name. In Listing 5.10, for example, the `CountDownTimer` alias is declared as a means of referring to the type `System.Timers.Timer`. Simply adding a `using System.Timers` directive will not sufficiently enable the code to avoid fully qualifying the `Timer` type. The reason is that `System.Threading` also includes a type called `Timer`; therefore, using just `Timer` within the code will be ambiguous.

Listing 5.10: Declaring a Type Alias

```
using System;
using System.Threading;
using CountDownTimer = System.Timers.Timer;

class HelloWorld
{
  static void Main()
  {
      CountDownTimer timer;

      // ...
  }
}
```

Listing 5.10 uses an entirely new name, `CountDownTimer`, as the alias. It is possible, however, to specify the alias as `Timer`, as shown in Listing 5.11.

Listing 5.11: Declaring a Type Alias with the Same Name

```
using System;
using System.Threading;

// Declare alias Timer to refer to System.Timers.Timer to
// avoid code ambiguity with System.Threading.Timer
using Timer = System.Timers.Timer;

class HelloWorld
{
  static void Main()
  {
      Timer timer;

      // ...
  }
}
```

Because of the alias directive, "Timer" is not an ambiguous reference. Furthermore, to refer to the `System.Threading.Timer` type, you will have to either qualify the type or define a different alias.

Returns and Parameters on `Main()`

So far, declaration of an executable's `Main()` method has been the simplest declaration possible. You have not included any parameters or non-`void` return type in your `Main()` method declarations. However, C# supports the ability to retrieve the command-line arguments when executing a program, and it is possible to return a status indicator from the `Main()` method.

The runtime passes the command-line arguments to `Main()` using a single `string` array parameter. All you need to do to retrieve the parameters is to access the array, as demonstrated in Listing 5.12. The purpose of this program is to download a file whose location is given by a URL. The first command-line argument identifies the URL, and the second argument is the filename to which to save the file. The listing begins with a `switch` statement that evaluates the number of parameters (`args.Length`) as follows:

1. If there are not two parameters, display an error indicating that it is necessary to provide the URL and filename.
2. The presence of two arguments indicates the user has provided both the URL of the resource and the download target filename.

Listing 5.12: Passing Command-Line Arguments to Main

```
using System;
using System.IO;
using System.Net.Http;

class Program
{
  static int Main(string[] args)
  {
      int result;
      switch (args.Length)
      {
          default:
              // Exactly two arguments must be specified; give an error
              Console.WriteLine(
                  "ERROR:  You must specify the "
                  + "URL and the file name");
              Console.WriteLine(
                  "Usage: Downloader.exe <URL> <TargetFileName>");
              result = 1;
              break;
          case 2:
              WebClient webClient = new WebClient();
              webClient.DownloadFile(args[0], args[1]);
              result = 0;
              break;
      }
      return result;
  }


}
```

The results of Listing 5.12 appear in Output 5.4.

Output 5.4

```
>Downloader.exe
ERROR:  You must specify the URL to be downloaded
Downloader.exe <URL> <TargetFileName>
```

If you were successful in calculating the target filename, you would use it to save the downloaded file. Otherwise, you would display the help text. The `Main()` method also returns an `int` rather than a `void`. This is optional for a `Main()` declaration, but if it is used, the program can return a status code to a caller (such as a script or a batch file). By convention, a return other than zero indicates an error.

Although all command-line arguments can be passed to `Main()` via an array of strings, sometimes it is convenient to access the arguments from inside a method other than `Main()`. The `System.Environment.GetCommandLineArgs()` method returns the command-line arguments array in the same form that `Main(string[] args)` passes the arguments into `Main()`.

ADVANCED TOPIC

Disambiguate Multiple `Main()` Methods

If a program includes two classes with `Main()` methods, it is possible to specify which one to use as the entry point. In Visual Studio, right-clicking on the project from Solution Explorer and selecting **Properties** provides a user interface on top of the project file. By selecting the **Application** tab on the left, you can edit the Startup Object and select which type's `Main()` method will start the program. On the command line, you can specify the same value, setting the `StartupObject` property when running a build. For example:

```
dotnet build /p:StartupObject=AddisonWesley.Program2
```

where `AddisonWesley.Program2` is the namespace and class that contains the selected `Main()` method.

BEGINNER TOPIC

Call Stack and Call Site

As code executes, methods call more methods, which in turn call additional methods, and so on. In the simple case of Listing 5.4, `Main()` calls `GetUserInput()`, which in turn calls `System.Console.ReadLine()`, which in turn calls even more methods internally. Every time a new method is invoked, the runtime creates an *activation frame* that contains information about the arguments passed to the new call, the local variables of the new call, and information about where control should resume when the new method returns. The set of calls within calls within calls, and so on, produces a series of activation frames that is termed the **call stack**.[3]

3. Except for async or iterator methods, which move their activator records onto the heap.

As program complexity increases, the call stack generally gets larger and larger as each method calls another method. As calls complete, however, the call stack shrinks until another method is invoked. The process of removing activation frames from the call stack is termed **stack unwinding**. Stack unwinding always occurs in the reverse order of the method calls. When the method completes, execution returns to the **call site**—that is, the location from which the method was invoked.

Advanced Method Parameters

So far this chapter's examples have returned data via the method return value. This section demonstrates how methods can return data via their method parameters and how a method may take a variable number of arguments.

Value Parameters

Arguments to method calls are usually **passed by value**, which means the value of the argument expression is copied into the target parameter. For example, in Listing 5.13, the value of each variable that `Main()` uses when calling `Combine()` will be copied into the parameters of the `Combine()` method. Output 5.5 shows the results of this listing.

Listing 5.13: Passing Variables by Value

```
class Program
{
  static void Main()
  {
      // ...
      string fullName;
      string driveLetter = "C:";
      string folderPath  = "Data";
      string fileName    = "index.html";

      fullName = Combine(driveLetter, folderPath, fileName);

      Console.WriteLine(fullName);
      // ...
  }

  static string Combine(
      string driveLetter, string folderPath, string fileName)
  {
      string path;
```

```
            path = string.Format("{1}{0}{2}{0}{3}",
                System.IO.Path.DirectorySeparatorChar,
                driveLetter, folderPath, fileName);
            return path;
        }
    }
```

Output 5.5

```
C:\Data\index.html
```

Even if the `Combine()` method assigns `null` to `driveLetter`, `folderPath`, and `fileName` before returning, the corresponding variables within `Main()` will maintain their original values because the variables are copied when calling a method. When the call stack unwinds at the end of a call, the copied data is thrown away.

Beginner Topic

Matching Caller Variables with Parameter Names

In Listing 5.13, the variable names in the caller exactly matched the parameter names in the called method. This matching is provided simply for readability purposes; whether names match is irrelevant to the behavior of the method call. The parameters of the called method and the local variables of the calling method are found in different declaration spaces and have nothing to do with each other.

Advanced Topic

Reference Types versus Value Types

For the purposes of this section, it is inconsequential whether the parameter passed is a value type or a reference type. Rather, the important issue is whether the called method can write a value into the caller's original variable. Since a copy of the caller variable's value is made, the caller's variable cannot be reassigned. Nevertheless, it is helpful to understand the difference between a variable that contains a value type and a variable that contains a reference type.

The value of a reference type variable is, as the name implies, a reference to the location where the data associated with the object is stored. How the runtime chooses to represent the value of a reference type variable is an implementation detail of the runtime; typically it is represented as the address of the memory location in which the object's data is stored, but it need not be.

If a reference type variable is passed by value, the reference itself is copied from the caller to the method parameter. As a result, the target method cannot update the caller variable's value, but it may update the data referred to by the reference.

Alternatively, if the method parameter is a value type, the value itself is copied into the parameter, and changing the parameter in the called method will not affect the original caller's variable.

Reference Parameters (`ref`)

Consider Listing 5.14, which calls a function to swap two values, and Output 5.6, which shows the results.

Listing 5.14: Passing Variables by Reference

```
class Program
{
  static void Main()
  {
      // ...
      string first = "hello";
      string second = "goodbye";
      Swap(ref first, ref second);

      Console.WriteLine(
          $@"first = ""{ first }"", second = ""{ second }""" );
      // ...
  }

  static void Swap(ref string x, ref string y)
  {
      string temp = x;
      x = y;
      y = temp;
  }
}
```

Output 5.6

```
first = "goodbye", second = "hello"
```

The values assigned to `first` and `second` are successfully switched. To do this, the variables are **passed by reference**. The obvious difference between the call to `Swap()` and Listing 5.13's call to `Combine()` is the inclusion of the keyword `ref` in front of the parameter's data type. This keyword changes the call such that the variables used as arguments are passed by reference, so the called method can update the original caller's variables with new values.

When the called method specifies a parameter as `ref`, the caller is required to supply a variable, not a value, as an argument and to place `ref` in front of the variables passed. In so doing, the caller explicitly recognizes that the target method could reassign the values of the variables associated with any `ref` parameters it receives. Furthermore, it is necessary to initialize any local variables passed as `ref` because target methods could read data from `ref` parameters without first assigning them. In Listing 5.14, for example, `temp` is assigned the value of `first`, assuming that the variable passed in `first` was initialized by the caller. Effectively, a `ref` parameter is an alias for the variable passed. In other words, it is essentially giving a parameter name to an existing variable, rather than creating a new variable and copying the value of the argument into it.

Begin 7.0

Output Parameters (out)

As mentioned earlier, a variable used as a `ref` parameter must be assigned before it is passed to the called method, because the called method might read from the variable. The "swap" example given previously must read and write from both variables passed to it. However, it is often the case that a method that takes a reference to a variable intends to write to the variable but not to read from it. In such cases, clearly it could be safe to pass an uninitialized local variable by reference.

To achieve this, code needs to decorate parameter types with the keyword `out`. This is demonstrated in the `TryGetPhoneButton()` method in Listing 5.15, which returns the phone button corresponding to a character.

Listing 5.15: Passing Variables Out Only

```
class ConvertToPhoneNumber
{
  static int Main(string[] args)
  {
      if(args.Length == 0)
```

```
        {
            Console.WriteLine(
                "ConvertToPhoneNumber.exe <phrase>");
            Console.WriteLine(
                "'_' indicates no standard phone button");
            return 1;
        }
        foreach(string word in args)
        {
            foreach(char character in word)
            {
                if(TryGetPhoneButton(character, out char button))
                {
                    Console.Write(button);
                }
                else
                {
                    Console.Write('_');
                }
            }
        }
        Console.WriteLine();
        return 0;
    }

    static bool TryGetPhoneButton(char character, out char button)
    {
        bool success = true;
        switch( char.ToLower(character) )
        {
            case '1':
                button = '1';
                break;
            case '2': case 'a': case 'b': case 'c':
                button = '2';
                break;

            // ...

            case '-':
                button = '-';
                break;
            default:
                  // Set the button to indicate an invalid value
                  button = '_';
                success = false;
                break;
        }
        return success;
    }
}
```

7.0

Output 5.7 shows the results of Listing 5.15.

Output 5.7

```
>ConvertToPhoneNumber.exe CSharpIsGood
274277474663
```

In this example, the `TryGetPhoneButton()` method returns `true` if it can successfully determine the `character`'s corresponding phone button. The function also returns the corresponding button by using the `button` parameter, which is decorated with `out`.

7.0

An `out` parameter is functionally identical to a `ref` parameter; the only difference is which requirements the language enforces regarding how the aliased variable is read from and written to. Whenever a parameter is marked with `out`, the compiler checks that the parameter is set for all code paths within the method that return normally (i.e., the code paths that do not throw an exception). If, for example, the code does not assign `button` a value in some code path, the compiler will issue an error indicating that the code didn't initialize `button`. Listing 5.15 assigns `button` to the underscore character because even though it cannot determine the correct phone button, it is still necessary to assign a value.

A common coding error when working with `out` parameters is to forget to declare the `out` variable before you use it. Starting with C# 7.0, it is possible to declare the `out` variable inline when invoking the function. Listing 5.15 uses this feature with the statement `TryGetPhoneButton(character, out char button)` without ever declaring the `button` variable beforehand. Prior to C# 7.0, it would be necessary to first declare the `button` variable and then invoke the function with `TryGetPhoneButton(character, out button)`.

Another C# 7.0 feature is the ability to discard an `out` parameter entirely. If, for example, you simply wanted to know whether a character was a valid phone button but not actually return the numeric value, you could discard the `button` parameter using an underscore: `TryGetPhoneButton(character, out _)`.

Prior to C# 7.0's tuple syntax, a developer of a method might declare one or more `out` parameters to get around the restriction that a method may have only one return type; a method that needs to return two values can do so by returning one value normally, as the return value of the method, and a second value by writing it into an aliased variable passed as an `out` parameter. Although this pattern is both common and legal, there are usually better ways to achieve that aim. For example, if you are considering returning two or more values from a method and C# 7.0 is available, it is likely preferable to use C# 7.0 tuple syntax. Prior to that, consider writing two methods, one for each value, or still using the `System.ValueTuple` type but without C# 7.0 syntax.

7.0

> **NOTE**
>
> Each and every normal code path must result in the assignment of all `out` parameters.

Read-Only Pass by Reference (`in`)

Begin 7.2

In C# 7.2, support was added for passing a value type by reference that was read only. Rather than passing the value type to a function so that it could be changed, read-only pass by reference was added: It allows the value type to be passed by reference so that not only copy of the value type occurs but, in addition, the invoked method cannot change the value type. In other words, the purpose of the feature is to reduce the memory copied when passing a value while still identifying it as read only, thus improving the performance. This syntax is to add an `in` modifier to the parameter. For example:

```
int Method(in int number) { ... }
```

With the `in` modifier, any attempts to reassign `number` (`number++`, for example) will result in a compile error indicating that `number` is read only.

End 7.2

Return by Reference

Another C# 7.0 addition is support for returning a reference to a variable. Consider, for example, a function that returns the first pixel in an image that is associated with red-eye, as shown in Listing 5.16.

LISTING 5.16: `ref` Return and `ref` Local Declaration

```
// Returning a reference
public static ref byte FindFirstRedEyePixel(byte[] image)
```

```
{
  // Do fancy image detection perhaps with machine learning
  for (int counter = 0; counter < image.Length; counter++)
  {
    if(image[counter] == (byte)ConsoleColor.Red)
    {
      return ref image[counter];
    }
  }
  throw new InvalidOperationException("No pixels are red.");
}
public static void Main()
{
  byte[] image = new byte[254];
  // Load image
  int index = new Random().Next(0, image.Length - 1);
  image[index] =
      (byte)ConsoleColor.Red;
  System.Console.WriteLine(
      $"image[{index}]={(ConsoleColor)image[index]}");
  // ...

  // Obtain a reference to the first red pixel
  ref byte redPixel = ref FindFirstRedEyePixel(image);
  // Update it to be Black
  redPixel = (byte)ConsoleColor.Black;
  System.Console.WriteLine(
      $"image[{index}]={(ConsoleColor)image[redPixel]}");
}
```

7.0

By returning a reference to the variable, the caller is then able to update the pixel to a different color, as shown in the highlighted lines of Listing 5.16. Checking for the update via the array shows that the value is now black.

There are two important restrictions on return by reference, both due to object lifetime: (1) Object references shouldn't be garbage collected while they're still referenced, and (2) they shouldn't consume memory when they no longer have any references. To enforce these restrictions, you can only return the following from a reference-returning function:

- References to fields or array elements
- Other reference-returning properties or functions
- References that were passed in as parameters to the by-reference-returning function

For example, `FindFirstRedEyePixel()` returns a reference to an item in the image array, which was a parameter to the function. Similarly, if the image was stored as a field within the class, you could return the field by reference:

```
byte[] _Image;
public ref byte[] Image { get {  return ref _Image; } }
```

In addition, `ref` locals are initialized to refer to a particular variable and can't be modified to refer to a different variable.

There are several return-by-reference characteristics of which to be cognizant:

- If you're returning a reference, you obviously must return it. This means, therefore, that in the example in Listing 5.16, even if no red-eye pixel exists, you still need to return a reference byte. The only workaround would be to throw an exception. In contrast, the by-reference parameter approach allows you to leave the parameter unchanged and return a `bool` indicating success. In many cases, this might be preferable.
- When declaring a reference local variable, initialization is required. This involves assigning it a `ref` return from a function or a reference to a variable:

  ```
  ref string text;  // Error
  ```

- Although it's possible in C# 7.0 to declare a reference local variable, declaring a field of type `ref` isn't allowed:

  ```
  class Thing { ref string _Text;  /* Error */ }
  ```

- You can't declare a by-reference type for an auto-implemented property:

  ```
  class Thing { ref string Text { get;set; }  /* Error */ }
  ```

- Properties that return a reference are allowed:

  ```
  class Thing { string _Text = "Inigo Montoya";
  ref string Text { get { return ref _Text; } } }
  ```

- A reference local variable can't be initialized with a value (such as `null` or a constant). It must be assigned from a by-reference-returning member or a local variable, field, or array element:

  ```
  ref int number = 42;  // ERROR
  ```

End 7.0

Parameter Arrays (params)

In the examples so far, the number of arguments that must be passed has been fixed by the number of parameters declared in the target method declaration. However, sometimes it is convenient if the number of arguments may vary. Consider the Combine() method from Listing 5.13. In that method, you passed the drive letter, folder path, and filename. What if the path had more than one folder, and the caller wanted the method to join additional folders to form the full path? Perhaps the best option would be to pass an array of strings for the folders. However, this would make the calling code a little more complex, because it would be necessary to construct an array to pass as an argument.

To make it easier on the callers of such a method, C# provides a keyword that enables the number of arguments to vary in the calling code instead of being set by the target method. Before we discuss the method declaration, observe the calling code declared within Main(), as shown in Listing 5.17.

Listing 5.17: Passing a Variable Parameter List

```
using System;
using System.IO;
class PathEx
{
  static void Main()
  {
      string fullName;

      // ...

      // Call Combine() with four arguments
      fullName = Combine(
          Directory.GetCurrentDirectory(),
          "bin", "config", "index.html");
      Console.WriteLine(fullName);

      // ...

      // Call Combine() with only three arguments
      fullName = Combine(
          Environment.SystemDirectory,
          "Temp", "index.html");
      Console.WriteLine(fullName);

      // ...
```

```
        // Call Combine() with an array
        fullName = Combine(
            new string[] {
                "C:\\", "Data",
                "HomeDir", "index.html"} );
        Console.WriteLine(fullName);
        // ...
    }

    static string Combine(params string[] paths)
    {
        string result = string.Empty;
        foreach (string path in paths)
        {
            result = Path.Combine(result, path);
        }
        return result;
    }
}
```

Output 5.8 shows the results of Listing 5.17.

Output 5.8

```
C:\Data\mark\bin\config\index.html
C:\WINDOWS\system32\Temp\index.html
C:\Data\HomeDir\index.html
```

In the first call to `Combine()`, four arguments are specified. The second call contains only three arguments. In the final call, a single argument is passed using an array. In other words, the `Combine()` method takes a variable number of arguments—presented either as any number of string arguments separated by commas or as a single array of strings. The former syntax is called the *expanded* form of the method call, and the latter form is called the *normal* form.

To allow invocation using either form, the `Combine()` method does the following:

1. Places `params` immediately before the last parameter in the method declaration
2. Declares the last parameter as an array

With a **parameter array** declaration, it is possible to access each corresponding argument as a member of the `params` array. In the `Combine()`

method implementation, you iterate over the elements of the `paths` array and call `System.IO.Path.Combine()`. This method automatically combines the parts of the path, appropriately using the platform-specific directory-separator character. Note that `PathEx.Combine()` is identical to `Path.Combine()`, except that `PathEx.Combine()` handles a variable number of parameters rather than simply two.

There are a few notable characteristics of the parameter array:

- The parameter array is not necessarily the only parameter on a method.
- The parameter array must be the last parameter in the method declaration. Since only the last parameter may be a parameter array, a method cannot have more than one parameter array.
- The caller can specify zero arguments that correspond to the parameter array parameter, which will result in an array of zero items being passed as the parameter array.
- Parameter arrays are type-safe: The arguments given must be compatible with the element type of the parameter array.
- The caller can use an explicit array rather than a comma-separated list of arguments. The resulting CIL code is identical.
- If the target method implementation requires a minimum number of parameters, those parameters should appear explicitly within the method declaration, forcing a compile error instead of relying on runtime error handling if required parameters are missing. For example, if you have a method that requires one or more integer arguments, declare the method as `int Max(int first, params int[] operands)` rather than as `int Max(params int[] operands)` so that at least one value is passed to `Max()`.

Using a parameter array, you can pass a variable number of arguments of the same type into a method. The section "Method Overloading," which appears later in this chapter, discusses a means of supporting a variable number of arguments that are not necessarily of the same type.

Guidelines

DO use parameter arrays when a method can handle any number—including zero—of additional arguments.

By the way, a path `Combine()` function is a contrived example since, in fact, `System.IO.Path.Combine()` is an existing function that is overloaded to support parameter arrays.

Recursion

Calling a method **recursively** or implementing the method using **recursion** refers to use of a method that calls itself. Recursion is sometimes the simplest way to implement a particular algorithm. Listing 5.18 counts the lines of all the C# source files (`*.cs`) in a directory and its subdirectory.

LISTING 5.18: Counting the Lines within *.cs Files, Given a Directory

```
#nullable enable
using System.IO;

public static class LineCounter
{
  // Use the first argument as the directory
  // to search, or default to the current directory
  public static void Main(string[] args)
  {
      int totalLineCount = 0;
      string directory;
      if (args.Length > 0)
      {
          directory = args[0];
      }
      else
      {
          directory = Directory.GetCurrentDirectory();
      }
      totalLineCount = DirectoryCountLines(directory);
      System.Console.WriteLine(totalLineCount);
    }

  static int DirectoryCountLines(string directory)
  {
      int lineCount = 0;
      foreach (string file in
          Directory.GetFiles(directory, "*.cs"))
      {
            lineCount += CountLines(file);
        }

      foreach (string subdirectory in
          Directory.GetDirectories(directory))
      {
          lineCount += DirectoryCountLines(subdirectory);
      }
```

```
        return lineCount;
    }

    private static int CountLines(string file)
    {
        string? line;
        int lineCount = 0;
        FileStream stream =
            new FileStream(file, FileMode.Open);4
        StreamReader reader = new StreamReader(stream);
        line = reader.ReadLine();

         while(line is object)
         {
             if (line.Trim() != "")
             {
                 lineCount++;
             }
             line = reader.ReadLine();
         }

        reader.Close();  // Automatically closes the stream
        return lineCount;
    }
}
```

Output 5.9 shows the results of Listing 5.18.

Output 5.9

```
104
```

The program begins by passing the first command-line argument to `DirectoryCountLines()` or by using the current directory if no argument is provided. This method first iterates through all the files in the current directory and totals the source code lines for each file. After processing each file in the directory, the code processes each subdirectory by passing the subdirectory back into the `DirectoryCountLines()` method, rerunning the method using the subdirectory. The same process is repeated recursively through each subdirectory until no more directories remain to process.

Readers unfamiliar with recursion may find it confusing at first. Regardless, it is often the simplest pattern to code, especially with hierarchical type data such as the filesystem. However, although it may be the

4. This code could be improved with a `using` statement—a construct that I have avoided because it has not yet been introduced.

most readable approach, it is generally not the fastest implementation. If performance becomes an issue, developers should seek an alternative solution to a recursive implementation. The choice generally hinges on balancing readability with performance.

BEGINNER TOPIC

Infinite Recursion Error

A common programming error in recursive method implementations appears in the form of a stack overflow during program execution. This usually happens because of **infinite recursion**, in which the method continually calls back on itself, never reaching a point that triggers the end of the recursion. It is a good practice for programmers to review any method that uses recursion and to verify that the recursion calls are finite.

A common pattern for recursion using pseudocode is as follows:

```
M(x)
{
  if x is trivial
    return the result
  else
    a. Do some work to make the problem smaller
    b. Recursively call M to solve the smaller problem
    c. Compute the result based on a and b
    return the result
}
```

Things go wrong when this pattern is not followed. For example, if you don't make the problem smaller or if you don't handle all possible "smallest" cases, the recursion never terminates.

Method Overloading

Listing 5.18 called `DirectoryCountLines()`, which counted the lines of `*.cs` files. However, if you want to count code in `*.h/*.cpp` files or in `*.vb` files, `DirectoryCountLines()` will not work. Instead, you need a method that takes the file extension but still keeps the existing method definition so that it handles `*.cs` files by default.

All methods within a class must have a unique signature, and C# defines uniqueness by variation in the method name, parameter data types, or number of parameters. This does not include method return data types;

defining two methods that differ only in their return data types will cause a compile error. This is true even if the return type is two different tuples. **Method overloading** occurs when a class has two or more methods with the same name and the parameter count and/or data types vary between the overloaded methods.

> **NOTE**
>
> A method is considered unique as long as there is variation in the method name, parameter data types, or number of parameters.

Method overloading is a type of **operational polymorphism**. Polymorphism occurs when the same logical operation takes on many ("poly") forms ("morphs") because the data varies. For example, calling `WriteLine()` and passing a format string along with some parameters is implemented differently than calling `WriteLine()` and specifying an integer. However, logically, to the caller, the method takes care of writing the data, and it is somewhat irrelevant how the internal implementation occurs. Listing 5.19 provides an example, and Output 5.10 shows the results.

Listing 5.19: Counting the Lines within *.cs Files Using Overloading

```
#nullable enable
using System.IO;

public static class LineCounter
{
  public static void Main(string[] args)
  {
      int totalLineCount;

      if (args.Length > 1)
      {
          totalLineCount =
              DirectoryCountLines(args[0], args[1]);
      }
      if (args.Length > 0)
      {
          totalLineCount = DirectoryCountLines(args[0]);
      }
      else
      {
          totalLineCount  = DirectoryCountLines();
      }
```

```
        System.Console.WriteLine(totalLineCount);
    }

    static int DirectoryCountLines()
    {
        return DirectoryCountLines(
            Directory.GetCurrentDirectory());
    }

    static int DirectoryCountLines(string directory)
    {
        return DirectoryCountLines(directory, "*.cs");
    }

    static int DirectoryCountLines(
        string directory, string extension)
    {
        int lineCount = 0;
        foreach (string file in
            Directory.GetFiles(directory, extension))
        {
            lineCount += CountLines(file);
        }

        foreach (string subdirectory in
            Directory.GetDirectories(directory))
        {
            lineCount += DirectoryCountLines(subdirectory);
        }

        return lineCount;
    }

    private static int CountLines(string file)
    {
        int lineCount = 0;
        string? line;
        FileStream stream =
            new FileStream(file, FileMode.Open);5
        StreamReader reader = new StreamReader(stream);
        line = reader.ReadLine();
        while(line is object)
        {
            if (line.Trim() != "")
            {
                lineCount++;
            }
            line = reader.ReadLine();
        }
```

5. This code could be improved with a `using` statement—a construct that we have avoided because it has not yet been introduced.

```
        reader.Close();  // Automatically closes the stream
        return lineCount;
      }
    }
```

Output 5.10

```
>LineCounter.exe .\ *.cs
28
```

The effect of method overloading is to provide optional ways to call the method. As demonstrated inside `Main()`, you can call the `DirectoryCountLines()` method with or without passing the directory to search and the file extension.

Notice that the parameterless implementation of `DirectoryCountLines()` was changed to call the single-parameter version, `int DirectoryCountLines (string directory)`. This is a common pattern when implementing overloaded methods. The idea is that developers implement only the core logic in one method, and all the other overloaded methods will call that single method. If the core implementation changes, it needs to be modified in only one location rather than within each implementation. This pattern is especially prevalent when using method overloading to enable optional parameters that do not have values determined at compile time, so they cannot be specified using optional parameters.

> **NOTE**
>
> Placing the core functionality into a single method that all other overloading methods invoke means that you can make changes in implementation in just the core method, which the other methods will automatically take advantage of.

Begin 4.0

Optional Parameters

The C# language designers also added support for **optional parameters**.[6] By allowing the association of a parameter with a constant value as part of

6. Introduced in C# 4.0.

the method declaration, it is possible to call a method without passing an argument for every parameter of the method (see Listing 5.20).

Listing 5.20: Methods with Optional Parameters

```
#nullable enable
using System.IO;

public static class LineCounter
{
  public static void Main(string[] args)
  {
      int totalLineCount;

      if (args.Length > 1)
      {
          totalLineCount =
              DirectoryCountLines(args[0], args[1]);
      }
      if (args.Length > 0)
      {
          totalLineCount = DirectoryCountLines(args[0]);
      }
      else
      {
          totalLineCount  = DirectoryCountLines();
      }

      System.Console.WriteLine(totalLineCount);
  }

  static int DirectoryCountLines()
  {
      // ...
  }

/*
  static int DirectoryCountLines(string directory)
  { ... }
*/

  static int DirectoryCountLines(
      string directory, string extension = "*.cs")
  {
      int lineCount = 0;
      foreach (string file in
          Directory.GetFiles(directory, extension))
      {
          lineCount += CountLines(file);
      }
```

4.0

```
        foreach (string subdirectory in
            Directory.GetDirectories(directory))
        {
            lineCount += DirectoryCountLines(subdirectory);
        }

        return lineCount;
    }

    private static int CountLines(string file)
    {
        // ...
    }
}
```

In Listing 5.20, the `DirectoryCountLines()` method declaration with a single parameter has been removed (commented out), but the call from `Main()` (specifying one parameter) remains. When no `extension` parameter is specified in the call, the value assigned to `extension` within the declaration (`*.cs` in this case) is used. This allows the calling code to not specify a value if desired, and it eliminates the additional overload that would otherwise be required. Note that optional parameters must appear after all required parameters (those that don't have default values). Also, the fact that the default value needs to be a constant, compile-time–resolved value is fairly restrictive. You cannot, for example, declare a method like

```
DirectoryCountLines(
  string directory = Environment.CurrentDirectory,
  string extension = "*.cs")
```

4.0

because `Environment.CurrentDirectory` is not a constant. In contrast, because `"*.cs"` is a constant, C# does allow it for the default value of an optional parameter.

Guidelines

DO provide good defaults for all parameters where possible.

DO provide simple method overloads that have a small number of required parameters.

CONSIDER organizing overloads from the simplest to the most complex.

A second method call feature is the use of **named arguments**.[7] With named arguments, it is possible for the caller to explicitly identify the name of the parameter to be assigned a value, rather than relying solely on parameter and argument order to correlate them (see Listing 5.21).

Listing 5.21: Specifying Parameters by Name

```
#nullable enable
using System.IO;

class Program
{
  static void Main()
  {
      DisplayGreeting(
          firstName: "Inigo", lastName: "Montoya");
  }

  public static void DisplayGreeting(
      string firstName,
      string? middleName = null,
      string? lastName = null
  )
  {

      // ...

  }
}
```

In Listing 5.21, the call to `DisplayGreeting()` from within `Main()` assigns a value to a parameter by name. Of the two optional parameters (`middleName` and `lastName`), only `lastName` is given as an argument. For cases where a method has lots of parameters and many of them are optional (a common occurrence when accessing Microsoft COM libraries), using the named argument syntax is certainly a convenience. However, along with the convenience comes an impact on the flexibility of the method interface. In the past, parameter names could be changed without causing C# code that invokes the method to no longer compile. With the addition of named parameters, the parameter name becomes part of the interface because changing the name would cause code that uses the named parameter to no longer compile.

4.0

7. Introduced in C# 4.0.

Guidelines

DO treat parameter names as part of the API, and avoid changing the names if version compatibility between APIs is important.

For many experienced C# developers, this is a surprising restriction. However, the restriction has been imposed as part of the Common Language Specification ever since .NET 1.0. Moreover, Visual Basic has always supported calling methods with named arguments. Therefore, library developers should already be following the practice of not changing parameter names to successfully interoperate with other .NET languages from version to version. In essence, named arguments now impose the same restriction on changing parameter names that many other .NET languages already require.

Given the combination of method overloading, optional parameters, and named parameters, resolving which method to call becomes less obvious. A call is **applicable** (compatible) with a method if all parameters have exactly one corresponding argument (either by name or by position) that is type-compatible, unless the parameter is optional (or is a parameter array). Although this restricts the possible number of methods that will be called, it doesn't identify a unique method. To further distinguish which specific method will be called, the compiler uses only explicitly identified parameters in the caller, ignoring all optional parameters that were not specified at the caller. Therefore, if two methods are applicable because one of them has an optional parameter, the compiler will resolve to the method without the optional parameter.

End 4.0

ADVANCED TOPIC

Method Resolution

When the compiler must choose which of several applicable methods is the best one for a particular call, the one with the *most specific* parameter types is chosen. Assuming there are two applicable methods, each requiring an implicit conversion from an argument to a parameter type, the method whose parameter type is the more derived type will be used.

For example, a method that takes a `double` parameter will be chosen over a method that takes an `object` parameter if the caller passes an argument of type `int`. This is because `double` is more specific than `object`. There are `objects` that are not `doubles`, but there are no `doubles` that are not `objects`, so `double` must be more specific.

If more than one method is applicable and no unique best method can be determined, the compiler will issue an error indicating that the call is ambiguous.

For example, given the following methods

```
static void Method(object thing){}
static void Method(double thing){}
static void Method(long thing){}
static void Method(int thing){}
```

a call of the form `Method(42)` will resolve as `Method(int thing)` because that is an exact match from the argument type to the parameter type. Were that method to be removed, overload resolution would choose the `long` version, because `long` is more specific than either `double` or `object`.

The C# specification includes additional rules governing implicit conversion between `byte`, `ushort`, `uint`, `ulong`, and the other numeric types. In general, though, it is better to use a cast to make the intended target method more recognizable.

Basic Error Handling with Exceptions

This section examines how to handle error reporting via a mechanism known as **exception handling**. With exception handling, a method is able to pass information about an error to a calling method without using a return value or explicitly providing any parameters to do so. Listing 5.22 contains a slight modification to Listing 1.16—the `HeyYou` program from Chapter 1. Instead of requesting the last name of the user, it prompts for the user's age.

Listing 5.22: Converting a `string` to an `int`

```
using System;

class ExceptionHandling
{
  static void Main()
  {
      string firstName;
      string ageText;
```

```
        int age;

        Console.WriteLine("Hey you!");

        Console.Write("Enter your first name: ");
        firstName = System.Console.ReadLine();

        Console.Write("Enter your age: ");
        ageText = Console.ReadLine();
        age = int.Parse(ageText);

        Console.WriteLine(
            $"Hi { firstName }!  You are { age*12 } months old.");
    }
}
```

Output 5.11 shows the results of Listing 5.22.

Output 5.11

```
Hey you!
Enter your first name: Inigo
Enter your age: 42
Hi Inigo!  You are 504 months old.
```

The return value from `System.Console.ReadLine()` is stored in a variable called `ageText` and is then passed to a method with the `int` data type, called `Parse()`. This method is responsible for taking a string value that represents a number and converting it to an `int` type.

Beginner Topic

42 as a String versus 42 as an Integer

C# requires that every non-`null` value have a well-defined type associated with it. Therefore, not only the data value but also the type associated with the data is important. A string value of `42`, therefore, is distinctly different from an integer value of `42`. The string is composed of the two characters `4` and `2`, whereas the `int` is the number 42.

Given the converted string, the final `System.Console.WriteLine()` statement will print the age in months by multiplying the age value by 12.

But what happens if the user does not enter a valid integer string? For example, what happens if the user enters "forty-two"? The `Parse()` method cannot handle such a conversion. It expects the user to enter a string that contains only digits. If the `Parse()` method is sent an invalid value, it needs some way to report this fact back to the caller.

Trapping Errors

To indicate to the calling method that the parameter is invalid, `int.Parse()` will **throw an exception**. Throwing an exception halts further execution in the current control flow and jumps into the first code block within the call stack that handles the exception.

Since you have not yet provided any such handling, the program reports the exception to the user as an **unhandled exception**. Assuming there is no registered debugger on the system, the error will appear on the console with a message such as that shown in Output 5.12.

OUTPUT 5.12

```
Hey you!
Enter your first name: Inigo
Enter your age: forty-two

Unhandled Exception: System.FormatException: Input string was
        not in a correct format.
    at System.Number.ParseInt32(String s, NumberStyles style,
        NumberFormatInfo info)
    at ExceptionHandling.Main()
```

Obviously, such an error is not particularly helpful. To fix this, it is necessary to provide a mechanism that handles the error, perhaps reporting a more meaningful error message back to the user.

This process is known as **catching an exception**. The syntax is demonstrated in Listing 5.23, and the output appears in Output 5.13.

LISTING 5.23: Catching an Exception

```
using System;

class ExceptionHandling
{
  static int Main()
  {
      string firstName;
      string ageText;
```

```
            int age;
            int result = 0;

            Console.Write("Enter your first name: ");
            firstName = Console.ReadLine();

            Console.Write("Enter your age: ");
            ageText = Console.ReadLine();

            try
            {
                age = int.Parse(ageText);
                Console.WriteLine(
                    $"Hi { firstName }! You are { age*12 } months old.");
            }
            catch (FormatException )
            {
                Console.WriteLine(
                    $"The age entered, { ageText }, is not valid.");
                result = 1;
            }
            catch(Exception exception)
            {
                Console.WriteLine(
                    $"Unexpected error:  { exception.Message }");
                result = 1;
            }
            finally
            {
                Console.WriteLine($"Goodbye { firstName }");
            }

            return result;
        }
    }
```

Output 5.13

```
Enter your first name: Inigo
Enter your age: forty-two
The age entered, forty-two, is not valid.
Goodbye Inigo
```

To begin, surround the code that could potentially throw an exception (`age = int.Parse()`) with a **try block**. This block begins with the `try` keyword. It indicates to the compiler that the developer is aware of the possibility that the code within the block might throw an exception, and if it does, one of the **catch blocks** will attempt to handle the exception.

One or more catch blocks (or the finally block) must appear immediately following a try block. The catch block header (see Advanced Topic: General Catch later in this chapter) optionally allows you to specify the data type of the exception. As long as the data type matches the exception type, the catch block will execute. If, however, there is no appropriate catch block, the exception will fall through and go unhandled as though there were no exception handling. The resultant control flow appears in Figure 5.1.

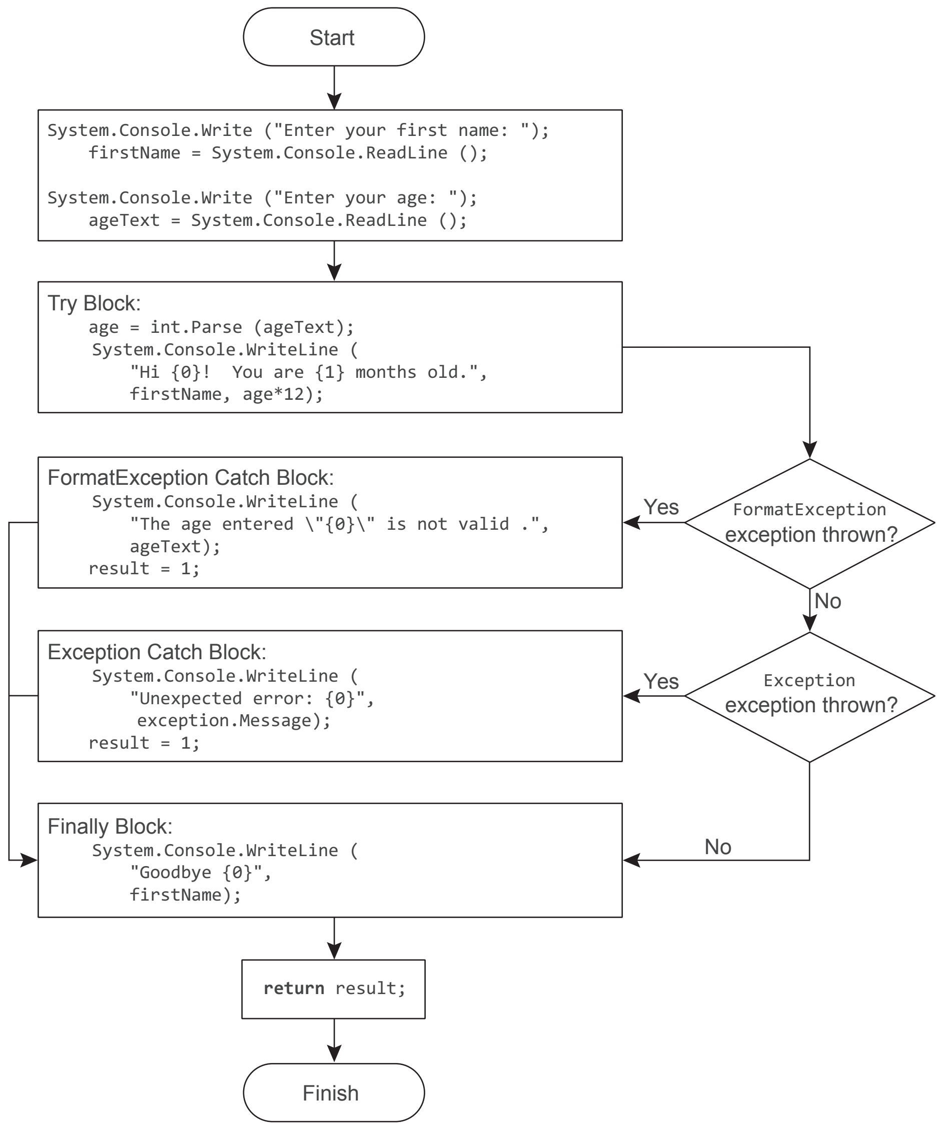


FIGURE 5.1: Exception-handling control flow

For example, assume the user enters "forty-two" for the age in the previous example. In this case, `int.Parse()` will throw an exception of type `System.FormatException`, and control will jump to the set of catch blocks. (`System.FormatException` indicates that the string was not of the correct format to be parsed appropriately.) Since the first catch block matches the type of exception that `int.Parse()` threw, the code inside this block will execute. If a statement within the try block threw a different exception, the second catch block would execute because all exceptions are of type `System.Exception`.

If there were no `System.FormatException` catch block, the `System.Exception` catch block would execute even though `int.Parse` throws a `System.FormatException`. This is because a `System.FormatException` is also of type `System.Exception`. (`System.FormatException` is a more specific implementation of the generic exception, `System.Exception`.)

The order in which you handle exceptions is significant. Catch blocks must appear from most specific to least specific. The `System.Exception` data type is least specific, so it appears last. `System.FormatException` appears first because it is the most specific exception that Listing 5.23 handles.

Regardless of whether control leaves the try block normally or because the code in the try block throws an exception, the **finally block** of code will execute after control leaves the try-protected region. The purpose of the finally block is to provide a location to place code that will execute regardless of how the try/catch blocks exit—with or without an exception. Finally blocks are useful for cleaning up resources, regardless of whether an exception is thrown. In fact, it is possible to have a try block with a finally block and no catch block. The finally block executes regardless of whether the try block throws an exception or whether a catch block is even written to handle the exception. Listing 5.24 demonstrates the try/finally block, and Output 5.14 shows the results.

LISTING 5.24: Finally Block without a Catch Block

```
using System;

class ExceptionHandling
{
  static int Main()
  {
      string firstName;
      string ageText;
```

```
        int age;
        int result = 0;

        Console.Write("Enter your first name: ");
        firstName = Console.ReadLine();

        Console.Write("Enter your age: ");
        ageText = Console.ReadLine();

        try
        {
            age = int.Parse(ageText);
            Console.WriteLine(
                $"Hi { firstName }! You are { age*12 } months old.");
        }
        finally
        {
            Console.WriteLine($"Goodbye { firstName }");
        }

        return result;
    }
}
```

Output 5.14

```
Enter your first name: Inigo
Enter your age: forty-two

Unhandled Exception: System.FormatException: Input string was not in a
correct format.
   at System.Number.StringToNumber(String str, NumberStyles options,
NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style,
NumberFormatInfo info)
   at ExceptionHandling.Main()
Goodbye Inigo
```

The attentive reader will have noticed something interesting here: The runtime first reported the unhandled exception and then ran the finally block. What explains this unusual behavior?

First, the behavior is legal because when an exception is unhandled, the behavior of the runtime is implementation defined—any behavior is legal! The runtime chooses this particular behavior because it knows before it chooses to run the finally block that the exception will be unhandled; the runtime has already examined all of the activation frames on the call stack and determined that none of them is associated with a catch block that matches the thrown exception.

As soon as the runtime determines that the exception will be unhandled, it checks whether a debugger is installed on the machine, because you might be the software developer who is analyzing this failure. If a debugger is present, it offers the user the chance to attach the debugger to the process *before* the finally block runs. If there is no debugger installed or if the user declines to debug the problem, the default behavior is to print the unhandled exception to the console and then see if there are any finally blocks that could run. Due to the "implementation-defined" nature of the situation, the runtime is not required to run finally blocks in this situation; an implementation may choose to do so or not.

Guidelines

AVOID explicitly throwing exceptions from finally blocks. (Implicitly thrown exceptions resulting from method calls are acceptable.)

DO favor try/finally and avoid using try/catch for cleanup code.

DO throw exceptions that describe which exceptional circumstance occurred and, if possible, how to prevent it.

ADVANCED TOPIC

Exception Class Inheritance

All objects thrown as exceptions derive from `System.Exception`.[8] (Objects thrown from other languages that do not derive from `System.Exception` are automatically "wrapped" by an object that does.) Therefore, they can be handled by the `catch(System.Exception exception)` block. It is preferable, however, to include a catch block that is specific to the most derived type (e.g., `System.FormatException`), because then it is possible to get the most information about an exception and handle it less generically. In so doing, the `catch` statement that uses the most derived type is able to handle the exception type specifically, accessing data related to the exception thrown and avoiding conditional logic to determine what type of exception occurred.

8. Starting in C# 2.0.

This is why C# enforces the rule that catch blocks appear from most derived to least derived. For example, a `catch` statement that catches `System.Exception` cannot appear before a statement that catches `System.FormatException` because `System.FormatException` derives from `System.Exception`.

A method could throw many exception types. Table 5.2 lists some of the more common ones within the framework.

TABLE 5.2: Common Exception Types

Exception Type	Description
`System.Exception`	The "base" exception from which all other exceptions derive.
`System.ArgumentException`	Indicates that one of the arguments passed into the method is invalid.
`System.ArgumentNullException`	Indicates that a particular argument is `null` and that this is not a valid value for that parameter.
`System.ApplicationException`	To be avoided. The original idea was that you might want to have one kind of handling for system exceptions and another for application exceptions, which, although plausible, doesn't actually work well in the real world.
`System.FormatException`	Indicates that the string format is not valid for conversion.
`System.IndexOutOfRangeException`	Indicates that an attempt was made to access an array or other collection element that does not exist.
`System.InvalidCastException`	Indicates that an attempt to convert from one data type to another was not a valid conversion.
`System.InvalidOperationException`	Indicates that an unexpected scenario has occurred such that the application is no longer in a valid state of operation.

continues

Table 5.2: Common Exception Types (*continued*)

Exception Type	Description
`System.NotImplementedException`	Indicates that although the method signature exists, it has not been fully implemented.
`System.NullReferenceException`	Thrown when code tries to find the object referred to by a reference that is `null`.
`System.ArithmeticException`	Indicates an invalid math operation, not including divide by zero.
`System.ArrayTypeMismatchException`	Occurs when attempting to store an element of the wrong type into an array.
`System.StackOverflowException`	Indicates an unexpectedly deep recursion.

ADVANCED TOPIC

General Catch

It is possible to specify a catch block that takes no parameters, as shown in Listing 5.25.

Listing 5.25: General Catch Blocks

```
// A previous catch clause already catches all exceptions
#pragma warning disable CS1058
...
try
{
    age = int.Parse(ageText);
    System.Console.WriteLine(
        $"Hi { firstName }!  You are { age*12 } months old.");
}
catch (System.FormatException exception)
{
    System.Console.WriteLine(
        $"The age entered ,{ ageText }, is not valid.");
    result = 1;
}
catch(System.Exception exception)
{
```

```
        System.Console.WriteLine(
            $"Unexpected error:  { exception.Message }");
        result = 1;
    }
    catch
    {
        System.Console.WriteLine("Unexpected error!");
        result = 1;
    }
    finally
    {
        System.Console.WriteLine($"Goodbye { firstName }");
    }
    ...
```

A catch block with no data type, called a **general catch block**, is equivalent to specifying a catch block that takes an `object` data type—for instance, `catch(object exception){...}`. For this reason, a warning is triggered stating that the catch block already exists; hence the `#pragma warning disable` directive.

Because all classes ultimately derive from `object`, a catch block with no data type must appear last.

General catch blocks are rarely used because there is no way to capture any information about the exception. In addition, C# doesn't support the ability to throw an exception of type `object`. (Only libraries written in languages such as C++ allow exceptions of any type.)

Guidelines

AVOID general catch blocks and replace them with a catch of `System.Exception`.

AVOID catching exceptions for which the appropriate action is unknown. It is better to let an exception go unhandled than to handle it incorrectly.

AVOID catching and logging an exception before rethrowing it. Instead, allow the exception to escape until it can be handled appropriately.

Reporting Errors Using a `throw` Statement

C# allows developers to throw exceptions from their code, as demonstrated in Listing 5.26 and Output 5.15.

Listing 5.26: Throwing an Exception

```
// A previous catch clause already catches all exceptions
using System;
public class ThrowingExceptions
{
  public static void Main()
  {
      try
      {
          Console.WriteLine("Begin executing");

          Console.WriteLine("Throw exception");
               throw new Exception("Arbitrary exception");
          Console.WriteLine("End executing");
      }
      catch(FormatException exception)
      {
         Console.WriteLine(
             "A FormateException was thrown");
      }
      catch(Exception exception)
      {
        Console.WriteLine(
             $"Unexpected error: { exception.Message }");
      }
      catch
      {
           Console.WriteLine("Unexpected error!");
      }

      Console.WriteLine(
          "Shutting down...");
  }
}
```

Output 5.15

```
Begin executing
Throw exception...
Unexpected error:  Arbitrary exception
Shutting down...
```

As the arrows in Listing 5.26 depict, throwing an exception causes execution to jump from where the exception is thrown into the first catch block within the stack that is compatible with the thrown exception type.[9] In this case, the second catch block handles the exception and writes out

9. Technically it could be caught by a compatible catch filter as well.

an error message. In Listing 5.26, there is no finally block, so execution falls through to the `System.Console.WriteLine()` statement following the try/catch block.

To throw an exception, it is necessary to have an instance of an exception. Listing 5.26 creates an instance using the keyword `new` followed by the type of the exception. Most exception types allow a message to be generated as part of throwing the exception, so that when the exception occurs, the message can be retrieved.

Sometimes a catch block will trap an exception but be unable to handle it appropriately or fully. In these circumstances, a catch block can rethrow the exception using the `throw` statement without specifying any exception, as shown in Listing 5.27.

LISTING 5.27: Rethrowing an Exception

```
...
      catch(Exception exception)
      {
          Console.WriteLine(
              $@"Rethrowing unexpected error:  {
                  exception.Message }");
          throw;
      }
...
```

In Listing 5.27, the `throw` statement is "empty" rather than specifying that the exception referred to by the `exception` variable is to be thrown. This illustrates a subtle difference: `throw;` preserves the *call stack* information in the exception, whereas `throw exception;` replaces that information with the current call stack information. For debugging purposes, it is usually better to know the original call stack.

Guidelines

DO prefer using an empty throw when catching and rethrowing an exception, so as to preserve the call stack.

DO report execution failures by throwing exceptions rather than returning error codes.

DO NOT have public members that return exceptions as return values or an `out` parameter. Throw exceptions to indicate errors; do not use them as return values to indicate errors.

Avoid Using Exception Handling to Deal with Expected Situations

Developers should avoid throwing exceptions for expected conditions or normal control flow. For example, developers should not expect users to enter valid text when specifying their age.[10] Therefore, instead of relying on an exception to validate data entered by the user, developers should provide a means of checking the data before attempting the conversion. (Better yet, they should prevent the user from entering invalid data in the first place.) Exceptions are designed specifically for tracking exceptional, unexpected, and potentially fatal situations. Using them for an unintended purpose such as expected situations will cause your code to be hard to read, understand, and maintain.

Begin 2.0

Consider, for example, the `int.Parse()` method we used in Chapter 2 to convert a string to an integer. In this scenario, the code converted user input that was expected to not always be a number. One of the problems with the `Parse()` method is that the only way to determine whether the conversion will be successful is to attempt the cast and then catch the exception if it doesn't work. Because throwing an exception is a relatively expensive operation, it is better to attempt the conversion without exception handling. Toward this effort, it is preferable to use one of the `TryParse()` methods, such as `int.TryParse()`. It requires the use of the `out` keyword because the return from the `TryParse()` function is a `bool` rather than the converted value. Listing 5.28 is a code snippet that demonstrates the conversion using `int.TryParse()`.

Listing 5.28: Conversion Using `int.TryParse()`

```
if (int.TryParse(ageText, out int age))
{
    Console.WriteLine(
        $"Hi { firstName }!  "
        + $"You are { age*12 } months old.");
}
else
{
    Console.WriteLine(
        $"The age entered, { ageText }, is not valid.");
}
```

10. In general, developers should expect their users to perform unexpected actions; in turn, they should code defensively to handle "stupid user tricks."

With the `TryParse()` method, it is no longer necessary to include a try/catch block simply for the purpose of handling the string-to-numeric conversion.

End 2.0

Another factor in favor of avoiding exceptions for expected scenarios is performance. Like most languages, C# incurs a slight performance hit when throwing an exception—taking microseconds compared to the nanoseconds most operations take. This delay is generally not noticeable in human time—except when the exception goes unhandled. For example, when Listing 5.22 is executed and the user enters an invalid age, the exception is unhandled and there is a noticeable delay while the runtime searches the environment to see whether there is a debugger to load. Fortunately, slow performance when a program is shutting down isn't generally a factor to be concerned with.

Guidelines

DO NOT use exceptions for handling normal, expected conditions; use them for exceptional, unexpected conditions.

Summary

This chapter discussed the details of declaring and calling methods, including the use of the keywords `out` and `ref` to pass and return variables rather than their values. In addition to method declaration, this chapter introduced exception handling.

A method is a fundamental construct that is a key to writing readable code. Instead of writing large methods with lots of statements, you should use methods to create "paragraphs" of roughly 10 or fewer statements within your code. The process of breaking large functions into smaller pieces is one of the ways you can refactor your code to make it more readable and maintainable.

The next chapter considers the class construct and describes how it encapsulates methods (behavior) and fields (data) into a single unit.

This page intentionally left blank

Index

C

E

F

M

T

U

V

W

This page intentionally left blank

Index of 8.0 Topics

This page intentionally left blank

Index of 7.0 Topics

Index of 6.0 Topics