THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES

ESSENTIAL C# 6.0

"Welcome to one of the greatest collaborations you could dream of in the world of C# books—and probably far beyond!"

—*From the Foreword by* **Mads Torgersen,** *C# Program Manager, Microsoft*

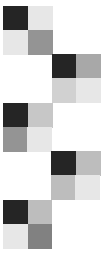
MARK MICHAELIS
with **ERIC LIPPERT**

IntelliTect

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Essential C# 6.0



Essential C# 6.0

Mark Michaelis
with Eric Lippert

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Michaelis, Mark.
Essential C# 6.0 / Mark Michaelis with Eric Lippert.
pages cm
Includes index.
ISBN 978-0-13-414104-6 (pbk. : alk. paper) — ISBN 0-13-414104-0 (pbk. : alk. paper)
1. C# (Computer program language) 2. Microsoft .NET Framework. I. Lippert, Eric, author. II. Title.
QA76.73.C154M5239 2016
005.13'3—dc23

2015025757

ISBN-13: 978-0-13-414104-6
ISBN-10: 0-13-414104-0
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan
First printing, September 2015

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!

This page intentionally left blank

Contents at a Glance

Contents

Figures

Tables

Foreword

WELCOME TO ONE of the greatest collaborations you could dream of in the world of C# books—and probably far beyond! Mark Michaelis' *Essential C#* series was already a classic when, for the previous edition, he teamed up with famous C# blogger Eric Lippert—a masterstroke.

You may think of Eric as writing blogs and Mark as writing books, but that is not how I first got to know them.

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 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 walkthroughs. 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, and busily pulling together his own picture.

As a newcomer to the C# community, I think 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 it—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 because 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.

Eric is, of course, my former colleague of seven years on the C# team. He'd been there much longer than I had, and the first I recall of him, he was explaining to the team how to untangle a bowl of spaghetti. More precisely, our C# compiler code base at the time was in need of some serious architectural TLC and was exceedingly hard to add new features to—something we desperately needed to be able to do with LINQ. Eric had been investigating what kind of architecture we ought to have (Phases! We didn't even really have those!), and more importantly, how to get from here to there, step by step. The remarkable thing was that as complex as this was, and as new as I was to the team and the code base, I immediately understood what he was saying!

You may recognize from his blogs the super-clear and well-structured untangling of the problem, the convincing clarity of enumerated solutions, and the occasional unmitigated hilarity. Well, you don't know the half of it! Every time Eric was grappling with a complex issue and was sharing his thoughts about it with the team, his emails about it were just as meticulous and every bit as hilarious. You fundamentally couldn't ignore an issue raised by Eric because you couldn't wait to read his prose about it. They were even purple, too! So I essentially got to enjoy a continuous supply of what amounts to unpublished installments of his blog, as well as, of course, his pleasant and insightful presence as a member of the C# compiler team and language design team. In his post-Microsoft days, Eric has continued to be a wonderful, insightful voice with a lot more influence on our decisions than he probably knows.

In summary, I am truly grateful to get to work with these two amazing people on a regular basis: Eric to help keep my thinking straight and Mark to help keep me honest. They share a great gift of providing clarity and elucidation, and by combining their "inside" and "outside" perspective on C#, their book reaches a new level of completeness. No one will help you get C# 6 like these two gentlemen do.

Enjoy!

—Mads Torgersen
C# Program Manager
Microsoft

This page intentionally left blank

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 is organized in such a way as to take you through similar paradigm shifts.

The beginning chapters of *Essential C# 6.0* take you through **sequential programming structure,** in which statements are written in the order in which they are executed. The problem with this model is that complexity increases exponentially as the requirements increase. To reduce this complexity, code blocks may be 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, growing programs may quickly become unwieldy and require further abstraction. Object-oriented programming, discussed in Chapter 5, was a response intended to rectify this situation. In subsequent chapters of this book, 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 17) 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#, it 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# 6.0 and .NET Framework 4.6.
- 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.

A number of topics are not covered in this book. You won't find coverage of topics such as ASP.NET, Entity Framework, smart client development such as WPF, distributed programming, and so on. Although these topics are relevant to the .NET Framework, to do them justice requires books of their own. Fortunately, Addison-Wesley's *.NET Development Series* provides a wealth of writing on these topics. *Essential C# 6.0* 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 terms 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 arsenal—another arrow in their quiver, as it were. 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—one 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 well throughout your programming career.

- *Structured programmers:* Just as it's best to learn a foreign language through immersion, so learning a computer language is most effective when you begin using that language before you know all of its 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 4, developers in this category should feel at home writing basic control flow programs. However, the key to excellence for C# developers is not just memorizing syntax. Rather, 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 5's Beginner Topics introduce classes and object-oriented development. The role filled by 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++ and Java programmers, and many experienced Visual Basic 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 similar to the C and C++ styles of 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 answers to language details and subtleties that are seldom addressed. Most importantly, 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, 4.0, 5.0, and now 6.0, some of the most prominent enhancements are as follows:
  - Implicitly typed variables (see Chapter 2)
  - Extension methods (see Chapter 5)
  - Partial methods (see Chapter 5)
  - Anonymous types (see Chapter 11)
  - Generics (see Chapter 11)
  - Lambda statements and expressions (see Chapter 12)

- Expression trees (see Chapter 12)
- Standard query operators (see Chapter 14)
- Query expressions (see Chapter 15)
- Dynamic programming (Chapter 17)
- Multithreaded programming with the Task Programming Library and `async` (Chapter 18)
- Parallel query processing with PLINQ (Chapter 18)
- Concurrent collections (Chapter 19)

These topics are covered in detail for those not already familiar with them. Also pertinent to advanced C# development is the subject of pointers, in Chapter 21. Often, even experienced C# developers do not understand this topic well.

Features of This Book

Essential C# 6.0 is a language book that adheres to the core C# Language 6.0 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 features in *Essential C# 6.0* is the inclusion of C# coding guidelines, as shown in the following example taken from Chapter 16:

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 is able to 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 facilitate 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.

Code Samples

The code snippets in most of this text can run on any implementation of the Common Language Infrastructure (CLI), including the Mono, DNX Core, and Microsoft .NET platforms. Platform- or vendor-specific libraries are seldom used, except when communicating important concepts relevant only to those platforms (appropriately handling the single-threaded user interface of Windows, for example). Any code that specifically requires C# 3.0, 4.0, or 5.0 compliance is called out in the C# version, and separate indexes at the end of the book.

Here is a sample code listing.

LISTING 1.17: Swapping the Indexed Placeholders and Corresponding Variables

```
System.Console.WriteLine("Your full name is {1}, {0}",
  firstName, lastName);
```

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.Write /* No new line */ (
```

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

```
System.Console.WriteLine(
     "Your full name is {0} {1}.",
```

- 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.

OUTPUT 1.4

```
>HeyYou.exe
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
```

User input for the program appears in **boldface.**

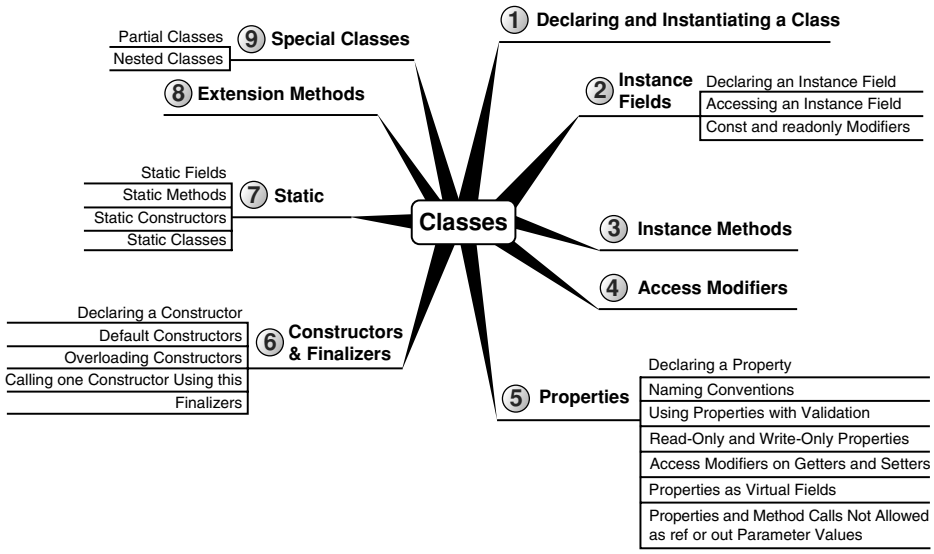
Although it might have been convenient to provide full code samples that you could copy into your own programs, doing so would detract from your learning a particular topic. Therefore, you need to modify the code samples before you can incorporate them into your programs. The core omission is error checking, 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 Intellitect.com/essentialcsharp and at informit.com/mstechseries. In addition, the code is available on Github—see http://itl.tc/EssentialCSharpSCC. Instructions for downloading the tools to compile the source code as well as the compilation instructions themselves are found in Appendix A.

You can also access the errata at http://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 5).



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 code blocks and tabs will help you navigate through the text.

- Beginner Topics provide definitions or explanations specifically targeted to entry-level programmers.
- Advanced Topics enable experienced developers to focus on the material that is most relevant to them.
- Callout notes highlight key principles 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.
- Page-edge **begin** and **end tabs** denote material specific to C# versions; where that material continues over multiple pages, just the version number appears in the tab.

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# 6.0*. Chapters 1–4 introduce structured programming, which enable you to start writing simple functioning code immediately. Chapters 5–9 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 11–13 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 a chapter 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 italics*** indicate the presence of C# 3.0–5.0 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. Chapter 1 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#. This includes coverage of two type categories, value types and reference types, along with conversion between types and support for arrays.
- ***Chapter 3***—*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 4***—*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 a parameter. Default parameter support was added in C# 4.0, and this chapter explains how to use this support.

- ***Chapter 5***—*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.
- ***Chapter 6***—*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 7***—*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.
- ***Chapter 8***—*Value Types:* Although it is more common to define 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 9—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 the process of defining namespaces and the use of XML comments, and discusses how to design classes for garbage collection.
- ***Chapter 10***—*Exception Handling:* This chapter expands on the exception-handling introduction from Chapter 4 and describes how exceptions follow a hierarchy that supports the creation of custom exceptions. It also includes some best practices on exception handling.
- ***Chapter 11***—*Generics:* Generics is 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 12****—Delegates and Lambda Expressions:* Delegates begin clearly distinguishing C# from its predecessors by defining patterns for handling events within code. This practice virtually eliminates the need for writing routines that poll. Lambda expressions are the key concept that make C# 3.0's LINQ possible. Chapter 12 explains how lambda expressions build on the delegate construct by providing a more elegant and succinct syntax. This chapter forms the foundation for the new collection API discussed next.
- ***Chapter 13****—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 14****—Collection Interfaces with Standard Query Operators:* The simple, 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 new `Enumerable` class. This class makes available an entirely new collection API known as the standard query operators, which is discussed in detail here.
- ***Chapter 15****—LINQ with Query Expressions:* Using standard query operators alone results in some long statements that can be challenging to decipher. However, query expressions provide an alternative syntax that matches closely with SQL, as described in this chapter.
- ***Chapter 16****—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 17****—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. It removed all type checking until runtime, a significant expansion of what can be done with C#.

- ***Chapter 18***—*Multithreading:* Most modern programs require the use of threads to execute long-running tasks while ensuring they provide an active response to simultaneous events. As programs become more sophisticated, they must take additional precautions to protect data in these dynamic environments. Programming multithreaded applications is complex. This chapter discusses how to work with threads and provides best practices to avoid the problems that plague multithreaded applications.
- ***Chapter 19***—*Thread Synchronization:* Building on the preceding chapter, Chapter 19 demonstrates some of the built-in threading pattern support that can simplify the explicit control of multithreaded code.
- ***Chapter 20***—*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 21***—*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.
- ***Appendix A***—*Downloading and Installing the C# Compiler and the CLI Platform:* This appendix provides instructions for setting up a C# compiler and the platform on which to run the code, Microsoft .NET or Mono.
- ***Appendix B***—*Tic-Tac-Toe Source Code Listing:* This appendix provides a full listing of the Tic-Tac-Toe program referred to in Chapters 3 and 4.
- ***C# 3.0, 4.0, 5.0, and 6.0 Indexes***—These indexes provide a quick reference for the features added in C# 3.0 through 6.0. They are specifically designed to help programmers quickly update their language skills to a more recent version.

Appendix C, Interfacing with Multithreading Patterns Prior to the TPL and C# 6.0, and Appendix D, Timers Prior to the Async/Await Pattern of C# 6.0, can be found on the book's website, http://www.informit.com/title/9780134141046. Teaching resources that accompany this book will be made available to qualified instructors through Pearson's Instructor Resource Center.

I hope that you find this book to be a great resource in establishing your C# expertise, and that you continue to reference it for the more obscure areas of C# and its inner workings.

—Mark Michaelis
Blog: http://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. By far, my family has made the biggest sacrifice to allow me to complete this project. 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. I would like to say it got easier with each edition but, alas, no: As the kids got older, life became more hectic, and without me Elisabeth was stretched to the breaking point almost all the time. A huge "Sorry" and ginormous "Thank You!"

Many technical editors reviewed each chapter in minute detail to ensure technical accuracy. I was often amazed by the subtle errors these folks managed to catch: Paul Bramsman, Kody Brown, Ian Davis, Doug Dechow, Gerard Frantz, Thomas Heavey, Anson Horton, Brian Jones, Shane Kercheval, Angelika Langer, Eric Lippert, John Michaelis, Jason Morse, Nicholas Paldino, Jon Skeet, Michael Stokesbary, Robert Stokesbary, John Timney, and Stephen Toub.

Eric is no less than amazing. His grasp of the C# vocabulary 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 no longer continues to fester. Eric has painstakingly reviewed every *Essential C#* chapter with amazing detail and precision. I am extremely grateful for

his contribution to making this book even better than the first two editions. Thanks, Eric! I can't imagine anyone better for the job. You deserve all the credit for raising the bar from good to great.

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

Thanks to everyone at Addison-Wesley for their patience in working with me in spite of my occasional tendency to focus on everything else except the manuscript. Thanks to Vicki Rowland, Ellie Bru, Curt Johnson, and Joan Murray. Joan deserves a special medal for her patience given the number of times I delayed not only providing deliverables but even responding to emails. Vicki is no less than amazing in her ability to work with technical authors. I was so appreciative of the updated, fully stylized manuscripts she provided following the *Essential C# 5.0* publication. It made writing *Essential C# 6.0* so much easier than my updates of the prior editions.

Thanks also to Mads Torgersen, for his willingness to write the Foreword. Even if only half of what he says is true, I am greatly honored.

About the Authors

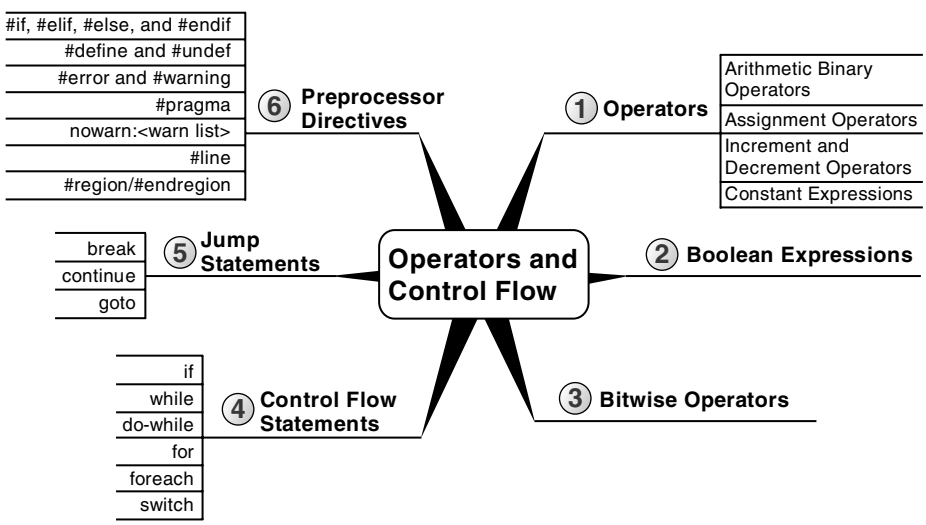
Mark Michaelis is the founder of IntelliTect and serves as the Chief Technical Architect and Trainer. Since 1996, he has been a Microsoft MVP for C#, Visual Studio Team System, and the Windows SDK; in 2007, he was recognized as a Microsoft Regional Director. He also serves on several Microsoft software design review teams, including C#, the Connected Systems, Office/SharePoint, and Visual Studio. Mark speaks at developer conferences and has written numerous articles and other books. He 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 stays busy with his family or training for another Ironman (having completed his first in 2008). Mark lives in Spokane, Washington, with his wife, Elisabeth, and three children, Benjamin, Hanna, and Abigail.

Eric Lippert works on tools for static analysis of C# at Coverity, now a part of Synopsys. Before joining Coverity, he was a principal developer on the C# compiler team at Microsoft. When not blogging or editing books about C#, Eric does his best to keep his tiny sailboat upright. He lives in Seattle, Washington, with his wife, Leah.

This page intentionally left blank

3
Operators and Control Flow

IN THIS CHAPTER, YOU WILL learn about operators, control flow statements, and the C# preprocessor. **Operators** provide syntax for performing different calculations or actions appropriate for the operands within the calculation. **Control flow statements** provide the means for conditional logic within a program or looping over a section of code multiple times. After introducing the `if` control flow statement, the chapter looks at the concept of Boolean expressions, which are embedded within many control flow statements. Included is mention of how integers cannot be converted



(even explicitly) to `bool` and the advantages of this restriction. The chapter ends with a discussion of the C# preprocessor directives.

Operators

Now that you have been introduced to the predefined data types (refer to Chapter 2), you can begin to learn more about how to use these data types in combination with operators to perform calculations. For example, you can make calculations on variables that you have declared.

BEGINNER TOPIC

Operators

Operators are used to perform mathematical or logical operations on values (or variables) called **operands** to produce a new value, called the **result.** For example, in Listing 3.1 the subtraction operator, `-`, is used to subtract two operands, the numbers `4` and `2`. The result of the subtraction is stored in the variable `difference`.

LISTING 3.1: A Simple Operator Example

```
int difference = 4 - 2;
```

Operators are generally classified into three categories—unary, binary, and ternary, corresponding to the number of operands (one, two, and three, respectively). This section covers some of the most basic unary and binary operators. The ternary operators are introduced later in the chapter.

Plus and Minus Unary Operators (+, -)

Sometimes you may want to change the sign of a numerical value. In these cases, the unary minus operator (`-`) comes in handy. For example, Listing 3.2 changes the total current U.S. debt to a negative value to indicate that it is an amount owed.

LISTING 3.2: Specifying Negative Values[1]

```
//National debt to the penny
decimal debt = -18125876697430.99M;
```

Using the minus operator *is equivalent to subtracting the operand from zero.*

The unary plus operator (+) rarely[2] has any effect on a value. It is a superfluous addition to the C# language and was included for the sake of symmetry.

Arithmetic Binary Operators (+, -, *, /, %)

Binary operators require two operands. C# uses infix notation for binary operators: The operator appears between the left and right operands. The result of every binary operator other than assignment must be used somehow—for example, by using it as an operand in another expression such as an assignment.

Language Contrast: C++—Operator-Only Statements

In contrast to the rule mentioned previously, C++ will allow a single binary expression to form the entirety of a statement, such as 4+5, to compile. In C#, only assignment, call, increment, decrement, and object creation expressions are allowed to be the entirety of a statement.

The subtraction example in Listing 3.3 illustrates the use of a binary operator—more specifically, an arithmetic binary operator. The operands appear on each side of the arithmetic operator, and then the calculated value is assigned. The other arithmetic binary operators are addition (+), division (/), multiplication (*), and remainder (%—sometimes called the mod operator).

1. As of February 5, 2015, according to www.treasurydirect.gov.
2. The unary + operator is defined to take operands of type `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` (and nullable versions of those types). Using it on other numeric types such as `short` will convert its operand to one of these types as appropriate.

LISTING 3.3: Using Binary Operators

```
class Division
{
  static void Main()
  {
      int numerator;
      int denominator;
      int quotient;
      int remainder;

      System.Console.Write("Enter the numerator: ");
      numerator = int.Parse(System.Console.ReadLine());

      System.Console.Write("Enter the denominator: ");
      denominator = int.Parse(System.Console.ReadLine());

      quotient = numerator / denominator;
      remainder = numerator % denominator;

      System.Console.WriteLine(
          $"{numerator} / {denominator} = {quotient} with remainder
↳{remainder}");
  }
}
```

Output 3.1 shows the results of Listing 3.3.

OUTPUT 3.1

```
Enter the numerator: 23
Enter the denominator: 3
23 / 3 = 7 with remainder 2
```

In the highlighted assignment statements, the division and remainder operations are executed before the assignments. The order in which operators are executed is determined by their **precedence** and **associativity.** The precedence for the operators used so far is as follows:

1. *, /, and % have highest precedence.
2. + and - have lower precedence.
3. = has the lowest precedence of these six operators.

Therefore, you can assume that the statement behaves as expected, with the division and remainder operators executing before the assignment.

If you forget to assign the result of one of these binary operators, you will receive the compile error shown in Output 3.2.

OUTPUT 3.2

```
... error CS0201: Only assignment, call, increment, decrement,
and new object expressions can be used as a statement
```

BEGINNER TOPIC

Parentheses, Associativity, Precedence, and Evaluation

When an expression contains multiple operators, it can be unclear precisely what the operands of each operator are. For example, in the expression `x+y*z`, clearly the expression `x` is an operand of the addition and `z` is an operand of the multiplication. But is `y` an operand of the addition or the multiplication?

Parentheses allow you to unambiguously associate an operand with its operator. If you wish `y` to be a summand, you can write the expression as `(x+y)*z`; if you want it to be a multiplicand, you can write `x+(y*z)`.

However, C# does not require you to parenthesize every expression containing more than one operator; instead, the compiler can use associativity and precedence to figure out from the context which parentheses you have omitted. **Associativity** determines how similar operators are parenthesized; **precedence** determines how dissimilar operators are parenthesized.

A binary operator may be "left-associative" or "right-associative," depending on whether the expression "in the middle" belongs to the operator on the left or the right. For example, `a-b-c` is assumed to mean `(a-b)-c`, and not `a-(b-c)`; subtraction is therefore said to be "left-associative." Most operators in C# are left-associative; the assignment operators are right-associative.

When the operators are dissimilar, the **precedence** for those operators is used to determine which side the operand in the middle belongs to. For example, multiplication has higher precedence than addition and, therefore, the expression `x+y*z` is evaluated as `x+(y*z)` rather than `(x+y)*z`.

It is often still a good practice to use parentheses to make the code more readable even when use of parentheses does not change the meaning of the

expression. For example, when performing a Celsius-to-Fahrenheit temperature conversion, `(c*9.0/5.0)+32.0` is easier to read than `c*9.0/5.0+32.0`, even though the parentheses are completely unnecessary.

Guidelines

DO use parentheses to make code more readable, particularly if the operator precedence is not clear to the casual reader.

Clearly, operators of higher precedence must execute before adjoining operators of lower precedence: in `x+y*z`, the multiplication must be executed before the addition because the result of the multiplication is the left-hand operand of the addition. However, it is important to realize that precedence and associativity affect only the order in which the *operators* themselves are executed; they do not in any way affect the order in which the *operands* are evaluated.

Operands are always evaluated from left to right in C#. In an expression with three method calls, such as `A()+B()*C()`, first `A()` is evaluated, then `B()`, then `C()`; then the multiplication operator determines the product; and finally the addition operator determines the sum. Just because `C()` is involved in a multiplication and `A()` is involved in a lower-precedence addition does not imply that method invocation `C()` happens before method invocation `A()`.

Language Contrast: C++: Evaluation Order of Operands

In contrast to the rule mentioned here, the C++ specification allows an implementation broad latitude to decide the evaluation order of operands. When given an expression such as `A()+B()*C()`, a C++ compiler can choose to evaluate the function calls in any order, just so long as the product is one of the summands. For example, a legal compiler could evaluate `B()`, then `A()`, then `C()`, then the product, and finally the sum.

Using the Addition Operator with Strings

Operators can also work with non-numeric operands. For example, it is possible to use the addition operator to concatenate two or more strings, as shown in Listing 3.4.

LISTING 3.4: Using Binary Operators with Non-numeric Types

```
class FortyTwo
{
  static void Main()
  {
      short windSpeed = 42;
      System.Console.WriteLine(
          "The original Tacoma Bridge in Washington\nwas "
          + "brought down by a "
          + windSpeed + " mile/hour wind.");
  }
}
```

Output 3.3 shows the results of Listing 3.4.

OUTPUT 3.3

```
The original Tacoma Bridge in Washington
was brought down by a 42 mile/hour wind.
```

Because sentence structure varies among languages in different cultures, developers should be careful not to use the addition operator with strings that possibly will require localization. Similarly, although we can embed expressions within a string using C# 6.0's string interpolation, localization to other languages still requires moving the string to a resource file, neutralizing the string interpolation. For this reason, you should use the addition operator sparingly, favoring composite formatting when localization is a possibility.

Guidelines

DO favor composite formatting over use of the addition operator for concatenating strings when localization is a possibility.

Using Characters in Arithmetic Operations

When introducing the `char` type in Chapter 2, we mentioned that even though it stores characters and not numbers, the `char` type is an **integral** type ("integral" means it is based on an integer). It can participate in arithmetic operations with other integer types. However, interpretation of the value of the `char` type is not based on the character stored within it, but rather on its underlying value. The digit `3`, for example, is represented by the Unicode value `0x33` (hexadecimal), which in base 10 is `51`. The digit `4` is represented by the Unicode value `0x34`, or `52` in base 10. Adding `3` and `4` in Listing 3.5 results in a hexadecimal value of `0x67`, or `103` in base 10, which is the Unicode value for the letter `g`.

Listing 3.5: Using the Plus Operator with the `char` Data Type

```
int n = '3' + '4';
char c = (char)n;
System.Console.WriteLine(c);  // Writes out g.
```

Output 3.4 shows the result of Listing 3.5.

Output 3.4

```
g
```

You can use this trait of character types to determine how far two characters are from each other. For example, the letter `f` is three characters away from the letter `c`. You can determine this value by subtracting the letter `c` from the letter `f`, as Listing 3.6 demonstrates.

Listing 3.6: Determining the Character Difference between Two Characters

```
int distance = 'f' - 'c';
System.Console.WriteLine(distance);
```

Output 3.5 shows the result of Listing 3.6.

Output 3.5

```
3
```

Special Floating-Point Characteristics

The binary floating-point types, `float` and `double`, have some special characteristics, such as the way they handle precision. This section looks at some specific examples, as well as some unique floating-point type characteristics.

A `float`, with seven decimal digits of precision, can hold the value 1,234,567 and the value 0.1234567. However, if you add these two `float`s together, the result will be rounded to 1234567, because the exact result requires more precision than the seven significant digits that a `float` can hold. The error introduced by rounding off to seven digits can become large compared to the value computed, especially with repeated calculations. (See also the Advanced Topic, "Unexpected Inequality with Floating-Point Types," later in this section.)

Internally, the binary floating-point types actually store a binary fraction, not a decimal fraction. Consequently, "representation error" inaccuracies can occur with a simple assignment, such as `double number = 140.6F`. The exact value of 140.6 is the fraction 703/5, but the denominator of that fraction is not a power of 2, so it cannot be represented exactly by a binary floating-point number. The value actually represented is the closest fraction with a power of 2 in the denominator that will fit into the 16 bits of a `float`.

Since the `double` can hold a more accurate value than the `float` can store, the C# compiler will actually evaluate this expression to `double number = 140.600006103516` because `140.600006103516` is the closest binary fraction to 140.6 as a `float`. This fraction is slightly larger than 140.6 when represented as a `double`.

Guidelines

AVOID binary floating-point types when exact decimal arithmetic is required; use the `decimal` floating-point type instead.

ADVANCED TOPIC

Unexpected Inequality with Floating-Point Types

Because floating-point numbers can be unexpectedly rounded off to non-decimal fractions, comparing floating-point values for equality can be quite confusing. Consider Listing 3.7.

Listing 3.7: Unexpected Inequality Due to Floating-Point Inaccuracies

```
decimal decimalNumber = 4.2M;
double doubleNumber1 = 0.1F * 42F;
double doubleNumber2 = 0.1D * 42D;
float floatNumber = 0.1F * 42F;

Trace.Assert(decimalNumber != (decimal)doubleNumber1);
// 1. Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    $"{decimalNumber} != {(decimal)doubleNumber1}");

Trace.Assert((double)decimalNumber != doubleNumber1);
// 2. Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    $"{(double)decimalNumber} != {doubleNumber1}");

Trace.Assert((float)decimalNumber != floatNumber);
// 3. Displays: (float)4.2M != 4.2F
System.Console.WriteLine(
    $"(float){(float)decimalNumber}M != {floatNumber}F");

Trace.Assert(doubleNumber1 != (double)floatNumber);
// 4. Displays: 4.20000006258488 != 4.20000028610229
System.Console.WriteLine(
    $"{doubleNumber1} != {(double)floatNumber}");

Trace.Assert(doubleNumber1 != doubleNumber2);
// 5. Displays: 4.20000006258488 != 4.2
System.Console.WriteLine(
    $"{doubleNumber1} != {doubleNumber2}");

Trace.Assert(floatNumber != doubleNumber2);
// 6. Displays: 4.2F != 4.2D
System.Console.WriteLine(
    $"{floatNumber}F != {doubleNumber2}D");

Trace.Assert((double)4.2F != 4.2D);
// 7. Displays: 4.19999980926514 != 4.2
System.Console.WriteLine(
    $"{(double)4.2F} != {4.2D}");

Trace.Assert(4.2F != 4.2D);
// 8. Displays: 4.2F != 4.2D
System.Console.WriteLine(
    $"{4.2F}F != {4.2D}D");
```

Output 3.6 shows the results of Listing 3.7.

OUTPUT 3.6

```
4.2 != 4.2000000625848
4.2 != 4.2000000625848
(float)4.2M != 4.2F
4.2000000625848 != 4.2000002861022
4.2000000625848 != 4.2
4.2F != 4.2D
4.1999998092651 != 4.2
4.2F != 4.2D
```

The `Assert()` methods alert the developer whenever arguments evaluate to `false`. However, of all the comparisons in this code listing, none of them are in fact equal. In spite of the apparent equality of the values in the code listing, they are not actually equivalent due to the inaccuracies associated with `float` values.

Guidelines

AVOID using equality conditionals with binary floating-point types. Either subtract the two values and see if their difference is less than a tolerance, or use the `decimal` type.

You should be aware of some additional unique floating-point characteristics as well. For instance, you would expect that dividing an integer by zero would result in an error—and it does with data types such as `int` and `decimal`. The `float` and `double` types, however, allow for certain special values. Consider Listing 3.8, and its resultant output, Output 3.7.

LISTING 3.8: Dividing a Float by Zero, Displaying NaN

```
float n=0f;
// Displays: NaN
System.Console.WriteLine(n / 0);
```

OUTPUT 3.7

```
NaN
```

In mathematics, certain mathematical operations are undefined, including dividing zero by itself. In C#, the result of dividing the `float` zero by zero results in a special "Not a Number" value; all attempts to print the output of such a number will result in `NaN`. Similarly, taking the square root of a negative number with `System.Math.Sqrt(-1)` will result in `NaN`.

A floating-point number could overflow its bounds as well. For example, the upper bound of the `float` type is approximately 3.4×10^{38}. Should the number overflow that bound, the result would be stored as "positive infinity" and the output of printing the number would be `Infinity`. Similarly, the lower bound of a `float` type is -3.4×10^{38}, and computing a value below that bound would result in "negative infinity," which would be represented by the string `-Infinity`. Listing 3.9 produces negative and positive infinity, respectively, and Output 3.8 shows the results.

Listing 3.9: Overflowing the Bounds of a `float`

```
// Displays: -Infinity
System.Console.WriteLine(-1f / 0);
// Displays: Infinity
System.Console.WriteLine(3.402823E+38f * 2f);
```

Output 3.8

```
-Infinity
Infinity
```

Further examination of the floating-point number reveals that it can contain a value very close to zero, without actually containing zero. If the value exceeds the lower threshold for the `float` or `double` type, the value of the number can be represented as "negative zero" or "positive zero," depending on whether the number is negative or positive, and is represented in output as `-0` or `0`.

Compound Assignment Operators (+=, -=, *=, /=, %=)

Chapter 1 discussed the simple assignment operator, which places the value of the right-hand side of the operator into the variable on the left-hand side. Compound assignment operators combine common binary operator calculations with the assignment operator. For example, consider Listing 3.10.

LISTING 3.10: Common Increment Calculation

```
int x = 123;
x = x + 2;
```

In this assignment, first you calculate the value of `x + 2` and then you assign the calculated value back to `x`. Since this type of operation is performed relatively frequently, an assignment operator exists to handle both the calculation and the assignment with one operator. The `+=` operator increments the variable on the left-hand side of the operator with the value on the right-hand side of the operator, as shown in Listing 3.11.

LISTING 3.11: Using the += Operator

```
int x = 123;
x += 2;
```

This code, therefore, is equivalent to Listing 3.10.

Numerous other "compound assignment" operators exist to provide similar functionality. You can also use the assignment operator the with subtraction, multiplication, division, and remainder operators (as demonstrated in Listing 3.12).

LISTING 3.12: Other Assignment Operator Examples

```
x -= 2;
x /= 2;
x *= 2;
x %= 2;
```

Increment and Decrement Operators (++, --)

C# includes special unary operators for incrementing and decrementing counters. The **increment operator,** `++`, increments a variable by one each time it is used. In other words, all of the code lines shown in Listing 3.13 are equivalent.

LISTING 3.13: Increment Operator

```
spaceCount = spaceCount + 1;
spaceCount += 1;
spaceCount++;
```

Similarly, you can decrement a variable by one using the **decrement operator,** `--`. Therefore, all of the code lines shown in Listing 3.14 are also equivalent.

Listing 3.14: Decrement Operator

```
lines = lines - 1;
lines -= 1;
lines--;
```

BEGINNER TOPIC

A Decrement Example in a Loop

The increment and decrement operators are especially prevalent in loops, such as the `while` loop described later in the chapter. For example, Listing 3.15 uses the decrement operator to iterate backward through each letter in the alphabet.

Listing 3.15: Displaying Each Character's Unicode Value in Descending Order

```
char current;
int unicodeValue;

// Set the initial value of current.
current = 'z';

do
{
  // Retrieve the Unicode value of current.
  unicodeValue = current;
  System.Console.Write($"{current}={unicodeValue}\t");

  // Proceed to the previous letter in the alphabet;
  current--;
}
while(current >= 'a');
```

Output 3.9 shows the results of Listing 3.15.

Output 3.9

```
z=122   y=121   x=120   w=119   v=118   u=117   t=116   s=115   r=114
q=113   p=112   o=111   n=110   m=109   l=108   k=107   j=106   i=105
h=104   g=103   f=102   e=101   d=100   c=99    b=98    a=97
```

The increment and decrement operators are used in Listing 3.15 to control how many times a particular operation is performed. In this example, notice that the increment operator is also used on a character (`char`) data type. You can use increment and decrement operators on various data types as long as some meaning is assigned to the concept of the "next" or "previous" value for that data type.

We saw that the assignment operator first computes the value to be assigned, and then performs the assignment. The result of the assignment operator is the value that was assigned. The increment and decrement operators are similar: They compute the value to be assigned, perform the assignment, and result in a value. It is therefore possible to use the assignment operator with the increment or decrement operator, though doing so carelessly can be extremely confusing. See Listing 3.16 and Output 3.10 for an example.

LISTING 3.16: Using the Post-Increment Operator

```
int count = 123;
int result;
result = count++;
System.Console.WriteLine(
  $"result = {result} and count = {count}");
```

OUTPUT 3.10

```
result = 123 and count = 124
```

You might be surprised that `result` was assigned the value that was `count` *before* `count` was incremented. Where you place the increment or decrement operator determines whether the assigned value should be the value of the operand before or after the calculation. If you want the value of `result` to be the value assigned to `count`, you need to place the operator before the variable being incremented, as shown in Listing 3.17.

LISTING 3.17: Using the Pre-Increment Operator

```
int count = 123;
int result;
result = ++count;
System.Console.WriteLine(
  $"result = {result} and count = {count}");
```

Output 3.11 shows the results of Listing 3.17.

OUTPUT 3.11

```
result = 124 and count = 124
```

In this example, the increment operator appears before the operand, so the result of the expression is the value assigned to the variable after the increment. If `count` is `123`, `++count` will assign `124` to `count` and produce the result `124`. By contrast, the postfix increment operator `count++` assigns `124` to `count` and produces the value that `count` held before the increment: `123`. Regardless of whether the operator is postfix or prefix, the variable `count` will be incremented before the value is produced; the only difference is which value is produced. The difference between prefix and postfix behavior is illustrated in Listing 3.18. The resultant output is shown in Output 3.12.

LISTING 3.18: Comparing the Prefix and Postfix Increment Operators

```
class IncrementExample
{
  static void Main()
  {
      int x = 123;
      // Displays 123, 124, 125.
      System.Console.WriteLine($"{x++}, {x++}, {x}");
      // x now contains the value 125.
      // Displays 126, 127, 127.
      System.Console.WriteLine($"{++x}, {++x}, {x}");
      // x now contains the value 127.
  }
}
```

OUTPUT 3.12

```
123, 124, 125
126, 127, 127
```

As Listing 3.18 demonstrates, where the increment and decrement operators appear relative to the operand can affect the result produced by the expression. The result of the prefix operators is the value that the variable had before it was incremented or decremented. The result of the postfix

operators is the value that the variable had after it was incremented or decremented. Use caution when embedding these operators in the middle of a statement. When in doubt as to what will happen, use these operators independently, placing them within their own statements. This way, the code is also more readable and there is no mistaking the intention.

Language Contrast: C++—Implementation-Defined Behavior

Earlier we discussed how the operands in an expression can be evaluated in any order in C++, whereas they are always evaluated from left to right in C#. Similarly, in C++ an implementation may legally perform the side effects of increments and decrements in any order. For example, in C++ a call of the form `M(x++, x++)`, where `x` begins as `1`, can legally call either `M(1,2)` or `M(2,1)` at the whim of the compiler. In contrast, C# will always call `M(1,2)` because C# makes two guarantees: (1) The arguments to a call are always computed from left to right, and (2) the assignment of the incremented value to the variable always happens before the value of the expression is used. C++ makes neither guarantee.

Guidelines

AVOID confusing usage of the increment and decrement operators.

DO be cautious when porting code between C, C++, and C# that uses increment and decrement operators; C and C++ implementations need not follow the same rules as C#.

ADVANCED TOPIC

Thread-Safe Incrementing and Decrementing

In spite of the brevity of the increment and decrement operators, these operators are not atomic. A thread context switch can occur during the execution of the operator and can cause a race condition. You could use a `lock` statement to prevent the race condition. However, for

simple increments and decrements, a less expensive alternative is to use the thread-safe `Increment()` and `Decrement()` methods from the `System.Threading.Interlocked` class. These methods rely on processor functions for performing fast thread-safe increments and decrements. See Chapter 19 for more details.

Constant Expressions and Constant Locals

The preceding chapter discussed literal values, or values embedded directly into the code. It is possible to combine multiple literal values in a **constant expression** using operators. By definition, a constant expression is one that the C# compiler can evaluate at compile time (instead of evaluating it when the program runs) because it is composed entirely of constant operands. Constant expressions can then be used to initialize constant locals, which allow you to give a name to a constant value (similar to the way local variables allow you to give a name to a storage location). For example, the computation of the number of seconds in a day can be a constant expression that is then used in other expressions by name.

The `const` keyword in Listing 3.19 declares a constant local. Since a constant local is by definition the opposite of a **variable**—"constant" means "not able to vary"—any attempt to modify the value later in the code would result in a compile-time error.

Guidelines

DO NOT use a constant for any value that can possibly change over time. The value of pi and the number of protons in an atom of gold are constants; the price of gold, the name of your company, and the version number of your program can change.

Note that the expression assigned to `secondsPerWeek` in Listing 3.19 is a constant expression because all the operands in the expression are also constants.

LISTING 3.19: Declaring a Constant

```
// ...
public long Main()
{
    const int secondsPerDay = 60 * 60 * 24;
    const int secondsPerWeek = secondsPerDay * 7;
    // ...
}
```

Constant Expression

Constant

Introducing Flow Control

Later in this chapter is a code listing (Listing 3.45) that shows a simple way to view a number in its binary form. Even such a simple program, however, cannot be written without using control flow statements. Such statements control the execution path of the program. This section discusses how to change the order of statement execution based on conditional checks. Later on, you will learn how to execute statement groups repeatedly through loop constructs.

A summary of the control flow statements appears in Table 3.1. Note that the General Syntax Structure column indicates common statement use, not the complete lexical structure. An `embedded-statement` in Table 3.1 may be any statement other than a labeled statement or a declaration, but it is typically a block statement.

Each C# control flow statement in Table 3.1 appears in the tic-tac-toe[3] program and is available in Appendix B and for download with the rest of the source code listings from the book. The program displays the tic-tac-toe board, prompts each player, and updates with each move.

The remainder of this chapter looks at each statement in more detail. After covering the `if` statement, it introduces code blocks, scope, Boolean expressions, and bitwise operators before continuing with the remaining control flow statements. Readers who find Table 3.1 familiar because of C#'s similarities to other languages can jump ahead to the section titled "C# Preprocessor Directives" or skip to the "Summary" section at the end of the chapter.

3. Known as noughts and crosses to readers outside the United States.

TABLE 3.1: Control Flow Statements

Statement	General Syntax Structure	Example

`if` statement	`if(`*`boolean-expression`*`)`
	  `embedded-statement`
		`if (input == "quit")`
		`{`
		`  System.Console.WriteLine(`
		`  "Game end");`
		`  return;`
		`}`

	`if(`*`boolean-expression`*`)`
	  `embedded-statement`
	`else`
	  `embedded-statement`
		`if (input == "quit")`
		`{`
		`  System.Console.WriteLine(`
		`  "Game end");`
		`  return;`
		`}`
		`else`
		`  GetNextMove();`

`while` statement	`while(`*`boolean-expression`*`)`
	  `embedded-statement`
		`while(`*`count < total`*`)`
		`{`
		`  System.Console.WriteLine(`
		`  "count = {count}");`
		`  count++;`
		`}`

`do while` statement	`do`
	  `embedded-statement` `while(`*`boolean-expression`*`);`
		`do`
		`{`
		`  System.Console.WriteLine(`
		`  "Enter name:");`
		`  input =`
		`  System.Console.ReadLine();`
		`}`
		`while(`*`input != "exit"`*`);`

TABLE 3.1: Control Flow Statements, (*continued*)

Statement	General Syntax Structure	Example

for statement	`for(`*`for-initializer;`*
	`  `*`boolean-expression;`*
	`  for-iterator)`
	`  `*`embedded-statement`*
		`for (int count = 1;`
		`  count <= 10;`
		`  count++)`
		`{`
		`  System.Console.WriteLine(`
		`  "count = {count}");`
		`}`

foreach statement	`foreach(`*`type identifier`* `in`
	`  `*`expression`*`)`
	`  `*`embedded-statement`*
		`foreach (char letter in email)`
		`{`
		`  if(!insideDomain)`
		`  {`
		`  if (letter == '@')`
		`  {`
		`  insideDomain = true;`
		`  }`
		`  continue;`
		`  }`
		`  System.Console.Write(`
		`  letter);`
		`}`

continue statement	`continue;`	

continues

TABLE 3.1: Control Flow Statements, (*continued*)

Statement	General Syntax Structure	Example

switch statement	`switch(governing-type-expression)`
	`{`
	`  ...`
	`  case const-expression:`
	`  statement-list`
	`  jump-statement`
	`  default:`
	`  statement-list`
	`  jump-statement`
	`}`
		`switch(input)`
		`{`
		`  case "exit":`
		`  case "quit":`
		`  System.Console.WriteLine(`
		`  "Exiting app....");`
		`  break;`
		`  case "restart":`
		`  Reset();`
		`  goto case "start";`
		`  case "start":`
		`  GetMove();`
		`  break;`
		`  default:`
		`  System.Console.WriteLine(`
		`  input);`
		`  break;`
		`}`

break statement	`break;`	
goto statement	`goto identifier;`	
	`goto case const-expression;`	
	`goto default;`	

if Statement

The `if` statement is one of the most common statements in C#. It evaluates a **Boolean expression** (an expression that results in either `true` or `false`) called the **condition.** If the condition is `true`, the **consequence statement** is executed. An `if` statement may optionally have an `else` clause that contains an **alternative statement** to be executed if the condition is `false`. The general form is as follows:

```
if (condition)
  consequence-statement
else
  alternative-statement
```

LISTING 3.20: `if/else` Statement Example

```
class TicTacToe      // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      string input;

      // Prompt the user to select a 1- or 2-player game.
      System.Console.Write(
          "1 - Play against the computer\n" +
          "2 - Play against another player.\n" +
          "Choose:"
      );
      input = System.Console.ReadLine();

      if(input=="1")
          // The user selected to play the computer.
          System.Console.WriteLine(
              "Play against computer selected.");
      else
          // Default to 2 players (even if user didn't enter 2).
          System.Console.WriteLine(
              "Play against another player.");
  }
}
```

In Listing 3.20, if the user enters `1`, the program displays `"Play against computer selected."` Otherwise, it displays `"Play against another player."`

Nested if

Sometimes code requires multiple `if` statements. The code in Listing 3.21 first determines whether the user has chosen to exit by entering a number less than or equal to `0`; if not, it checks whether the user knows the maximum number of turns in tic-tac-toe.

LISTING 3.21: Nested if Statements

```
1.  class TicTacToeTrivia
2.  {
3.    static void Main()
4.    {
5.        int input;    // Declare a variable to store the input.
6.
7.        System.Console.Write(
8.            "What is the maximum number " +
9.            "of turns in tic-tac-toe?" +
10.           "(Enter 0 to exit.): ");
11.
12.       // int.Parse() converts the ReadLine()
13.       // return to an int data type.
14.       input = int.Parse(System.Console.ReadLine());
15.
16.       if (input <= 0) // line 16
17.               // Input is less than or equal to 0.
18.           System.Console.WriteLine("Exiting...");
19.       else
20.           if (input < 9)  // line 20
21.               // Input is less than 9.
22.               System.Console.WriteLine(
23.                   $"Tic-tac-toe has more than {input}" +
24.                   " maximum turns.");
25.           else
26.               if(input > 9) // line 26
27.                   // Input is greater than 9.
28.                   System.Console.WriteLine(
29.                       $"Tic-tac-toe has fewer than {input}" +
30.                       " maximum turns.");
31.               else
32.                   // Input equals 9.
33.                   System.Console.WriteLine( // line 33
34.                       "Correct, tic-tac-toe " +
35.                       "has a maximum of 9 turns.");
36.   }
37. }
```

Output 3.13 shows the results of Listing 3.21.

OUTPUT 3.13

```
What is the maximum number of turns in tic-tac-toe? (Enter 0 to exit.): 9
Correct, tic-tac-toe has a maximum of 9 turns.
```

Assume the user enters `9` when prompted at line 14. Here is the execution path:

1. *Line 16:* Check if input is less than 0. Since it is not, jump to line 20.
2. *Line 20:* Check if input is less than 9. Since it is not, jump to line 26.
3. *Line 26:* Check if input is greater than 9. Since it is not, jump to line 33.
4. *Line 33:* Display that the answer was correct.

Listing 3.21 contains nested `if` statements. To clarify the nesting, the lines are indented. However, as you learned in Chapter 1, whitespace does not affect the execution path. If this code was written without the indenting and without the newlines, the execution would be the same. The code that appears in the nested `if` statement in Listing 3.22 is equivalent to Listing 3.21.

LISTING 3.22: `if/else` Formatted Sequentially

```
if (input < 0)
  System.Console.WriteLine("Exiting...");
else if (input < 9)
  System.Console.WriteLine(
      $"Tic-tac-toe has more than {input}" +
      " maximum turns.");
else if(input < 9)
  System.Console.WriteLine(
      $"Tic-tac-toe has less than {input}" +
      " maximum turns.");
else
  System.Console.WriteLine(
      "Correct, tic-tac-toe has a maximum " +
      " of 9 turns.");
```

Although the latter format is more common, in each situation you should use the format that results in the clearest code.

Both of the `if` statement listings omit the braces. However, as discussed next, this is not in accordance with the guidelines, which advocate the use of code blocks except, perhaps, in the simplest of single-line scenarios.

Code Blocks ({})

In the previous `if` statement examples, only one statement follows `if` and `else`: a single `System.Console.WriteLine()`, similar to Listing 3.23.

Listing 3.23: `if` Statement with No Code Block

```
if(input < 9)
  System.Console.WriteLine("Exiting");
```

With curly braces, however, we can combine statements into a single statement called a **block statement** or **code block,** allowing the grouping of multiple statements into a single statement that is the consequence. Take, for example, the highlighted code block in the radius calculation in Listing 3.24.

Listing 3.24: `if` Statement Followed by a Code Block

```
class CircleAreaCalculator
{
  static void Main()
  {
      double radius;  // Declare a variable to store the radius.
      double area;    // Declare a variable to store the area.

      System.Console.Write("Enter the radius of the circle: ");

      // double.Parse converts the ReadLine()
      // return to a double.
      radius = double.Parse(System.Console.ReadLine());
      if(radius >= 0)
      {
          // Calculate the area of the circle.
          area = Math.PI * radius * radius;
          System.Console.WriteLine(
              $"The area of the circle is: { area : 0.00 }");
      }
      else
      {
          System.Console.WriteLine(
              $"{ radius } is not a valid radius.");
      }
  }
}
```

Output 3.14 shows the results of Listing 3.24.

OUTPUT 3.14

```
Enter the radius of the circle: 3
The area of the circle is: 28.27
```

In this example, the `if` statement checks whether the `radius` is positive. If so, the area of the circle is calculated and displayed; otherwise, an invalid radius message is displayed.

Notice that in this example, two statements follow the first `if`. However, these two statements appear within curly braces. The curly braces combine the statements into a **code block, which is itself a single statement.**

If you omit the curly braces that create a code block in Listing 3.24, only the statement immediately following the Boolean expression executes conditionally. Subsequent statements will execute regardless of the `if` statement's Boolean expression. The invalid code is shown in Listing 3.25.

LISTING 3.25: Relying on Indentation, Resulting in Invalid Code

```
if(radius >= 0)
  area = Math.PI * radius *radius;
  System.Console.WriteLine(
      $"The area of the circle is: { area:0.00}");
```

In C#, indentation is used solely to enhance the code readability. The compiler ignores it, so the previous code is semantically equivalent to Listing 3.26.

LISTING 3.26: Semantically Equivalent to Listing 3.25

```
if(radius >= 0)
{
  area = Math.PI * radius * radius;
}
System.Console.WriteLine(
    $"The area of the circle is:{ area:0.00}");
```

Programmers should take great care to avoid subtle bugs such as this, perhaps even going so far as to always include a code block after a control flow statement, even if there is only one statement. A widely accepted coding guideline is to avoid omitting braces, except possibly for the simplest of single-line `if` statements.

Although unusual, it is possible to have a code block that is not lexically a direct part of a control flow statement. In other words, placing curly braces on their own (without a conditional or loop, for example) is legal syntax.

In Listing 3.25 and Listing 3.26, the value of pi was represented by the `PI` constant in the `System.Math` class. Instead of hardcoding a value, such as 3.14 for constants such as pi and Euler's constant (e), code should use `System.Math.PI` and `System.Math.E`.

Guidelines

AVOID omitting braces, except for the simplest of single-line `if` statements.

Code Blocks, Scopes, and Declaration Spaces

Code blocks are often referred to as "scopes," but the two terms are not exactly interchangeable. The **scope** of a named thing is the region of source code in which it is legal to refer to the thing by its unqualified name. The scope of a local variable, for example, is exactly the text of the code block that encloses it, which explains why it is common to refer to code blocks as "scopes."

Scopes are often confused with declaration spaces. A **declaration space** is a logical container of named things in which two things may not have the same name. A code block defines not only a scope, but also a local variable declaration space. It is illegal for two local variable declarations with the same name to appear in the same declaration space. Similarly, it is not possible to declare two methods with the signature of `Main()` within the same class. (This rule is relaxed somewhat for methods: Two methods may have the same name in a declaration space provided that they have different signatures. The signature of a method includes its name and the number and types of its parameters.) Within a block, a local variable can be mentioned by name and must be the unique thing that is declared with that name in the block. Outside the declaring block, there is no way to refer to a local variable by its name; the local variable is said to be "out of scope" outside the block.

In summary, a scope is used to determine what thing a name refers to; a declaration space determines when two things declared with the same name conflict with each other. In Listing 3.27, declaring the local variable `message` inside the block statement embedded in the `if` statement restricts its scope to the block statement only; the local variable is "out of scope" when its name is used later on in the method. To avoid an error, you must declare the variable outside the block.

LISTING 3.27: Variables Inaccessible outside Their Scope

```
class Program
{
   static void Main(string[] args)
   {
       int playerCount;
       System.Console.Write(
           "Enter the number of players (1 or 2):");
       playerCount = int.Parse(System.Console.ReadLine());
       if (playerCount != 1 && playerCount != 2)
       {
           string message =
               "You entered an invalid number of players.";
       }
       else
       {
           // ...
       }
       // Error:  message is not in scope.
       System.Console.WriteLine(message);
   }
}
```

Output 3.15 shows the results of Listing 3.27.

OUTPUT 3.15

```
...

...\Program.cs(18,26): error CS0103: The name 'message' does not exist
in the current context
```

The declaration space in which a local variable's name must be unique encompasses all the child code blocks textually enclosed within the block that originally declared the local. The C# compiler prevents the name of a local variable declared immediately within a method code block (or as a

parameter) from being reused within a child code block. In Listing 3.27, because `args` and `playerCount` are declared within the method code block, they cannot be declared again anywhere within the method.

The name `message` refers to this local variable throughout the scope of the local variable—that is, the block immediately enclosing the declaration. Similarly, `playerCount` refers to the same variable throughout the block containing the declaration, including within both of the child blocks that are the consequence and the alternative of the `if` statement.

Language Contrast: C++—Local Variable Scope

In C++, a local variable declared in a block is in scope from the point of the declaration statement through the end of the block. Thus an attempt to refer to the local variable before its declaration will fail to find the local variable because that variable is not in scope. If there is another thing with that name "in scope," the C++ language will resolve the name to that thing, which might not be what you intended. In C#, the rule is subtly different: A local variable is in scope throughout the entire block in which it is declared, but it is illegal to refer to the local variable before its declaration. That is, the attempt to find the local variable will succeed, and the usage will then be treated as an error. This is just one of C#'s many rules that attempt to prevent errors common in C++ programs.

Boolean Expressions

The parenthesized condition of the `if` statement is a **Boolean expression.** In Listing 3.28, the condition is highlighted.

Listing 3.28: Boolean Expression

```
if (input < 9)
{
    // Input is less than 9.
    System.Console.WriteLine(
        $"Tic-tac-toe has more than { input }" +
         " maximum turns.");
}
// ...
```

Boolean expressions appear within many control flow statements. Their key characteristic is that they always evaluate to `true` or `false`. For `input < 9` to be allowed as a Boolean expression, it must result in a `bool`. The compiler disallows `x = 42`, for example, because this expression assigns `x` and results in the value that was assigned, instead of checking whether the value of the variable is `42`.

Language Contrast: C++—Mistakenly Using = in Place of ==

C# eliminates a coding error commonly found in C and C++. In C++, Listing 3.29 is allowed.

LISTING 3.29: C++, But Not C#, Allows Assignment As a Condition

```
if (input = 9)    // Allowed in C++, not in C#.
  System.Console.WriteLine(
      "Correct, tic-tac-toe has a maximum of 9 turns.");
```

Although at first glance this code appears to check whether `input` equals `9`, Chapter 1 showed that = represents the assignment operator, not a check for equality. The return from the assignment operator is the value assigned to the variable—in this case, `9`. However, `9` is an `int`, and as such it does not qualify as a Boolean expression and is not allowed by the C# compiler. The C and C++ languages treat integers that are nonzero as `true`, and integers that are zero as `false`. C#, by contrast, requires that the condition actually be of a Boolean type; integers are not allowed.

Relational and Equality Operators

Relational and **equality** operators determine whether a value is greater than, less than, or equal to another value. Table 3.2 lists all the relational and equality operators. All are binary operators.

The C# syntax for equality uses ==, just as many other programming languages do. For example, to determine whether `input` equals `9`, you use `input == 9`. The equality operator uses two equal signs to distinguish it from the assignment operator, =. The exclamation point signifies NOT in C#, so to test for inequality you use the inequality operator, `!=`.

TABLE 3.2: Relational and Equality Operators

Operator	Description	Example
<	Less than	`input<9;`
>	Greater than	`input>9;`
<=	Less than or equal to	`input<=9;`
>=	Greater than or equal to	`input>=9;`
==	Equality operator	`input==9;`
!=	Inequality operator	`input!=9;`

Relational and equality operators always produce a `bool` value, as shown in Listing 3.30.

LISTING 3.30: Assigning the Result of a Relational Operator to a bool Variable

```
bool result = 70 > 7;
```

In the tic-tac-toe program (see Appendix B), you use the equality operator to determine whether a user has quit. The Boolean expression of Listing 3.31 includes an OR (`||`) logical operator, which the next section discusses in detail.

LISTING 3.31: Using the Equality Operator in a Boolean Expression

```
if (input == "" || input == "quit")
{
  System.Console.WriteLine($"Player {currentPlayer} quit!!");
  break;
}
```

Logical Boolean Operators

The **logical operators** have Boolean operands and produce a Boolean result. Logical operators allow you to combine multiple Boolean expressions to form more complex Boolean expressions. The logical operators are `|`, `||`, `&`, `&&`, and `^`, corresponding to OR, AND, and exclusive OR. The `|` and `&` versions of OR and AND are rarely used for Boolean logic, for reasons which we discuss in this section.

OR Operator (||)

In Listing 3.31, if the user enters `quit` or presses the Enter key without typing in a value, it is assumed that she wants to exit the program. To enable two ways for the user to resign, you can use the logical OR operator, `||`. The `||` operator evaluates Boolean expressions and results in a `true` value if *either* operand is `true` (see Listing 3.32).

LISTING 3.32: Using the OR Operator

```
if ((hourOfTheDay > 23) || (hourOfTheDay < 0))
  System.Console.WriteLine("The time you entered is invalid.");
```

It is not necessary to evaluate both sides of an OR expression, because if either operand is `true`, the result is known to be `true` regardless of the value of the other operand. Like all operators in C#, the left operand is evaluated before the right one, so if the left portion of the expression evaluates to `true`, the right portion is ignored. In the example in Listing 3.32, if `hourOfTheDay` has the value `33`, then `(hourOfTheDay > 23)` will evaluate to `true` and the OR operator will ignore the second half of the expression, **short-circuiting** it. Short-circuiting an expression also occurs with the Boolean AND operator. (Note that the parentheses are not necessary here; the logical operators are of higher precedence than the relational operators. However, it is clearer to the novice reader to parenthesize the subexpressions for clarity.)

AND Operator (&&)

The Boolean AND operator, `&&`, evaluates to `true` only if both operands evaluate to `true`. If either operand is `false`, the result will be `false`. Listing 3.33 writes a message if the given variable is both greater than 10 and less than 24.[4] Similarly to the OR operator, the AND operator will not always evaluate the right side of the expression. If the left operand is determined to be `false`, the overall result will be `false` regardless of the value of the right operand, so the runtime skips evaluating the right operand.

LISTING 3.33: Using the AND Operator

```
if ((10 < hourOfTheDay) && (hourOfTheDay < 24))
  System.Console.WriteLine(
  "Hi-Ho, Hi-Ho, it's off to work we go.");
```

4. The typical hours that programmers work each day.

Exclusive OR Operator (^)

The caret symbol, ^, is the "exclusive OR" (XOR) operator. When applied to two Boolean operands, the XOR operator returns `true` only if exactly one of the operands is true, as shown in Table 3.3.

TABLE 3.3: Conditional Values for the XOR Operator

Left Operand	Right Operand	Result
True	True	False
True	False	True
False	True	True
False	False	False

Unlike the Boolean AND and Boolean OR operators, the Boolean XOR operator does not short-circuit: It always checks both operands, because the result cannot be determined unless the values of both operands are known. Note that the XOR operator is exactly the same as the Boolean inequality operator.

Logical Negation Operator (!)

The **logical negation operator, or NOT operator,** `!`, inverts a `bool` value. This operator is a unary operator, meaning it requires only one operand. Listing 3.34 demonstrates how it works, and Output 3.16 shows the result.

LISTING 3.34: Using the Logical Negation Operator

```
bool valid = false;
bool result = !valid;
// Displays "result = True".
System.Console.WriteLine($"result = { result }");
```

OUTPUT 3.16

```
result = True
```

At the beginning of Listing 3.34, `valid` is set to `false`. You then use the negation operator on `valid` and assign the value to `result`.

Conditional Operator (?:)

In place of an `if-else` statement used to select one of two values, you can use the **conditional** operator. The conditional operator uses both a question mark and a colon; the general format is as follows:

```
condition ? consequence : alternative
```

The conditional operator is a "ternary" operator because it has three operands: `condition`, `consequence`, and `alternative`. (As it is the only ternary operator in C#, it is often called "the ternary operator," but it is clearer to refer to it by its name than by the number of operands it takes.) Like the logical operators, the conditional operator uses a form of short-circuiting. If the condition evaluates to `true`, the conditional operator evaluates only `consequence`. If the conditional evaluates to `false`, it evaluates only `alternative`. The result of the operator is the evaluated expression.

Listing 3.35 illustrates the use of the conditional operator. The full listing of this program appears in Appendix B.

Listing 3.35: Conditional Operator

```
class TicTacToe
{
  static string Main()
  {
      // Initially set the currentPlayer to Player 1
      int currentPlayer = 1;

      // ...

      for (int turn = 1; turn <= 10; turn++)
      {
         // ...

         // Switch players
         currentPlayer = (currentPlayer == 2) ? 1 : 2;
      }
  }
}
```

The program swaps the current player. To do so, it checks whether the current value is `2`. This is the "conditional" portion of the conditional expression. If the result of the condition is `true`, the conditional operator results in the "consequence" value `1`. Otherwise, it results in the "alternative"

value 2. Unlike an `if` statement, the result of the conditional operator must be assigned (or passed as a parameter); it cannot appear as an entire statement on its own.

Guidelines

CONSIDER using an `if/else` statement instead of an overly complicated conditional expression.

The C# language requires that the consequence and alternative expressions in a conditional operator be typed consistently, and that the consistent type be determined without examination of the surrounding context of the expression. For example, `f ? "abc" : 123` is not a legal conditional expression because the consequence and alternative are a string and a number, neither of which is convertible to the other. Even if you say `object result = f ? "abc" : 123;` the C# compiler will flag this expression as illegal because the type that is consistent with both expressions (that is, `object`) is found outside the conditional expression.

Begin 2.0

Null-Coalescing Operator (??)

The **null-coalescing operator** is a concise way to express "If this value is null, then use this other value." It has the following form:

expression1 `??` *expression2*

The null-coalescing operator also uses a form of short-circuiting. If `expression1` is not null, its value is the result of the operation and the other expression is not evaluated. If `expression1` does evaluate to null, the value of `expression2` is the result of the operator. Unlike the conditional operator, the null-coalescing operator is a binary operator.

Listing 3.36 illustrates the use of the null-coalescing operator.

LISTING 3.36: Null-Coalescing Operator

```
string fileName = GetFileName();
// ...
string fullName = fileName ?? "default.txt";
// ...
```

In this listing, we use the null-coalescing operator to set `fullName` to `"default.txt"` if `fileName` is null. If `fileName` is not null, `fullName` is simply assigned the value of `fileName`.

The null-coalescing operator "chains" nicely. For example, an expression of the form `x ?? y ?? z` results in `x` if `x` is not null; otherwise, it results in `y` if `y` is not null; otherwise, it results in `z`. That is, it goes from left to right and picks out the first non-null expression, or uses the last expression if all the previous expressions were null.

The null-coalescing operator was added to C# in version 2.0, along with nullable value types. This operator works on both operands of nullable value types and reference types.

End 2.0

Null-Conditional Operator (?.)

Begin 6.0

Whenever you invoke a method on a value that is null, the runtime will throw a `System.NullReferenceException`, which almost always indicates an error in the programming logic. In recognition of the frequency of this pattern (that is, checking for null before invoking a member), C# 6.0 introduces the "?." operator, known as the null-conditional operator:

Listing 3.37: Null-Conditional Operator

```
class Program
{
  static void Main(string[] args)
  {
      if (args?.Length == 0)
      {
          System.Console.WriteLine(
              "ERROR: File missing. "
              + "Use:\n\tfind.exe file:<filename>");
      }
      else
      {
          if (args[0]?.ToLower().StartsWith("file:")??false)
          {
              string fileName = args[0]?.Remove(0, 5);
              // ...
          }
      }
  }
}
```

The null-conditional operator checks whether the operand (the first `args` in Listing 3.37) is null prior to invoking the method or property (`Length` in the first example in this listing). The logically equivalent explicit code would be the following (although in the C# 6.0 syntax the value of `args` is evaluated only once):

6.0

```
(args != null) ? (int?)args.Length : null
```

What makes the null-conditional operator especially convenient is that it can be chained. If, for example, you invoke `args[0]?.ToLower()` `.StartsWith("file:")`, both `ToLower()` and `StartsWith()` will be invoked only if `args[0]` is not null. When expressions are chained, if the first operand is null, the expression evaluation is short-circuited, and no further invocation within the expression call chain will occur.

Be careful, however, that you don't unintentionally neglect additional null-conditional operators. Consider, for example, what would happen if (hypothetically, in this case) `args[0]?.ToLower()` could also return null. In this scenario, a `NullReferenceException` would occur upon invocation of `StartsWith()`. This doesn't mean you must use a chain of null-conditional operators, but rather that you should be intentional about the logic. In this example, because `ToLower()` can never be null, no additional null-conditional operator is necessary.

An important thing to note about the null-conditional operator is that, when utilized with a member that returns a value type, it always returns a nullable version of that type. For example, `args?.Length` returns an `int?`, not simply an `int`. Similarly, `args[0]?.ToLower().StartsWith("file:")` returns a `bool?` (a `Nullable<bool>`). Also, because an `if` statement requires a `bool` data type, it is necessary to follow the `StartsWith()` expression with the null-coalescing operator (`??`).

Although perhaps a little peculiar (in comparison to other operator behavior), the return of a nullable value type is produced only at the end of the call chain. Consequently, calling the dot (".") operator on `Length` allows invocation of only `int` (not `int?`) members. However, encapsulating `args?.Length` in parentheses—thereby forcing the `int?` result via parentheses operator precedence—will invoke the `int?` return and make the `Nullable<T>` specific members (`HasValue` and `Value`) available.

Null-conditional operators can also be used in combination with an index operator, as shown in Listing 3.38.

LISTING 3.38: Null-Conditional Operator with Index Operator

```
class Program
{
  public static void Main(string[] args)
  {
      // CAUTION: args?.Length not verified.
      string directoryPath = args?[0];
      string searchPattern = args?[1];
      // ...
  }
}
```


In this listing, the first and second elements of `args` are assigned to their respective variables only if `args` is not null. If it is, null will be assigned instead.

Unfortunately, this example is naïve, if not dangerous, because the null-conditional operator gives a false sense of security, implying that if `args` isn't null, then the element must exist. Of course, this isn't the case: The element may not exist even if `args` isn't null. Also, because checking for the element count with `args?.Length` verifies that `args` isn't null, you never really need to use the null-conditional operator when indexing the collection after checking the length.

In conclusion, you should avoid using the null-conditional operator in combination with the index operator if the index operator throws an `IndexOutOfRangeException` for nonexistent indexes. Doing so leads to a false sense of code validity.

ADVANCED TOPIC

Leveraging the Null-Conditional Operator with Delegates

The null-conditional operator is a great feature on its own. However, using it in combination with a delegate invocation resolves a C# pain point that has existed since C# 1.0. Notice in the code near the top of the next page how the `PropertyChange` event handler is assigned to a local copy (`propertyChanged`) before we check the value for null and finally fire the event. This is the easiest thread-safe way to invoke events without running the risk that an event unsubscribe will occur between the time when the check for null occurs and the time when the event is fired. Unfortunately, this approach is non-intuitive, and frequently code

neglects to follow this pattern—with the result of throwing inconsistent `NullReferenceExceptions`. Fortunately, with the introduction of the null-conditional operator in C# 6.0, this issue has been resolved.

With C# 6.0, the check for a delegate value changes from

```
PropertyChangedEventHandler propertyChanged =
  PropertyChanged;
if (propertyChanged != null)
{
  propertyChanged(this,
      new PropertyChangedEventArgs(nameof(Name)));
}
```

to simply

```
PropertyChanged?.Invoke(propertyChanged(
  this, new PropertyChangedEventArgs(nameof(Name)));
```

End 6.0

Because an event is just a delegate, the pattern of invoking a delegate via the null-conditional operator and an `Invoke()` is always possible.

Bitwise Operators (<<, >>, |, &, ^, ~)

An additional set of operators that is common to virtually all programming languages is the set of operators for manipulating values in their binary formats: the bit operators.

BEGINNER TOPIC

Bits and Bytes

All values within a computer are represented in a binary format of 1s and 0s, called **binary digits** (**bits**). Bits are grouped together in sets of eight, called **bytes.** In a byte, each successive bit corresponds to a value of 2 raised to a power, starting from 2^0 on the right and moving to 2^7 on the left, as shown in Figure 3.1.

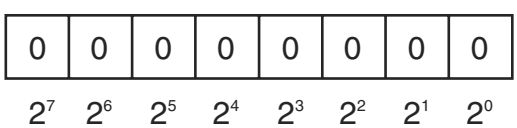


FIGURE 3.1: Corresponding Placeholder Values

In many scenarios, particularly when dealing with low-level or system services, information is retrieved as binary data. To manipulate these devices and services, you need to perform manipulations of binary data.

In Figure 3.2, each box corresponds to a value of 2 raised to the power shown. The value of the byte (8-bit number) is the sum of the powers of 2 of all of the eight bits that are set to 1.

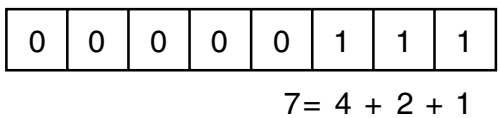


FIGURE 3.2: Calculating the Value of an Unsigned Byte

The binary translation just described is significantly different for signed numbers. Signed numbers (`long`, `short`, `int`) are represented using a "two's complement" notation. This practice ensures that addition continues to work when adding a negative number to a positive number, as though both were positive operands. With this notation, negative numbers behave differently from positive numbers. Negative numbers are identified by a 1 in the leftmost location. If the leftmost location contains a 1, you add the locations with 0s rather than the locations with 1s. Each location corresponds to the negative power of 2 value. Furthermore, from the result, it is also necessary to subtract 1. This is demonstrated in Figure 3.3.

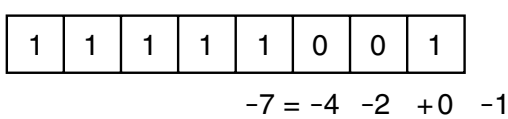


FIGURE 3.3: Calculating the Value of a Signed Byte

Therefore, `1111 1111 1111 1111` corresponds to –1, and `1111 1111 1111 1001` holds the value –7. The binary representation `1000 0000 0000 0000` corresponds to the lowest negative value that a 16-bit integer can hold.

Shift Operators (<<, >>, <<=, >>=)

Sometimes you want to shift the binary value of a number to the right or left. In executing a left shift, all bits in a number's binary representation are shifted to the left by the number of locations specified by the operand on the right of the shift operator. Zeroes are then used to backfill the locations on the right side of the binary number. A right-shift operator does almost the same thing in the opposite direction. However, if the number is a negative value of a signed type, the values used to backfill the left side of the binary number are 1s and not 0s. The shift operators are `>>` and `<<`, known as the right-shift and left-shift operators, respectively. In addition, there are combined shift and assignment operators, `<<=` and `>>=`.

Consider the following example. Suppose you had the `int` value `-7`, which would have a binary representation of `1111 1111 1111 1111 1111 1111 1111 1001`. In Listing 3.39, you right-shift the binary representation of the number –7 by two locations.

Listing 3.39: Using the Right-Shift Operator

```
int x;
x = (-7 >> 2); // 11111111111111111111111111111001 becomes
               // 11111111111111111111111111111110
// Write out "x is -2."
System.Console.WriteLine($"x = { x }.");
```

Output 3.17 shows the results of Listing 3.39.

Output 3.17

```
x = -2.
```

Because of the right shift, the value of the bit in the rightmost location has "dropped off" the edge and the negative bit indicator on the left shifts by two locations to be replaced with 1s. The result is `-2`.

Although legend has it that `x << 2` is faster than `x * 4`, you should not use bit-shift operators for multiplication or division. This difference might have held true for certain C compilers in the 1970s, but modern compilers and modern microprocessors are perfectly capable of optimizing arithmetic. Using shifting for multiplication or division is confusing and frequently leads to errors when code maintainers forget that the shift operators are lower precedence than the arithmetic operators.

Bitwise Operators (&, |, ^)

In some instances, you might need to perform logical operations, such as AND, OR, and XOR, on a bit-by-bit basis for two operands. You do this via the &, |, and ^ operators, respectively.

BEGINNER TOPIC

Logical Operators Explained

If you have two numbers, as shown in Figure 3.4, the bitwise operations will compare the values of the locations beginning at the leftmost significant value and continuing right until the end. The value of "1" in a location is treated as "true," and the value of "0" in a location is treated as "false."

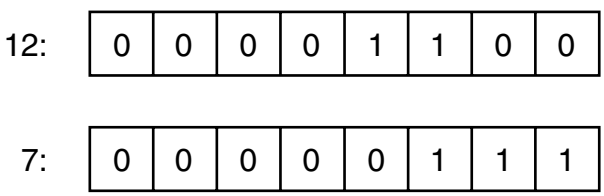


FIGURE 3.4: The Numbers 12 and 7 Represented in Binary

Therefore, the bitwise AND of the two values in Figure 3.4 would entail the bit-by-bit comparison of bits in the first operand (12) with the bits in the second operand (7), resulting in the binary value `000000100`, which is 4. Alternatively, a bitwise OR of the two values would produce `00001111`, the binary equivalent of 15. The XOR result would be `00001011`, or decimal 11.

Listing 3.40 demonstrates the use of these bitwise operators. The results of Listing 3.40 appear in Output 3.18.

LISTING 3.40: Using Bitwise Operators

```
byte and, or, xor;
and = 12 & 7;   // and = 4
or = 12 | 7;    // or = 15
xor = 12 ^ 7;   // xor = 11
System.Console.WriteLine(
    $"and = { and } \nor = { or }\nxor = { xor }");
```

OUTPUT 3.18

```
and = 4
or = 15
xor = 11
```

In Listing 3.40, the value 7 is the **mask**; it is used to expose or eliminate specific bits within the first operand using the particular operator expression. Note that, unlike the AND (&&) operator, the & operator always evaluates *both* sides even if the left portion is false. Similarly, the | version of the OR operator is *not* "short-circuiting." It always evaluates both operands even if the left operand is true. The bit versions of the AND and OR operators, therefore, are not short-circuiting.

To convert a number to its binary representation, you need to iterate across each bit in a number. Listing 3.41 is an example of a program that converts an integer to a string of its binary representation. The results of Listing 3.41 appear in Output 3.19.

LISTING 3.41: Getting a String Representation of a Binary Display

```
class BinaryConverter
{
  static void Main()
  {
      const int size = 64;
      ulong value;
      char bit;

      System.Console.Write ("Enter an integer: ");
      // Use long.Parse()to support negative numbers
      // Assumes unchecked assignment to ulong.
      value = (ulong)long.Parse(System.Console.ReadLine());

      // Set initial mask to 100...
      ulong mask = 1UL << size - 1;
      for (int count = 0; count < size; count++)
      {
          bit = ((mask & value) != 0) ? '1': '0';
          System.Console.Write(bit);
          // Shift mask one location over to the right
          mask >>= 1;
      }
      System.Console.WriteLine();
  }
}
```

OUTPUT 3.19

```
Enter an integer: 42
00000000000000000000000000000000000000000000000000000000000000000000000000101010
```

Within each iteration of the `for` loop in Listing 3.41 (as discussed later in this chapter), we use the right-shift assignment operator to create a mask corresponding to each bit position in `value`. By using the `&` bit operator to mask a particular bit, we can determine whether the bit is set. If the mask test produces a nonzero result, we write `1` to the console; otherwise, we write `0`. In this way, 23 create output describing the binary value of an unsigned `long`.

Note also that the parentheses in `(mask & value) != 0` are necessary because inequality is higher precedence than the AND operator. Without the explicit parentheses, this expression would be equivalent to `mask & (value != 0)`, which does not make any sense; the left side of the `&` is a `ulong` and the right side is a `bool`.

This particular example is provided for learning purposes only. There is actually a built-in CLR method, `System.Convert.ToString(value, 2)` that does such a conversion. In fact, the second argument specifies the base (for example, 2 for binary, 10 for decimal, or 16 for hexadecimal), allowing for more than just conversion to binary.

Bitwise Compound Assignment Operators (&=, |=, ^=)

Not surprisingly, you can combine these bitwise operators with assignment operators as follows: `&=`, `|=`, and `^=`. As a result, you could take a variable, OR it with a number, and assign the result back to the original variable, which Listing 3.42 demonstrates.

LISTING 3.42: Using Logical Assignment Operators

```
byte and = 12, or = 12, xor = 12;
and &= 7;   // and = 4
or  |= 7;   // or = 15
xor ^= 7;   // xor = 11
System.Console.WriteLine(
    $"and = { and } \nor = { or }\nxor = { xor }");
```

The results of Listing 3.42 appear in Output 3.20.

OUTPUT 3.20

```
and = 4
or = 15
xor = 11
```

Combining a bitmap with a mask using something like `fields &= mask` clears the bits in `fields` that are not set in the `mask`. The opposite, `fields &= ~mask`, clears out the bits in `fields` that are set in `mask`.

Bitwise Complement Operator (~)

The **bitwise complement operator** takes the complement of each bit in the operand, where the operand can be an `int`, `uint`, `long`, or `ulong`. The expression `~1`, therefore, returns the value with binary notation `1111 1111 1111 1111 1111 1111 1111 1110`, and `~(1<<31)` returns the number with binary notation `0111 1111 1111 1111 1111 1111 1111 1111`.

Control Flow Statements, Continued

Now that we've described Boolean expressions in more detail, we can more clearly describe the control flow statements supported by C#. Many of these statements will be familiar to experienced programmers, so you can skim this section looking for details specific to C#. Note in particular the `foreach` loop, as this may be new to many programmers.

The `while` and `do/while` Loops

Thus far you have learned how to write programs that do something only once. However, computers can easily perform similar operations multiple times. To do this, you need to create an instruction loop. The first instruction loop we will discuss is the `while` loop, because it is the simplest conditional loop. The general form of the `while` statement is as follows:

```
while (condition)
 statement
```

The computer will repeatedly execute the statement that is the "body" of the loop as long as the condition (which must be a Boolean expression) evaluates to `true`. If the condition evaluates to `false`, code execution skips

the body and executes the code following the loop statement. Note that `statement` will continue to execute even if it causes the condition to become `false`. The loop exits only when the condition is reevaluated "at the top of the loop." The Fibonacci calculator shown in Listing 3.43 demonstrates the `while` loop.

LISTING 3.43: `while` Loop Example

```
class FibonacciCalculator
{
  static void Main()
  {
      decimal current;
      decimal previous;
      decimal temp;
      decimal input;

      System.Console.Write("Enter a positive integer:");

      // decimal.Parse convert the ReadLine to a decimal.
      input = decimal.Parse(System.Console.ReadLine());

      // Initialize current and previous to 1, the first
      // two numbers in the Fibonacci series.
      current = previous = 1;

      // While the current Fibonacci number in the series is
      // less than the value input by the user.
      while (current <= input)
      {
          temp = current;
          current = previous + current;
          previous = temp; // Executes even if previous
          // statement caused current to exceed input
      }

      System.Console.WriteLine(
        $"The Fibonacci number following this is { current }");
  }
}
```

A **Fibonacci number** is a member of the **Fibonacci series,** which includes all numbers that are the sum of the previous two numbers in the series, beginning with 1 and 1. In Listing 3.43, you prompt the user for an integer. Then you use a `while` loop to find the first Fibonacci number that is greater than the number the user entered.

BEGINNER TOPIC

When to Use a `while` Loop

The remainder of this chapter considers other statements that cause a block of code to execute repeatedly. The term *loop body* refers to the statement (frequently a code block) that is to be executed within the `while` statement, since the code is executed in a "loop" until the exit condition is achieved. It is important to understand which loop construct to select. You use a `while` construct to iterate while the condition evaluates to `true`. A `for` loop is used most appropriately whenever the number of repetitions is known, such as when counting from 0 to *n*. A `do/while` is similar to a `while` loop, except that it will always execute the loop body at least once.

The `do/while` loop is very similar to the `while` loop except that a `do/while` loop is preferred when the number of repetitions is from 1 to *n* and *n* is not known when iterating begins. This pattern frequently occurs when prompting a user for input. Listing 3.44 is taken from the tic-tac-toe program.

LISTING 3.44: `do/while` Loop Example

```
// Repeatedly request player to move until he
// enters a valid position on the board.
bool valid;
do
{
  valid = false;

  // Request a move from the current player.
  System.Console.Write(
      $"\nPlayer {currentPlayer}: Enter move:");
  input = System.Console.ReadLine();

  // Check the current player's input.
  // ...

} while (!valid);
```

In Listing 3.44, you initialize `valid` to `false` at the beginning of each **iteration,** or loop repetition. Next, you prompt and retrieve the number the user input. Although not shown here, you then check whether the input was correct, and if it was, you assign `valid` equal to `true`. Since the code uses a `do/while` statement rather than a `while` statement, the user will be prompted for input at least once.

The general form of the `do/while` loop is as follows:

```
do
  statement
while (condition);
```

As with all the control flow statements, a code block is generally used as the single statement to allow multiple statements to be executed as the loop body. However, any single statement except for a labeled statement or a local variable declaration can be used.

The for Loop

The `for` loop iterates a code block until a specified condition is reached. In that way, it is very similar to the `while` loop. The difference is that the `for` loop has built-in syntax for initializing, incrementing, and testing the value of a counter, known as the **loop variable.** Because there is a specific location in the loop syntax for an increment operation, the increment and decrement operators are frequently used as part of a `for` loop.

Listing 3.45 shows the `for` loop used to display an integer in binary form (functionality the equivalent calling the BCL static function `System.Convert.ToString()` with a `toBase` value of 2). The results of this listing appear in Output 3.21.

LISTING 3.45: Using the for Loop

```
class BinaryConverter
{
  static void Main()
  {
      const int size = 64;
      ulong value;
      char bit;

      System.Console.Write("Enter an integer: ");
      // Use long.Parse() so as to support negative numbers.
      // Assumes unchecked assignment to ulong.
      value = (ulong)long.Parse(System.Console.ReadLine());

      // Set initial mask to 100....
      ulong mask = 1UL << size - 1;
      for (int count = 0; count < size; count++)
      {
          bit = ((mask & value) > 0) ? '1': '0';
          System.Console.Write(bit);
          // Shift mask one location over to the right
          mask >>= 1;
      }
  }
}
```

OUTPUT 3.21

```
Enter an integer: -42
11111111111111111111111111111111111111111111111111111111111111111111111111010110
```

Listing 3.45 performs a bit mask 64 times, once for each bit in the number. The three parts of the `for` loop header first declare and initialize the variable `count`, then describe the condition that must be met for the loop body to be executed, and finally describe the operation that updates the loop variable. The general form of the `for` loop is as follows:

```
for (initial ; condition ; loop)
  statement
```

Here is a breakdown of the `for` loop.

- The `initial` section performs operations that precede the first iteration. In Listing 3.45, it declares and initializes the variable `count`. The `initial` expression does not have to be a declaration of a new variable (though it frequently is). It is possible, for example, to declare the variable beforehand and simply initialize it in the `for` loop, or to skip the initialization section entirely by leaving it blank. Variables declared here are in scope throughout the header and body of the `for` statement.
- The `condition` portion of the `for` loop specifies an end condition. The loop exits when this condition is `false` exactly like the `while` loop does. The `for` loop will execute the body only as long as the condition evaluates to `true`. In the preceding example, the loop exits when `count` is greater than or equal to 64.
- The `loop` expression executes after each iteration. In the preceding example, `count++` executes after the right shift of the mask (`mask >>= 1`), but before the condition is evaluated. During the sixty-fourth iteration, `count` is incremented to 64, causing the condition to become `false`, and therefore terminating the loop.
- The `statement` portion of the `for` loop is the "loop body" code that executes while the conditional expression remains `true`.

If you wrote out each `for` loop execution step in pseudocode without using a `for` loop expression, it would look like this:

1. Declare and initialize `count` to 0.
2. If `count` is less than 64, continue to step 3; otherwise, go to step 7.
3. Calculate `bit` and display it.
4. Shift the mask.
5. Increment `count` by 1.
6. Jump back to line 2.
7. Continue the execution of the program after the loop.

The `for` statement doesn't require any of the elements in its header. The expression `for(;;){ ... }` is perfectly valid; although there still needs to be a means to escape from the loop so that it will not continue to execute indefinitely. (If the condition is missing, it is assumed to be the constant `true`.)

The initial and loop expressions have an unusual syntax to support loops that require multiple loop variables, as shown in Listing 3.46.

LISTING 3.46: for Loop Using Multiple Expressions

```
for (int x = 0, y = 5; ((x <= 5) && (y >=0 )); y--, x++)
{
  System.Console.Write(
      $"{ x }{ ((x > y) ? '>' : '<' )}{ y }\t";
}
```

The results of Listing 3.46 appear in Output 3.22.

OUTPUT 3.22

```
0<5     1<4     2<3     3>2     4>1     5>0
```

Here the initialization clause contains a complex declaration that declares and initializes two loop variables, but this is at least similar to a declaration statement that declares multiple local variables. The loop clause is quite unusual, as it can consist of a comma-separated list of expressions, not just a single expression.

Guidelines

CONSIDER refactoring the method to make the control flow easier to understand if you find yourself writing `for` loops with complex conditionals and multiple loop variables.

The `for` loop is little more than a more convenient way to write a `while` loop; you can always rewrite a `for` loop like this:

```
{
  initial;
  while (condition)
  {
    statement;
    loop;
  }
}
```

Guidelines

DO use the `for` loop when the number of loop iterations is known in advance and the "counter" that gives the number of iterations executed is needed in the loop.

DO use the `while` loop when the number of loop iterations is not known in advance and a counter is not needed.

The foreach Loop

The last loop statement in the C# language is `foreach`. The `foreach` loop iterates through a collection of items, setting a loop variable to represent each item in turn. In the body of the loop, operations may be performed on the item. A nice property of the `foreach` loop is that every item is iterated over exactly once; it is not possible to accidentally miscount and iterate past the end of the collection, as can happen with other loops.

The general form of the `foreach` statement is as follows:

```
foreach(type variable in collection)
  statement
```

Here is a breakdown of the `foreach` statement:

- `type` is used to declare the data type of the variable for each item within the collection. It may be `var`, in which case the compiler infers the type of the item from the type of the collection.
- `variable` is a read-only variable into which the `foreach` loop will automatically assign the next item within the collection. The scope of the variable is limited to the body of the loop.

- `collection` is an expression, such as an array, representing any number of items.
- `statement` is the loop body that executes for each iteration of the loop.

Consider the `foreach` loop in the context of the simple example shown in Listing 3.47.

LISTING 3.47: Determining Remaining Moves Using the `foreach` Loop

```
class TicTacToe      // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      // Hardcode initial board as follows
      // ---+---+---
      //  1 | 2 | 3
      // ---+---+---
      //  4 | 5 | 6
      // ---+---+---
      //  7 | 8 | 9
      // ---+---+---
      char[] cells = {
        '1', '2', '3', '4', '5', '6', '7', '8', '9'
      };

      System.Console.Write(
          "The available moves are as follows: ");

      // Write out the initial available moves
      foreach (char cell in cells)
      {
        if (cell != 'O' && cell != 'X')
        {
            System.Console.Write($"{ cell } ");
        }
      }
  }
}
```

Output 3.23 shows the results of Listing 3.47.

OUTPUT 3.23

```
The available moves are as follows: 1 2 3 4 5 6 7 8 9
```

When the execution engine reaches the `foreach` statement, it assigns to the variable `cell` the first item in the `cells` array—in this case, the value

'1'. It then executes the code within the block that makes up the foreach loop body. The if statement determines whether the value of cell is 'O' or 'X'. If it is neither, the value of cell is written out to the console. The next iteration then assigns the next array value to cell, and so on.

Note that the compiler prevents modification of the variable (cell) during the execution of a foreach loop. Also, the loop variable has a subtly different behavior in C# 5 and higher than it did in previous versions; the difference is apparent only when the loop body contains a lambda expression or anonymous method that uses the loop variable. See Chapter 12 for details.

BEGINNER TOPIC

Where the switch Statement Is More Appropriate

Sometimes you might compare the same value in several continuous if statements, as shown with the input variable in Listing 3.48.

Listing 3.48: Checking the Player's Input with an if Statement

```
// ...

bool valid = false;

// Check the current player's input.
if( (input == "1") ||
  (input == "2") ||
  (input == "3") ||
  (input == "4") ||
  (input == "5") ||
  (input == "6") ||
  (input == "7") ||
  (input == "8") ||
  (input == "9") )
{
    // Save/move as the player directed.
    // ...

    valid = true;
}
else if( (input == "") || (input == "quit") )
{
    valid = true;
}
else
{
    System.Console.WriteLine(
```

```
        "\nERROR:  Enter a value from 1-9. "
        + "Push ENTER to quit");
}

// ...
```

This code validates the text entered to ensure that it is a valid tic-tac-toe move. If the value of `input` were `9`, for example, the program would have to perform nine different evaluations. It would be preferable to jump to the correct code after only one evaluation. To enable this, you use a `switch` statement.

The `switch` Statement

A `switch` statement is simpler to understand than a complex `if` statement when you have a value that must be compared against many different constant values. The `switch` statement looks like this:

```
switch (expression)
{
   case constant:
      statements
    default:
      statements
}
```

Here is a breakdown of the `switch` statement:

- `expression` is the value that is being compared against the different constants. The type of this expression determines the "governing type" of the switch. Allowable governing data types are `bool`, `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, any `enum` type (covered in Chapter 8), the corresponding nullable types of each of those value types, and `string`.
- `constant` is any constant expression compatible with the governing type.
- A group of one or more case labels (or the default label) followed by a group of one or more statements is called a **switch section.** The pattern given previously has two switch sections; Listing 3.49 shows a `switch` statement with three switch sections.

- `statements` is one or more statements to be executed when the expression equals one of the constant values mentioned in a label in the switch section. The end point of the group of statements must not be reachable. Typically the last statement is a jump statement such as a `break`, `return`, or `goto` statement.

Guidelines

DO NOT use `continue` as the jump statement that exits a switch section. This is legal when the `switch` is inside a loop, but it is easy to become confused about the meaning of `break` in a later switch section.

A `switch` statement should have at least one switch section; `switch(x){}` is legal but will generate a warning. Also, the guideline provided earlier (see page 116) was to avoid omitting braces in general. One exception to this rule of thumb is to omit braces for `case` and `break` statements because these keywords serve to indicate the beginning and end of a block, so no braces are needed.

Listing 3.49, with a `switch` statement, is semantically equivalent to the series of `if` statements in Listing 3.48.

Listing 3.49: Replacing the `if` Statement with a `switch` Statement

```
static bool ValidateAndMove(
  int[] playerPositions, int currentPlayer, string input)
{
  bool valid = false;

  // Check the current player's input.
  switch (input)
  {
    case "1" :
    case "2" :
    case "3" :
    case "4" :
    case "5" :
    case "6" :
    case "7" :
    case "8" :
    case "9" :
      // Save/move as the player directed.
      ...
      valid = true;
      break;
```

```
      case "" :
      case "quit" :
        valid = true;
        break;
      default :
        // If none of the other case statements
        // is encountered then the text is invalid.
        System.Console.WriteLine(
          "\nERROR:  Enter a value from 1-9. "
          + "Push ENTER to quit");
        break;
    }

    return valid;
  }
```

In Listing 3.49, `input` is the test expression. Since `input` is a string, the governing type is `string`. If the value of `input` is one of the strings `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`, the move is valid and you change the appropriate cell to match that of the current user's token (X or O). Once execution encounters a `break` statement, control leaves the `switch` statement.

The next switch section describes how to handle the empty string or the string `quit`; it sets `valid` to `true` if `input` equals either value. The `default` switch section is executed if no other switch section had a case label that matched the test expression.

Language Contrast: C++—switch Statement Fall-Through

In C++, if a switch section does not end with a jump statement, control "falls through" to the next switch section, executing its code. Because unintended fall-through is a common error in C++, C# does not allow control to accidentally fall through from one switch section to the next. The C# designers believed it was better to prevent this common source of bugs and encourage better code readability than to match the potentially confusing C++ behavior. If you do want one switch section to execute the statements of another switch section, you may do so explicitly with a `goto` statement, as demonstrated later in this chapter.

There are several things to note about the `switch` statement:

- A `switch` statement with no switch sections will generate a compiler warning, but the statement will still compile.
- Switch sections can appear in any order; the `default` section does not have to appear last. In fact, the `default` switch section does not have to appear at all—it is optional.
- The C# language requires that the end point of every switch section, including the last section, be unreachable. This means that switch sections usually end with a `break`, `return`, `throw`, or `goto`.

Jump Statements

It is possible to alter the execution path of a loop. In fact, with jump statements, it is possible to escape out of the loop or to skip the remaining portion of an iteration and begin with the next iteration, even when the loop condition remains `true`. This section considers some of the ways to jump the execution path from one location to another.

The break Statement

To escape out of a loop or a `switch` statement, C# uses a `break` statement. Whenever the `break` statement is encountered, control immediately leaves the loop or switch. Listing 3.50 examines the `foreach` loop from the tic-tac-toe program.

LISTING 3.50: Using break to Escape Once a Winner Is Found

```
class TicTacToe      // Declares the TicTacToe class.
{
  static void Main() // Declares the entry point of the program.
  {
      int winner = 0;
      // Stores locations each player has moved.
      int[] playerPositions = { 0, 0 };

      // Hardcoded board position.
      //  X | 2 | O
      // ---+---+---
      //  O | O | 6
      // ---+---+---
      //  X | X | X
      playerPositions[0] = 449;
      playerPositions[1] = 28;
```

```
        // Determine if there is a winner.
        int[] winningMasks = {
              7, 56, 448, 73, 146, 292, 84, 273 };

        // Iterate through each winning mask to determine
        // if there is a winner.
        foreach (int mask in winningMasks)
        {
            if ((mask & playerPositions[0]) == mask)
            {
                winner = 1;
                break;
            }
            else if ((mask & playerPositions[1]) == mask)
            {
                winner = 2;
                break;
            }
        }

        System.Console.WriteLine(
            $"Player { winner } was the winner");
    }
}
```

Output 3.24 shows the results of Listing 3.50.

OUTPUT 3.24

```
Player 1 was the winner
```

Listing 3.50 uses a `break` statement when a player holds a winning position. The `break` statement forces its enclosing loop (or a `switch` statement) to cease execution, and control moves to the next line outside the loop. For this listing, if the bit comparison returns `true` (if the board holds a winning position), the `break` statement causes control to jump and display the winner.

BEGINNER TOPIC

Bitwise Operators for Positions

The tic-tac-toe example (the full listing is available in Appendix B) uses the bitwise operators to determine which player wins the game. First, the code

saves the positions of each player into a bitmap called `playerPositions`. (It uses an array so that the positions for both players can be saved.)

To begin, both `playerPositions` are `0`. As each player moves, the bit corresponding to the move is set. If, for example, the player selects cell 3, `shifter` is set to `3 - 1`. The code subtracts 1 because C# is zero based and you need to adjust for 0 as the first position instead of 1. Next, the code sets `position`, the bit corresponding to cell 3, using the shift operator `000000000000001 << shifter`, where `shifter` now has a value of `2`. Lastly, it sets `playerPositions` for the current player (subtracting 1 again to shift to zero based) to `0000000000000100`. Listing 3.51 uses `|=` so that previous moves are combined with the current move.

Listing 3.51: Setting the Bit That Corresponds to Each Player's Move

```
int shifter;  // The number of places to shift
              // over to set a bit.
int position;  // The bit that is to be set.

// int.Parse() converts "input" to an integer.
// "int.Parse(input) - 1" because arrays
// are zero based.
shifter = int.Parse(input) - 1;

// Shift mask of 00000000000000000000000000000001
// over by cellLocations.
position = 1 << shifter;

// Take the current player cells and OR them to set the
// new position as well.
// Since currentPlayer is either 1 or 2,
// subtract 1 to use currentPlayer as an
// index in a zero based array.
playerPositions[currentPlayer-1] |= position;
```

Later in the program, you can iterate over each mask corresponding to winning positions on the board to determine whether the current player has a winning position, as shown in Listing 3.50.

The `continue` Statement

You might have a block containing a series of statements within a loop. If you determine that some conditions warrant executing only a portion of these statements for some iterations, you can use the `continue` statement to jump to the end of the current iteration and begin the next iteration. The

`continue` statement exits the current iteration (regardless of whether additional statements remain) and jumps to the loop condition. At that point, if the loop conditional is still `true`, the loop will continue execution.

Listing 3.52 uses the `continue` statement so that only the letters of the domain portion of an email are displayed. Output 3.25 shows the results of Listing 3.52.

LISTING 3.52: Determining the Domain of an Email Address

```
class EmailDomain
{
  static void Main()
  {
      string email;
      bool insideDomain = false;
      System.Console.WriteLine("Enter an email address: ");

      email = System.Console.ReadLine();

      System.Console.Write("The email domain is: ");

      // Iterate through each letter in the email address.
      foreach (char letter in email)
      {
          if (!insideDomain)
          {
              if (letter == '@')
              {
                  insideDomain = true;
              }
              continue;
          }

          System.Console.Write(letter);
      }
  }
}
```

OUTPUT 3.25

```
Enter an email address:
mark@dotnetprogramming.com
The email domain is: dotnetprogramming.com
```

In Listing 3.52, if you are not yet inside the domain portion of the email address, you can use a `continue` statement to move control to the end of the loop, and process the next character in the email address.

You can almost always use an `if` statement in place of a `continue` statement, and this is usually more readable. The problem with the `continue` statement is that it provides multiple flows of control within a single iteration, which compromises readability. In Listing 3.53, the sample has been rewritten, replacing the `continue` statement with the `if/else` construct to demonstrate a more readable version that does not use the `continue` statement.

Listing 3.53: Replacing a `continue` Statement with an `if` Statement

```
foreach (char letter in email)
{
  if (insideDomain)
  {
      System.Console.Write(letter);
  }
  else
  {
      if (letter == '@')
      {
          insideDomain = true;
      }
  }
}
```

The goto Statement

Early programming languages lacked the relatively sophisticated "structured" control flows that modern languages such as C# have as a matter of course, and instead relied upon simple conditional branching (`if`) and unconditional branching (`goto`) statements for most of their control flow needs. The resultant programs were often hard to understand. The continued existence of a `goto` statement within C# seems like an anachronism to many experienced programmers. However, C# supports `goto`, and it is the only method for supporting fall-through within a `switch` statement. In Listing 3.54, if the `/out` option is set, code execution jumps to the `default` case using the `goto` statement, and similarly for `/f`.

Listing 3.54: Demonstrating a `switch` with `goto` Statements

```
// ...
static void Main(string[] args)
{
  bool isOutputSet = false;
```

```
    bool isFiltered = false;

    foreach (string option in args)
    {
        switch (option)
        {
            case "/out":
                isOutputSet = true;
                isFiltered = false;
                goto default;
            case "/f":
                isFiltered = true;
                isRecursive = false;
                goto default;
            default:
                if (isRecursive)
                {
                    // Recurse down the hierarchy
                    // ...

                }
                else if (isFiltered)
                {
                    // Add option to list of filters.
                    // ...
                }
                break;
        }
    }

    // ...

}
```

Output 3.26 shows how to execute the code shown in Listing 3.54.

Output 3.26

```
C:\SAMPLES>Generate /out fizbottle.bin /f "*.xml" "*.wsdl"
```

To branch to a switch section label other than the default label, you can use the syntax `goto case constant;`, where `constant` is the constant associated with the case label you wish to branch to. To branch to a statement that is not associated with a switch section, precede the target statement with any identifier followed by a colon; you can then use that identifier with the `goto` statement. For example, you could have a labeled statement `myLabel : Console.WriteLine();`. The statement `goto myLabel;` would

then branch to the labeled statement. Fortunately, C# prevents you from using `goto` to branch *into* a code block; instead, `goto` may be used only to branch within a code block or to an enclosing code block. By making these restrictions, C# avoids most of the serious `goto` abuses possible in other languages.

In spite of the improvements, use of `goto` is generally considered to be inelegant, difficult to understand, and symptomatic of poorly structured code. If you need to execute a section of code multiple times or under different circumstances, either use a loop or extract code to a method of its own.

Guidelines

AVOID using `goto`.

C# Preprocessor Directives

Control flow statements evaluate expressions at runtime. In contrast, the C# preprocessor is invoked during compilation. The preprocessor commands are directives to the C# compiler, specifying the sections of code to compile or identifying how to handle specific errors and warnings within the code. C# preprocessor commands can also provide directives to C# editors regarding the organization of code.

Language Contrast: C++—Preprocessing

Languages such as C and C++ use a **preprocessor** to perform actions on the code based on special tokens. Preprocessor directives generally tell the compiler how to compile the code in a file and do not participate in the compilation process itself. In contrast, the C# compiler handles "preprocessor" directives as part of the regular lexical analysis of the source code. As a result, C# does not support preprocessor macros beyond defining a constant. In fact, the term preprocessor is generally a misnomer for C#.

Each preprocessor directive begins with a hash symbol (#), and all preprocessor directives must appear on one line. A newline rather than a semicolon indicates the end of the directive.

A list of each preprocessor directive appears in Table 3.4.

TABLE 3.4: Preprocessor Directives

Statement or Expression	General Syntax Structure	Example

`#if` directive	`#if preprocessor-expression`
	`  code`
	`#endif`
		`#if CSHARP2PLUS`
		`  Console.Clear();`
		`#endif`

`#elif` directive	`#if preprocessor-expression1`
	`  code`
	`#elif preprocessor-expression2`
	`  code`
	`#endif`
		`#if LINUX`
		`...`
		`#elif WINDOWS`
		`...`
		`#endif`

`#else` directive	`#if`
	`  code`
	`#else`
	`  code`
	`#endif`
		`#if CSHARP1`
		`...`
		`#else`
		`...`
		`#endif`

`#define` directive	`#define conditional-symbol`	`#define CSHARP2PLUS`
`#undef` directive	`#undef conditional-symbol`	`#undef CSHARP2PLUS`
`#error` directive	`#error preproc-message`	`#error Buggy implementation`
`#warning` directive	`#warning preproc-message`	`#warning Needs code review`
`#pragma` directive	`#pragma warning`	`#pragma warning disable 1030`

`#line` directive	`#line org-line new-line`
	`...`
	`#line default`
		`#line 467 "TicTacToe.cs"`
		`...`
		`#line default`

`#region` directive	`#region pre-proc-message`
	`  code`
	`#endregion`
		`#region Methods`
		`  ...`
		`#endregion`

Excluding and Including Code (#if, #elif, #else, #endif)

Perhaps the most common use of preprocessor directives is in controlling when and how code is included. For example, to write code that could be compiled by both C# 2.0 and later compilers and the prior version 1.0 compilers, you would use a preprocessor directive to exclude C# 2.0–specific code when compiling with a version 1.0 compiler. You can see this in the tic-tac-toe example and in Listing 3.55.

Listing 3.55: Excluding C# 2.0 Code from a C# 1.x Compiler

```
#if CSHARP2PLUS
System.Console.Clear();
#endif
```

In this case, you call the `System.Console.Clear()` method, which is available only in CLI 2.0 and later versions. Using the `#if` and `#endif` preprocessor directives, this line of code will be compiled only if the preprocessor symbol `CSHARP2PLUS` is defined.

Another use of the preprocessor directive would be to handle differences among platforms, such as surrounding Windows- and Linux-specific APIs with `WINDOWS` and `LINUX` `#if` directives. Developers often use these directives in place of multiline comments (`/*...*/`) because they are easier to remove by defining the appropriate symbol or via a search and replace.

A final common use of the directives is for debugging. If you surround code with an `#if DEBUG`, you will remove the code from a release build on most IDEs. The IDEs define the `DEBUG` symbol by default in a debug compile and `RELEASE` by default for release builds.

To handle an else-if condition, you can use the `#elif` directive within the `#if` directive, instead of creating two entirely separate `#if` blocks, as shown in Listing 3.56.

Listing 3.56: Using #if, #elif, and #endif Directives

```
#if LINUX
...
#elif WINDOWS
...
#endif
```

Defining Preprocessor Symbols (#define, #undef)

You can define a preprocessor symbol in two ways. The first is with the `#define` directive, as shown in Listing 3.57.

LISTING 3.57: A #define Example

```
#define CSHARP2PLUS
```

The second method uses the `define` option when compiling for .NET, as shown in Output 3.27.

OUTPUT 3.27

```
>csc.exe /define:CSHARP2PLUS TicTacToe.cs
```

Output 3.28 shows the same functionality using the Mono compiler.

OUTPUT 3.28

```
>mcs.exe -define:CSHARP2PLUS TicTacToe.cs
```

To add multiple definitions, separate them with a semicolon. The advantage of the `define` compiler option is that no source code changes are required, so you may use the same source files to produce two different binaries.

To undefine a symbol, you use the `#undef` directive in the same way you use `#define`.

Emitting Errors and Warnings (#error, #warning)

Sometimes you may want to flag a potential problem with your code. You do this by inserting `#error` and `#warning` directives to emit an error or a warning, respectively. Listing 3.58 uses the tic-tac-toe sample to warn that the code does not yet prevent players from entering the same move multiple times. The results of Listing 3.58 appear in Output 3.29.

LISTING 3.58: Defining a Warning with #warning

```
#warning    "Same move allowed multiple times."
```

OUTPUT 3.29

```
Performing main compilation...
...\tictactoe.cs(471,16): warning CS1030: #warning: '"Same move
allowed multiple times."'

Build complete -- 0 errors, 1 warnings
```

By including the `#warning` directive, you ensure that the compiler will report a warning, as shown in Output 3.29. This particular warning is a way of flagging the fact that there is a potential enhancement or bug within the code. It could be a simple way of reminding the developer of a pending task.

Begin 2.0

Turning Off Warning Messages (#pragma)

Warnings are helpful because they point to code that could potentially be troublesome. However, sometimes it is preferred to turn off particular warnings explicitly because they can be ignored legitimately. C# 2.0 and later compilers provide the preprocessor `#pragma` directive for just this purpose (see Listing 3.59).

LISTING 3.59: Using the Preprocessor #pragma Directive to Disable the #warning Directive

```
#pragma warning disable  1030
```

Note that warning numbers are prefixed with the letters *CS* in the compiler output. However, this prefix is not used in the `#pragma` warning directive. The number corresponds to the warning error number emitted by the compiler when there is no preprocessor command.

To reenable the warning, `#pragma` supports the `restore` option following the warning, as shown in Listing 3.60.

LISTING 3.60: Using the Preprocessor #pragma Directive to Restore a Warning

```
#pragma warning restore  1030
```

In combination, these two directives can surround a particular block of code where the warning is explicitly determined to be irrelevant.

Perhaps one of the most common warnings to disable is CS1591. This warning appears when you elect to generate XML documentation using the

/doc compiler option, but you neglect to document all of the public items within your program.

nowarn:<warn list> Option

In addition to the #pragma directive, C# compilers generally support the nowarn:<warn list> option. This achieves the same result as #pragma, except that instead of adding it to the source code, you can insert the command as a compiler option. The nowarn option affects the entire compilation, whereas the #pragma option affects only the file in which it appears. Turning off the CS1591 warning, for example, would appear on the command line as shown in Output 3.30.

Output 3.30

```
> csc /doc:generate.xml /nowarn:1591 /out:generate.exe Program.cs
```

End 2.0

Specifying Line Numbers (#line)

The #line directive controls on which line number the C# compiler reports an error or warning. It is used predominantly by utilities and designers that emit C# code. In Listing 3.61, the actual line numbers within the file appear on the left.

Listing 3.61: The #line Preprocessor Directive

```
124         #line 113 "TicTacToe.cs"
125         #warning "Same move allowed multiple times."
126         #line default
```

Including the #line directive causes the compiler to report the warning found on line 125 as though it was on line 113, as shown in the compiler error message in Output 3.31.

Output 3.31

```
Performing main compilation...
...\tictactoe.cs(113,18): warning CS1030: #warning: '"Same move
allowed multiple times."'

Build complete -- 0 errors, 1 warnings
```

Following the `#line` directive with `default` reverses the effect of all prior `#line` directives and instructs the compiler to report true line numbers rather than the ones designated by previous uses of the `#line` directive.

Hints for Visual Editors (#region, #endregion)

C# contains two preprocessor directives, `#region` and `#endregion`, that are useful only within the context of visual code editors. Code editors, such as Microsoft Visual Studio, can search through source code and find these directives to provide editor features when writing code. C# allows you to declare a region of code using the `#region` directive. You must pair the `#region` directive with a matching `#endregion` directive, both of which may optionally include a descriptive string following the directive. In addition, you may nest regions within one another.

Listing 3.62 shows the tic-tac-toe program as an example.

LISTING 3.62: #region and #endregion Preprocessor Directives

```
...
#region Display Tic-tac-toe Board

#if CSHARP2PLUS
  System.Console.Clear();
#endif

// Display the current board;
border = 0;   //  set the first border (border[0] = "|")

// Display the top line of dashes.
// ("\n---+---+---\n")
System.Console.Write(borders[2]);
foreach (char cell in cells)
{
  // Write out a cell value and the border that comes after it.
  System.Console.Write($" { cell } { borders[border] }");

  // Increment to the next border.
  border++;

  // Reset border to 0 if it is 3.
  if (border == 3)
  {
      border = 0;
  }
}
#endregion Display Tic-tac-toe Board
...
```

These preprocessor directives are used, for example, with Microsoft Visual Studio. Visual Studio examines the code and provides a tree control to open and collapse the code (on the left-hand side of the code editor window) that matches the region demarcated by the `#region` directives (see Figure 3.5).

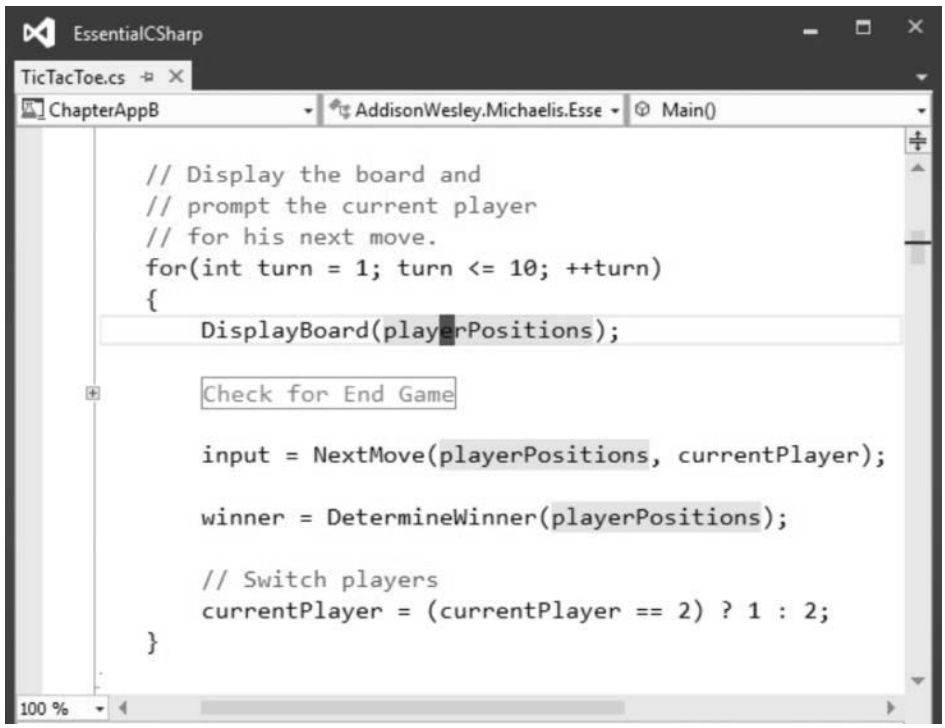


FIGURE 3.5: Collapsed Region in Microsoft Visual Studio .NET

SUMMARY

This chapter began by introducing the C# operators related to assignment and arithmetic. Next, we used the operators along with the `const` keyword to declare constants. Coverage of all the C# operators was not sequential, however. Before discussing the relational and logical comparison operators, the chapter introduced the `if` statement and the important concepts of code blocks and scope. To close out the coverage of operators, we discussed the bitwise operators, especially regarding masks. We also discussed other control flow statements such as loops, `switch`, and `goto`, and ended the chapter with a discussion of the C# preprocessor directives.

Operator precedence was discussed earlier in the chapter; Table 3.5 summarizes the order of precedence across all operators, including several that are not yet covered.

TABLE 3.5: Operator Order of Precedence*

Category	Operators
Primary	`x.y  f(x)  a[x]  x++  x--  new typeof(T)  checked(x)  unchecked(x) default(T) nameof(x)  delegate{}  ()`
Unary	`+  -  !  ~  ++x  --x  (T)x  await x`
Multiplicative	`*  /  %`
Additive	`+  -`
Shift	`<<  >>`
Relational and type testing	`<  >  <=  >=  is  as`
Equality	`==  !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Null coalescing	`??`
Conditional	`?:`
Assignment and lambda	`= *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=  =>`

* Rows appear in order of precedence from highest to lowest.

Perhaps one of the best ways to review all of the content covered in Chapters 1–3 is to look at the tic-tac-toe program found in Appendix B. By reviewing this program, you can see one way in which you can combine all that you have learned into a complete program.

Index

Operators

A

B

C

D

E

G

H

I

J

K

L

M

N

O

P

S

T

U

V

W

X

Y

Z