DEITEL® DEVELOPER SERIES

SIXTH EDITION

C# 6 for Programmers

Use with Windows® 7, 8 or 10

PAUL DEITEL • HARVEY DEITEL

C# 6 for Programmers
Sixth Edition
Deitel® Developer Series

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests.

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

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

Visit us on the Web: `informit.com/ph`

Library of Congress Control Number: 2016946157

ISBN-13: 978-0-13-459632-7
ISBN-10: 0-13-459632-3

Text printed in the United States at RR Donnelley in Crawfordsville, Indiana.
First printing, August 2016

C# 6 FOR PROGRAMMERS
SIXTH EDITION
DEITEL® DEVELOPER SERIES

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

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

Deitel® Series Page

Deitel® Developer Series

Android™ 6 for Programmers: An App-Driven Approach, 3/E
C for Programmers with an Introduction to C11
C++11 for Programmers
C# 6 for Programmers
iOS® 8 for Programmers: An App-Driven Approach with Swift™
Java™ for Programmers, 3/E
JavaScript for Programmers
Swift™ for Programmers

How To Program Series

Android™ How to Program, 3/E
C++ How to Program, 10/E
C How to Program, 8/E
Java™ How to Program, Early Objects Version, 10/E
Java™ How to Program, Late Objects Version, 10/E
Internet & World Wide Web How to Program, 5/E
Visual Basic® 2012 How to Program, 6/E
Visual C#® How to Program, 6/E

Simply Series

Simply Visual Basic® 2010: An App-Driven Approach, 4/E
Simply C++: An App-Driven Tutorial Approach

VitalSource Web Books

`http://bit.ly/DeitelOnVitalSource`
Android™ How to Program, 2/E and 3/E
C++ How to Program, 8/E and 9/E
Java™ How to Program, 9/E and 10/E
Simply C++: An App-Driven Tutorial Approach
Simply Visual Basic® 2010: An App-Driven Approach, 4/E
Visual Basic® 2012 How to Program, 6/E
Visual C#® 2012 How to Program, 5/E
Visual C#® How to Program, 6/E

LiveLessons Video Learning Products

`http://informit.com/deitel`
Android™ 6 App Development Fundamentals, 3/E
C++ Fundamentals
Java™ Fundamentals, 2/E
C# 6 Fundamentals
C# 2012 Fundamentals
iOS® 8 App Development Fundamentals with Swift™, 3/E
JavaScript Fundamentals
Swift™ Fundamentals

REVEL™ Interactive Multimedia

REVEL™ for Deitel Java™

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please join the Deitel communities on

- Facebook®—`http://facebook.com/DeitelFan`
- Twitter®—`http://twitter.com/deitel`
- LinkedIn®—`http://linkedin.com/company/deitel-&-associates`
- YouTube™—`http://youtube.com/DeitelTV`
- Google+™—`http://google.com/+DeitelFan`

and register for the free *Deitel® Buzz Online* e-mail newsletter at:

`http://www.deitel.com/newsletter/subscribe.html`

To communicate with the authors, send e-mail to:

`deitel@deitel.com`

For information on programming-languages corporate training seminars offered by Deitel & Associates, Inc. worldwide, write to `deitel@deitel.com` or visit:

`http://www.deitel.com/training/`

For continuing updates on Pearson/Deitel publications visit:

`http://www.deitel.com`
`http://www.pearsonhighered.com/deitel/`

Visit the Deitel Resource Centers, which will help you master programming languages, software development, Android™ and iOS® app development, and Internet- and web-related topics:

`http://www.deitel.com/ResourceCenters.html`

Trademarks

In memory of William Siebert, Professor Emeritus of Electrical Engineering and Computer Science at MIT:

Your use of visualization techniques in your Signals and Systems lectures inspired the way generations of engineers, computer scientists, educators and authors present their work.

Harvey and Paul Deitel

Contents

Preface

Welcome to the world of leading-edge software development with Microsoft's® Visual C#® programming language. *C# 6 for Programmers, 6/e* is based on C# 6 and related Microsoft software technologies.[1] You'll be using the .NET platform and the Visual Studio® Integrated Development Environment on which you'll conveniently write, test and debug your applications and run them on Windows® devices. The Windows operating system runs on desktop and notebook computers, mobile phones and tablets, game systems and a great variety of devices associated with the emerging "Internet of Things." We believe that this book will give you an informative, engaging, challenging and entertaining introduction to C#.

You'll study C# in the context of four of today's most popular programming paradigms:

- object-oriented programming,
- structured programming,
- generic programming and
- functional programming (new in this edition).

If you haven't already done so, please read the back cover and check out the additional reviewer comments on the inside back cover—these capture the essence of the book concisely. In this Preface we provide more detail.

The book is loaded with "live-code" examples—most new concepts are presented in the context of complete working C# apps, followed by one or more executions showing program inputs and outputs. In the few cases where we show a code snippet, to ensure correctness first we tested it in a working program then copied the code and pasted it into the book. We include a broad range of example apps selected from business, education, computer science, personal utilities, mathematics, simulation, game playing, graphics and many other areas. We also provide abundant tables, line drawings and UML diagrams.

Read the Before You Begin section after this Preface for instructions on setting up your computer to run the 170+ code examples and to enable you to develop your own C# apps. The source code for all of the book's examples is available at

```
http://www.deitel.com/books/CSharp6FP
```

Use the source code we provide to compile and run each program as you study it—this will help you master C# and related Microsoft technologies faster and at a deeper level.

1. At the time of this writing, Microsoft has not yet released the official C# 6 Specification. To view an unofficial copy, visit `https://github.com/ljw1004/csharpspec/blob/gh-pages/README.md`

Contacting the Authors

As you read the book, if you have a question, we're easy to reach at

```
deitel@deitel.com
```

We'll respond promptly.

Join the Deitel & Associates, Inc. Social Media Communities

For book updates, visit

```
http://www.deitel.com/books/CSharp6FP
```

subscribe to the *Deitel®* *Buzz Online* newsletter

```
http://www.deitel.com/newsletter/subscribe.html
```

and join the conversation on

- Facebook®—`http://facebook.com/DeitelFan`
- LinkedIn®—`http://linkedin.com/company/deitel-&-associates`
- YouTube®—`http://youtube.com/DeitelTV`
- Twitter®—`http://twitter.com/Deitel`
- Instagram®—`http://instagram.com/DeitelFan`
- Google+™—`http://google.com/+DeitelFan`

New C# 6 Features

6 We introduce key new C# 6 language features throughout the book (Fig. 1)—each defining occurrence is marked with a "6" margin icon as shown next to this paragraph.

C# 6 new language feature	First introduced in
`string` interpolation	Section 3.5
expression-bodied methods and `get` accessors	Section 7.15
auto-implemented property initializers	Section 8.6.1
getter-only auto-implemented properties	Section 8.6.1
`nameof` operator	Section 10.5.1
null-conditional operator (`?.`)	Section 13.9.1
`when` clause for exception filtering	Section 13.10
`using static` directive	Section 19.3.1
null conditional operator (`?[]`)	Section 19.6
collection initializers for any collection with an `Add` extension method	Section 19.7
index initializers	Section 19.7

Fig. 1 | C# 6 new language features.

A Tour of the Book

Here's a quick walkthrough of the book's key features.

Introduction to Visual C# and Visual Studio 2015 Community Edition

The discussions in

- Chapter 1, Introduction
- Chapter 2, Introduction to Visual Studio and Visual Programming

introduce the C# programming language, Microsoft's .NET platform and Visual Programming. The vast majority of the book's examples will run on Windows 7, 8 and 10 using the *Visual Studio 2015 Community* edition with which we test-drive a **Painter** app in Section 1.7. Chapter 1 briefly reviews object-oriented programming terminology and concepts on which the rest of the book depends.

Introduction to C# Fundamentals

The discussions in

- Chapter 3, Introduction to C# App Programming
- Chapter 4, Introduction to Classes, Objects, Methods and `string`s
- Chapter 5, Control Statements: Part 1
- Chapter 6, Control Statements: Part 2
- Chapter 7, Methods: A Deeper Look
- Chapter 8, Arrays; Introduction to Exception Handling

present rich coverage of C# programming fundamentals (data types, classes, objects, operators, control statements, methods and arrays) through a series of object-oriented programming case studies. Chapter 8 briefly introduces exception handling with an example that demonstrates attempting to access an element outside an array's bounds.

Object-Oriented Programming: A Deeper Look

The discussions in

- Chapter 9, Introduction to LINQ and the `List` Collection
- Chapter 10, Classes and Objects: A Deeper Look
- Chapter 11, Object-Oriented Programming: Inheritance
- Chapter 12, OOP: Polymorphism and Interfaces
- Chapter 13, Exception Handling: A Deeper Look

provide a deeper look at object-oriented programming, including classes, objects, inheritance, polymorphism, interfaces and exception handling. An online two-chapter case study on designing and implementing the object-oriented software for a simple ATM is described later in this preface.

Chapter 9 introduces Microsoft's Language Integrated Query (LINQ) technology, which provides a uniform syntax for manipulating data from various data sources, such as arrays, collections and, as you'll see in later chapters, databases and XML. Chapter 9 is intentionally simple and brief to encourage readers to begin using LINQ technology early.

Section 9.4 introduces the `List` collection. Later in the book, we take a deeper look at LINQ, using LINQ to Entities (for querying databases) and LINQ to XML.

Windows `Forms` Graphical User Interfaces (GUIs)
The discussions in

- Chapter 14, Graphical User Interfaces with Windows Forms: Part 1
- Chapter 15, Graphical User Interfaces with Windows Forms: Part 2

present a detailed introduction to building GUIs using Windows `Forms`. We also use Windows `Forms` GUIs in several later chapters.

Strings and Files
The discussions in

- Chapter 16, Strings and Characters: A Deeper Look
- Chapter 17, Files and Streams

investigate `string`s in more detail, and introduce text-file processing and object-serialization for inputting and outputting entire objects.

Generics and Generic Collections
The discussions in

- Chapter 18, Generics
- Chapter 19, Generic Collections; Functional Programming with LINQ/PLINQ

introduce generics and generic collections. Chapter 18 introduces C# generics and demonstrates how to create type-safe generic methods and a type-safe generic class. Rather than "reinventing the wheel," most C# programmers should use .NET's built-in searching, sorting and generic collections (prepackaged data structures) capabilities, which are discussed in Chapter 19.

Functional Programming with LINQ, PLINQ, Lambdas, Delegates and Immutability
In addition to generic collections, Chapter 19 now introduces functional programming, showing how to use it with LINQ to Objects to write code more concisely and with fewer bugs than programs written using previous techniques. In Section 19.12, with one additional method call, we demonstrate with timing examples how PLINQ (Parallel LINQ) can improve LINQ to Objects performance substantially on multicore systems.

Database with LINQ to Entities and SQL Server
The discussions in

- Chapter 20, Databases and LINQ

introduce database programming with the ADO.NET Entity Framework, LINQ to Entities and Microsoft's free version of SQL Server that's installed with the Visual Studio 2015 Community edition.

Asynchronous Programming
The discussions in

- Chapter 21, Asynchronous Programming with `async` and `await`

show how to take advantage of multicore architectures by writing applications that can process tasks asynchronously, which can improve app performance and GUI responsiveness in apps with long-running or compute-intensive tasks. The `async` modifier and `await` operator greatly simplify asynchronous programming, reduce errors and enable your apps to take advantage of the processing power in today's multicore computers, smartphones and tablets. In this edition, we added a case study that uses the Task Parallel Library (TPL), `async` and `await` in a GUI app—we keep a progress bar moving along in the GUI thread in parallel with a lengthy, compute-intensive calculation in another thread.

Online Bonus Content

Figure 2 shows online bonus content available with the publication of the book.

Online topics
XML and LINQ to XML
Windows Presentation Foundation (WPF) GUI and XAML
Windows Presentation Foundation (WPF) Graphics and Multimedia
ATM Case Study, Part 1: Object-Oriented Design with the UML
ATM Case Study, Part 2: Implementing an OO Design in C#
Appendix: Using the Visual Studio Debugger

Fig. 2 | Online topics on the *C# 6 for Programmers, 6/e* Companion Website.

Accessing the Bonus Content

To access these materials—and for downloads, updates and corrections as they become available—register your copy of *C# 6 for Programmers, 6/e* at `informit.com`. To register:

1. Go to

```
http://informit.com/register
```

2. Log in or create an account.
3. Enter the product ISBN—`9780134596327`—and click **Submit**.

Once you've registered your book, you'll find any available bonus content under **Registered Products**. Here's a quick walkthrough of the initial online content.

XML and LINQ to XML

The Extensible Markup Language (XML), introduced briefly in Chapter 21, is pervasive in the software-development industry, e-business and throughout the .NET platform. XML is required to understand XAML—a Microsoft XML vocabulary that's used to describe graphical user interfaces, graphics and multimedia for Windows Presentation Foundation (WPF) apps, Universal Windows Platform (UWP) apps and Windows 10 Mobile apps. We present XML in more depth, then discuss LINQ to XML, which allows you to query XML content using LINQ syntax.

Windows Presentation Foundation (WPF) GUI, Graphics and Multimedia

Windows Presentation Foundation (WPF)—created after Windows `Forms` and before UWP—is another Microsoft technology for building robust GUI, graphics and multimedia desktop apps. We discuss WPF in the context of a painting app, a text editor, a color chooser, a book-cover viewer, a television video player, various animations, and speech synthesis and recognition apps.

We featured WPF in the previous edition of this book. Our plans now are to move on to UWP for creating apps that can run on desktop, mobile and other Windows devices. For this reason, the WPF chapters are provided as is from the previous edition—we'll no longer evolve this material. Many professionals are still actively using Windows `Forms` and WPF.

Case Study: Using the UML to Develop an Object-Oriented Design and C# Implementation of the Software for an ATM (Automated Teller Machine)

The UML™ (Unified Modeling Language™) is a popular graphical language for visually modeling object-oriented systems. We introduce the UML in the early chapters. We then provide an online object-oriented design case study in which we use the UML to design and implement the software for a simple ATM. We analyze a typical *requirements document* that specifies the details of the system to be built, i.e., *what* the system is supposed to do. We then design the system, specifying *how* it should work—in particular, we

- determine the *classes* needed to implement that system,
- determine the *attributes* the classes need to have,
- determine the *behaviors* the classes' methods need to exhibit and
- specify how the classes must *interact* with one another to meet the system requirements.

From the design, we then produce a complete working C# implementation. Students in our professional courses often report a "light bulb moment"—the case study helps them "tie it all together" and truly understand object orientation.

Future Online Bonus Content

Periodically, we *may* make additional bonus chapters and appendices available at

```
http://www.informit.com/title/9780134596327
```

to registered users of the book. Check this website and/or write to us at `deitel@deitel.com` for the status of this content. These *may* cover:

- Universal Windows Platform (UWP) GUI, graphics and multimedia
- ASP.NET web app development
- Web Services
- Microsoft Azure™ Cloud Computing

Universal Windows Platform (UWP) for Desktop and Mobile Apps

The Universal Windows Platform (UWP) is designed to provide a common platform and user experience across all Windows devices, including personal computers, smartphones, tablets, Xbox and even Microsoft's new HoloLens virtual reality and augmented reality holographic headset—all using nearly identical code.[2]

REST Web Services

Web services enable you to package app functionality in a manner that turns the web into a library of *reusable* services. We used a Flickr REST-based web service in Chapter 21.

Microsoft Azure™ Cloud Computing

Microsoft Azure's web services enable you to develop, manage and distribute your apps in "the cloud."

Notes About the Presentation

C# 6 for Programmers, 6/e contains a rich collection of examples. We concentrate on building well-engineered, high performance software and stress program clarity.

Syntax Shading. For readability, we syntax shade the code, similar to the way Visual Studio colors the code. Our syntax-shading conventions are:

```
comments appear like this
keywords appear like this
constants and literal values appear like this
all other code appears in black
```

Code Highlighting. We emphasize key code segments by placing them in gray rectangles.

Using Fonts for Emphasis. We place the key terms and the index's page reference for each defining occurrence in **bold** text for easy reference. We show on-screen components in the **bold Helvetica** font (for example, the **File** menu) and Visual C# program text in the `Lucida` font (for example, `int count = 5;`). We use *italics* for emphasis.

Objectives. The chapter objectives preview the topics covered in the chapter.

Programming Tips. We include programming tips that focus on important aspects of program development. These tips and practices represent the best we've gleaned from a combined nine decades of programming, professional training and college teaching experience.

Good Programming Practices

The Good Programming Practices *call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

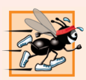
2. As of Summer 2016, Windows Forms, WPF and UWP apps all can be posted for distribution, either free or for sale, via the Windows Store. See `http://bit.ly/DesktopToUWP` for more information.

Common Programming Errors

Pointing out these Common Programming Errors *reduces the likelihood that you'll make them.*

Error-Prevention Tips

These tips contain suggestions for exposing and removing bugs from your programs; many of the tips describe aspects of Visual C# that prevent bugs from getting into programs.



Performance Tips

These tips highlight opportunities for making your programs run faster or minimizing the amount of memory that they occupy.

Software Engineering Observations

The Software Engineering Observations *highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.*

Look-and-Feel Observations

These observations help you design attractive, user-friendly graphical user interfaces that conform to industry norms.

Index. We've included an extensive index for reference. Defining occurrences of key terms in the index are highlighted with a **bold** page number.

Obtaining the Software Used in *C# 6 for Programmers, 6/e*

We wrote the book's code examples in *C# 6 for Programmers, 6/e* using Microsoft's free Visual Studio 2015 Community edition. See the Before You Begin section that follows this preface for download and installation instructions.

Microsoft DreamSpark™

Microsoft provides many of its professional developer tools to students for free via a program called DreamSpark (`http://www.dreamspark.com`). If you're a student using this book in a college course, see the website for details on verifying your status so you take advantage of this program.

Acknowledgments

We'd like to thank Barbara Deitel of Deitel & Associates, Inc. She painstakingly researched the latest versions of Visual C#, Visual Studio, .NET and other key technologies. We'd also like to acknowledge Frank McCown, Ph.D., Associate Professor of Computer Science, Harding University for his suggestion to include an example that used a `ProgressBar` with `async` and `await` in Chapter 21—so we ported to C# a similar example from our book *Java for Programmers, 3/e*.

We're fortunate to have worked with the dedicated team of publishing professionals at Pearson. We appreciate the extraordinary efforts and mentorship of our friend and professional colleague, Mark L. Taub, Editor-in-Chief of the Pearson IT Professional Group.

Kristy Alaura did an extraordinary job recruiting the book's reviewers and managing the review process. Julie Nahil did a wonderful job bringing the book to publication and Chuti Prasertsith worked his magic on the cover design.

Reviewers

The book was scrutinized by industry C# experts and academics teaching C# courses. They provided countless suggestions for improving the presentation. Any remaining flaws in the book are our own.

Sixth Edition Reviewers: Lucian Wischik (Microsoft Visual C# Team), Octavio Hernandez (Microsoft Certified Solutions Developer, Principal Software Engineer at Advanced Bionics), José Antonio González Seco (Parliament of Andalusia, Spain), Bradley Sward (College of Dupage) and Qian Chen (Department of Engineering Technology: Computer Science Technology Program, Savannah State University).

Other recent edition reviewers: Douglas B. Bock (MCSD.NET, Southern Illinois University Edwardsville), Dan Crevier (Microsoft), Shay Friedman (Microsoft Visual C# MVP), Amit K. Ghosh (University of Texas at El Paso), Marcelo Guerra Hahn (Microsoft), Kim Hamilton (Software Design Engineer at Microsoft and co-author of *Learning UML 2.0*), Huanhui Hu (Microsoft Corporation), Stephen Hustedde (South Mountain College), James Edward Keysor (Florida Institute of Technology), Narges Kasiri (Oklahoma State University), Helena Kotas (Microsoft), Charles Liu (University of Texas at San Antonio), Chris Lovett (Software Architect at Microsoft), Bashar Lulu (INETA Country Leader, Arabian Gulf), John McIlhinney (Spatial Intelligence; Microsoft MVP Visual Developer, Visual Basic), Ged Mead (Microsoft Visual Basic MVP, DevCity.net), Anand Mukundan (Architect, Polaris Software Lab Ltd.), Dr. Hamid R. Nemati (The University of North Carolina at Greensboro), Timothy Ng (Microsoft), Akira Onishi (Microsoft), Jeffrey P. Scott (Blackhawk Technical College), Joe Stagner (Senior Program Manager, Developer Tools & Platforms, Microsoft), Erick Thompson (Microsoft), Jesús Ubaldo Quevedo-Torrero (University of Wisconsin–Parkside, Department of Computer Science), Shawn Weisfeld (Microsoft MVP and President and Founder of UserGroup.tv) and Zijiang Yang (Western Michigan University).

As you read the book, we'd sincerely appreciate your comments, criticisms, corrections and suggestions for improvement. Please address all correspondence to:

```
deitel@deitel.com
```

We'll respond promptly. It was fun writing *C# 6 for Programmers, 6/e*—we hope you enjoy reading it!

Paul Deitel
Harvey Deitel

About the Authors

Paul Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., has over 35 years of experience in computing. He is a graduate of MIT, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered hundreds of corporate programming training courses worldwide to clients, including Cisco, IBM, Boeing, Siemens, Sun Microsystems (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, NOAA (National Oceanic and Atmospheric Administration), White Sands Missile Range, Rogue Wave Software, SunGard, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey Deitel, are the world's best-selling programming-language professional book/textbook/video authors.

Paul was named a Microsoft® Most Valuable Professional (MVP) for C# in 2012–2014. According to Microsoft, "the Microsoft MVP Award is an annual award that recognizes exceptional technology community leaders worldwide who actively share their high quality, real-world expertise with users and Microsoft." He also holds the Java Certified Programmer and Java Certified Developer designations and is an Oracle Java Champion.

C# MVP 2012–2014

Dr. Harvey Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has over 55 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University—he studied computing in each of these programs before they spun off Computer Science programs. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul. The Deitels' publications have earned international recognition, with translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, government, military and academic clients.

About Deitel & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in computer programming languages, object technology, Internet and web software technology, and Android and iOS app development. The company's clients include many of the world's largest corporations, government agencies, branches of the military and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including C#®, C++, C, Java™, Android app development, iOS app development, Swift™, Visual Basic® and Internet and web programming.

Through its 40-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., creates leading-edge programming professional books, college textbooks, *LiveLessons* video products, e-books and REVEL™ interactive multimedia courses with integrated labs and assessment (`http://revel.pearson.com`). Deitel & Associates, Inc. and the authors can be reached at:

```
deitel@deitel.com
```

To learn more about Deitel's corporate training curriculum, visit

```
http://www.deitel.com/training
```

To request a proposal for worldwide on-site, instructor-led training at your organization, send an e-mail to `deitel@deitel.com`.

Individuals wishing to purchase Deitel books can do so via

```
http://bit.ly/DeitelOnAmazon
```

Individuals wishing to purchase Deitel *LiveLessons* video training can do so at:

```
http://bit.ly/DeitelOnInformit
```

Deitel books and *LiveLessons* videos are generally available electronically to Safari Books Online subscribers at:

```
http://SafariBooksOnline.com
```

You can get a free 10-day Safari Books Online trial at:

```
https://www.safaribooksonline.com/register/
```

Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

```
http://www.informit.com/store/sales.aspx
```

Before You Begin

Please read this section before using the book to ensure that your computer is set up properly.

Font and Naming Conventions

We use fonts to distinguish between features, such as menu names, menu items, and other elements that appear in the program-development environment. Our convention is

- to emphasize Visual Studio features in a **sans-serif bold font** (e.g., **Properties** window) and
- to emphasize program text in a fixed-width `sans-serif font` (e.g., `bool x = true`).

Visual Studio 2015 Community Edition

This book uses Windows 10 and the free Microsoft Visual Studio 2015 Community edition—Visual Studio also can run on various older Windows versions. Ensure that your system meets Visual Studio 2015 Community edition's minimum hardware and software requirements listed at:

```
https://www.visualstudio.com/en-us/visual-studio-2015-system-
   requirements-vs
```

Next, download the installer from

```
https://www.visualstudio.com/products/visual-studio-express-vs
```

then execute it and follow the on-screen instructions to install Visual Studio.

Though we developed the book's examples on Windows 10, most of the examples will run on Windows 7 and higher. Most examples without graphical user interfaces (GUIs) also will run on other C# and .NET implementations—see "If You're Not Using Microsoft Visual C#…" later in this Before You Begin for more information.

Viewing File Extensions

Several screenshots in *C# 6 for Programmers, 6/e* display file names with file-name extensions (e.g., `.txt`, `.cs`, `.png`, etc.). You may need to adjust your system's settings to display file-name extensions. If you're using Windows 7:

1. Open **Windows Explorer**.
2. Press the *Alt* key to display the menu bar, then select **Folder Options…** from the **Tools** menu.
3. In the dialog that appears, select the **View** tab.

4. In the **Advanced settings** pane, *uncheck* the box to the left of the text **Hide extensions for known file types.**
5. Click **OK** to apply the setting and close the dialog.

If you're using Windows 8 or higher:

1. Open **File Explorer**.
2. Click the **View** tab.
3. Ensure that the **File name extensions** checkbox is *checked*.

Obtaining the Source Code

C# 6 for Programmers, 6/e's source-code examples are available for download at

```
http://www.deitel.com/books/CSharp6FP
```

Click the **Examples** link to download the ZIP archive file to your computer—most browsers will save the file into your user account's `Downloads` folder. You can extract the ZIP file's contents using built-in Windows capabilities, or using a third-party archive-file tool such as WinZip (`www.winzip.com`) or 7-zip (`www.7-zip.org`).

Throughout the book, steps that require you to access our example code on your computer assume that you've extracted the examples from the ZIP file and placed them in your user account's `Documents` folder. You can extract them anywhere you like, but if you choose a different location, you'll need to update our steps accordingly. To extract the ZIP file's contents using the built-in Windows capabilities:

1. Open **Windows Explorer** (Windows 7) or **File Explorer** (Windows 8 and higher).
2. Locate the ZIP file on your system, typically in your user account's `Downloads` folder.
3. Right click the ZIP file and select **Extract All…**.
4. In the dialog that appears, navigate to the folder where you'd like to extract the contents, then click the **Extract** button.

Configuring Visual Studio for Use with This Book

In this section, you'll use Visual Studio's **Options** dialog to configure several Visual Studio options. Setting these options is not required, but will make your Visual Studio match what we show in the book's Visual Studio screen captures.

Visual Studio Theme

Visual Studio has three color themes—**Blue**, **Dark** and **Light**. We used the **Blue** theme with light colored backgrounds to make the book's screen captures easier to read. To switch themes:

1. In the Visual Studio **Tools** menu, select **Options…** to display the **Options** dialog.
2. In the left column, select **Environment**.
3. Select the **Color theme** you wish to use.

Keep the **Options** dialog open for the next step.

Line Numbers

Throughout the book's discussions, we refer to code in our examples by line number. Many programmers find it helpful to display line numbers in Visual Studio as well. To do so:

1. Expand the **Text Editor** node in the **Options** dialog's left pane.
2. Select **All Languages**.
3. In the right pane, check the **Line numbers** checkbox.

Keep the **Options** dialog open for the next step.

Tab Size for Code Indents

Microsoft recommends four-space indents in source code, which is the Visual Studio default. Due to the fixed and limited width of code lines in print, we use three-space indents—this reduces the number of code lines that wrap to a new line, making the code a bit easier to read. If you wish to use three-space indents:

1. Expand the **C#** node in the **Options** dialog's left pane and select **Tabs**.
2. Ensure that **Insert spaces** is selected.
3. Enter **3** for both the **Tab size** and **Indent size** fields.
4. Click **OK** to save your settings.

If You're Not Using Microsoft Visual C#…

C# can be used on other platforms via two open-source projects managed by the .NET Foundation (`http://www.dotnetfoundation.org`)—the Mono Project and .NET Core.

Mono Project

The **Mono Project** is an open source, cross-platform C# and .NET Framework implementation that can be installed on Linux, OS X (soon to be renamed as macOS) and Windows. The code for most of the book's console (non-GUI) apps will compile and run using the Mono Project. Mono also supports Windows `Forms` GUI, which is used in Chapters 14–15 and several later examples. For more information and to download Mono, visit:

```
http://www.mono-project.com/
```

.NET Core

.NET Core is a new cross-platform .NET implementation for Windows, Linux, OS X and FreeBSD. The code for most of the book's console (non-GUI) apps will compile and run using .NET Core. At the time of this writing, a .NET Core version for Windows was available and versions were still under development for other platforms. For more information and to download .NET Core, visit:

```
https://dotnet.github.io/
```

You're now ready to get started with C# and the .NET platform using *C# 6 for Programmers, 6/e*. We hope you enjoy the book!

1

Introduction

Objectives

In this chapter you'll:

- Understand the history of the Visual C# programming language and the Windows operating system.
- Learn what cloud computing with Microsoft Azure is.
- Review the basics of object technology.
- Understand the parts that Windows, .NET, Visual Studio and C# play in the C# ecosystem.
- Test-drive a Visual C# drawing app.

Outline

1.1 Introduction

Welcome to C#[1]—a powerful computer-programming language that's used to build substantial computer applications. There are billions of personal computers in use and an even larger number of mobile devices with computers at their core. Since it was released in 2001, C# has been used primarily to build applications for personal computers and systems that support them. The explosive growth of mobile phones, tablets and other devices also is creating significant opportunities for programming mobile apps. With this new sixth edition of *C# 6 for Programmers*, you'll be able to use Microsoft's new Universal Windows Platform (UWP) with Windows 10 to build C# apps for both personal computers and Windows 10 Mobile devices. With Microsoft's purchase of Xamarin, you also can develop C# mobile apps for Android devices and for iOS devices, such as iPhones and iPads.

1.2 Object Technology: A Brief Review

C# is an object-oriented programming language. In this section we'll review the basics of object technology.

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. **Objects**, or more precisely—as we'll see in Chapter 4—the **classes** objects come from, are essentially *reusable* software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any *noun* can be reasonably represented as a software object in terms of *attributes* (e.g., name, color and size) and *behaviors* (e.g., calculating, moving and communicating). Software developers have discovered that using a modular, object-oriented design-and-implementation approach can make software-development groups much more productive than was possible with earlier techniques—object-oriented programs are often easier to understand, correct and modify.

The Automobile as an Object

Let's begin with a simple analogy. Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal*. What must happen before you can do this? Well, before you

1. The name C#, pronounced "C-sharp," is based on the musical # notation for "sharp" notes.

can drive a car, someone has to *design* it. A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel *hides* the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Before you can drive a car, it must be *built* from the engineering drawings that describe it. A completed car has an *actual* accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must *press* the pedal to accelerate the car.

Methods and Classes

Let's use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a **method**. The method houses the program statements that actually perform the task. It *hides* these statements from its user, just as a car's accelerator pedal hides from the driver the mechanisms of making the car go faster. In C#, we create a program unit called a class to house the set of methods that perform the class's tasks. For example, a class that represents a bank account might contain one method to *deposit* money to an account and another to *withdraw* money from an account. A class is similar in concept to a car's engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

Making Objects from Classes

Just as someone has to *build a car* from its engineering drawings before you can actually drive a car, you must *build an object* from a class before a program can perform the tasks that the class's methods define. The process of doing this is called *instantiation*. An object is then referred to as an **instance** of its class.

Reuse

Just as a car's engineering drawings can be *reused* many times to build many cars, you can *reuse* a class many times to build many objects. Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems, because existing classes and components often have gone through extensive *testing* (to locate problems), *debugging* (to correct those problems) and *performance tuning*. Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that's been spurred by object technology.

Messages and Method Calls

When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster. Similarly, you *send messages to an object*. Each message is implemented as a **method call** that tells a method of the object to perform its task. For example, a program might call a particular bank-account object's *deposit* method to increase the account's balance.

Attributes and Instance Variables

A car, besides having capabilities to accomplish tasks, also has *attributes*, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its *own* attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of *other* cars.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank-account object has a *balance attribute* that represents the amount of money in the account. Each bank-account object knows the balance in the account it represents, but *not* the balances of the *other* accounts in the bank. Attributes are specified by the class's **instance variables**.

Properties, `get` Accessors and `set` Accessors

Attributes are not necessarily accessible directly. The car manufacturer does not want drivers to take apart the car's engine to observe the amount of gas in its tank. Instead, the driver can check the fuel gauge on the dashboard. The bank does not want its customers to walk into the vault to count the amount of money in an account. Instead, the customers talk to a bank teller or check personalized online bank accounts. Similarly, you do not need to have access to an object's instance variables in order to use them. You should use the **properties** of an object. Properties contain **`get` accessors** for reading the values of variables, and **`set` accessors** for storing values into them.

Encapsulation

Classes **encapsulate** (i.e., wrap) attributes and methods into objects created from those classes—an object's attributes and methods are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves. This **information hiding**, as we'll see, is crucial to good software engineering.

Inheritance

A new class of objects can be created quickly and conveniently by **inheritance**—the new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of class "convertible" certainly *is an* object of the more *general* class "automobile," but more *specifically*, the roof can be raised or lowered.

Object-Oriented Analysis and Design (OOAD)

Soon you'll be writing programs in C#. How will you create the code for your programs? Perhaps, like many programmers, you'll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the early chapters of the book), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of thousands of software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed **analysis** process for determining your project's **requirements** (i.e., defining *what* the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Languages like C# are object oriented—programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

The UML (Unified Modeling Language)

Although many different OOAD processes exist, a single graphical language for communicating the results of *any* OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML), is now the most widely used graphical scheme for modeling object-oriented systems. We present our first UML diagrams in Chapters 4 and 5, then use them in our deeper treatment of object-oriented programming through Chapter 12. In our online ATM Software Engineering Case Study, we present a simple subset of the UML's features as we guide you through an object-oriented design and implementation experience.

1.3 C#

In 2000, Microsoft announced the **C#** programming language. C# has roots in the C, C++ and Java programming languages. It has similar capabilities to Java and is appropriate for the most demanding app-development tasks, especially for building today's desktop apps, large-scale enterprise apps, and web-based, mobile and cloud-based apps.

1.3.1 Object-Oriented Programming

C# is *object oriented*—we've discussed the basics of object technology and we present a rich treatment of object-oriented programming throughout the book. C# has access to the powerful **.NET Framework Class Library**—a vast collection of prebuilt classes that enable you to develop apps quickly (Fig. 1.1). We'll say more about .NET in Section 1.4.

Some key capabilities in the .NET Framework Class Library	
Database	Debugging
Building web apps	Multithreading
Graphics	File processing
Input/output	Security
Computer networking	Web communication
Permissions	Graphical user interface
Mobile	Data structures
String processing	Universal Windows Platform GUI

Fig. 1.1 | Some key capabilities in the .NET Framework Class Library.

1.3.2 Event-Driven Programming

C# graphical user interfaces (GUIs) are **event driven**. You can write programs that respond to user-initiated **events** such as mouse clicks, keystrokes, timer expirations and *touches* and *finger swipes*—gestures that are widely used on smartphones and tablets.

1.3.3 Visual Programming

Visual Studio enables you to use C# as a *visual programming language*—in addition to writing program statements to build portions of your apps, you'll also use Visual Studio to drag and drop predefined GUI objects like *buttons* and *textboxes* into place on your screen, and label and resize them. Visual Studio will write much of the GUI code for you.

1.3.4 Generic and Functional Programming

Generic Programming

It's common to write a program that processes a collection—e.g., a collection of numbers, a collection of contacts, a collection of videos, etc. Historically, you had to program separately to handle each type of collection. With generic programming, you write code that handles a collection "in the general" and C# handles the specifics for each collection type, saving you a great deal of work. Chapters 18–19 present generics and generic collections.

Functional Programming

With *functional programming*, you specify *what* you want to accomplish in a task, but *not how* to accomplish it. For example, with Microsoft's LINQ—which we introduce in Chapter 9, then use in many later chapters—you can say, "Here's a collection of numbers, give me the sum of its elements." You do *not* need to specify the mechanics of walking through the elements and adding them into a running total one at a time—LINQ handles all that for you. Functional programming speeds application development and reduces errors. We take a deeper look at functional programming in Chapter 19.

1.3.5 An International Standard

C# has been standardized through ECMA International:

```
http://www.ecma-international.org
```

This enables other implementations of the language besides Microsoft's Visual C#. At the time of this writing, the C# standard document—ECMA-334—was still being updated
6 for C# 6. For information on ECMA-334, visit

```
http://www.ecma-international.org/publications/standards/Ecma-334.htm
```

Visit the Microsoft download center to find the latest version of Microsoft's C# 6 specification, other documentation and software downloads.

1.3.6 C# on Non-Windows Platforms

Microsoft originally developed C# for Windows development, but it can be used on other platforms via the **Mono Project** and **.NET Core**—both managed by the .NET Foundation

```
http://www.dotnetfoundation.org/
```

For more information, see the Before You Begin section after the Preface.

1.3.7 Internet and Web Programming

Today's apps can be written with the aim of communicating among the world's computers. As you'll see, this is the focus of Microsoft's .NET strategy. Later in the book, you'll build web-based apps with C# and Microsoft's **ASP.NET** technology.

1.3.8 Asynchronous Programming with `async` and `await`

In most programming today, each task in a program must finish executing before the next task can begin. This is called *synchronous programming* and is the style we use for most of this book. C# also allows *asynchronous programming* in which multiple tasks can be performed at the *same* time. Asynchronous programming can help you make your apps more responsive to user interactions, such as mouse clicks and keystrokes, among many other uses.

Asynchronous programming in early versions of Visual C# was difficult and error prone. C#'s `async` and `await` capabilities simplify asynchronous programming by enabling the compiler to hide much of the associated complexity from the developer. In Chapter 21, we provide an introduction to asynchronous programming with `async` and `await`.

1.4 Microsoft's .NET

In 2000, Microsoft announced its **.NET initiative** (`www.microsoft.com/net`), a broad vision for using the Internet and the web in the development, engineering, distribution and use of software. Rather than forcing you to use a single programming language, .NET permits you to create apps in *any* .NET-compatible language (such as C#, Visual Basic, Visual C++ and many others). Part of the initiative includes Microsoft's ASP.NET technology for building web-based applications.

1.4.1 .NET Framework

The **.NET Framework Class Library** provides many capabilities that you'll use to build substantial C# apps quickly and easily. It contains *thousands* of valuable *prebuilt* classes that have been tested and tuned to maximize performance. You'll learn how to create your own classes, but you should *re-use* the .NET Framework classes whenever possible to speed up the software-development process, while enhancing the quality and performance of the software you develop.

1.4.2 Common Language Runtime

The **Common Language Runtime (CLR)**, another key part of the .NET Framework, executes .NET programs and provides functionality to make them easier to develop and debug. The CLR is a **virtual machine** (**VM**)—software that manages the execution of programs and hides from them the underlying operating system and hardware. The source code for programs that are executed and managed by the CLR is called *managed code*. The CLR provides various services to managed code, such as

- integrating software components written in different .NET languages,
- error handling between such components,
- enhanced security,
- automatic memory management and more.

Unmanaged-code programs do not have access to the CLR's services, which makes unmanaged code more difficult to write.[2] Managed code is compiled into machine-specific instructions in the following steps:

1. First, the code is compiled into **Microsoft Intermediate Language** (**MSIL**). Code converted into MSIL from other languages and sources can be woven together by the CLR—this allows programmers to work in their preferred .NET programming language. The MSIL for an app's components is placed into the app's *executable file*—the file that causes the computer to perform the app's tasks.
2. When the app executes, another compiler (known as the **just-in-time compiler** or **JIT compiler**) in the CLR translates the MSIL in the executable file into machine-language code (for a particular platform).
3. The machine-language code executes on that platform.

1.4.3 Platform Independence

If the .NET Framework exists and is installed for a platform, that platform can run *any* .NET program. The ability of a program to run without modification across multiple platforms is known as **platform independence**. Code written once can be used on another type of computer without modification, saving time and money. In addition, software can target a wider audience. Previously, companies had to decide whether converting their programs to different platforms—a process called **porting**—was worth the cost. With .NET, porting programs is no longer an issue, at least once .NET itself has been made available on the platforms.

1.4.4 Language Interoperability

The .NET Framework provides a high level of **language interoperability**. Because software components written in different .NET languages (such as C# and Visual Basic) are all compiled into MSIL, the components can be combined to create a single unified program. Thus, MSIL allows the .NET Framework to be **language independent**.

The .NET Framework Class Library can be used by any .NET language. The latest release of .NET includes .NET 4.6 and .NET Core:

- NET 4.6 introduces many improvements and new features, including ASP.NET 5 for web-based applications, improved support for today's high-resolution 4K screens and more.
- .NET Core is the cross-platform subset of .NET for Windows, Linux, OS X and FreeBSD.

1.5 Microsoft's Windows® Operating System

Microsoft's Windows is the most widely personal-computer, desktop operating system worldwide. **Operating systems** are software systems that make using computers more convenient for users, developers and system administrators. They provide *services* that allow each app to execute safely, efficiently and *concurrently* (i.e., in parallel) with other apps.

---

2. `http://msdn.microsoft.com/library/8bs2ecf4`.

Other popular desktop operating systems include macOS (formerly OS X) and Linux. *Mobile operating systems* used in smartphones and tablets include Microsoft's Windows 10 Mobile, Google's Android and Apple's iOS (for iPhone, iPad and iPod Touch devices). Figure 1.2 presents the evolution of the Windows operating system.

Version	Description
Windows in the 1990s	In the mid-1980s, Microsoft developed the **Windows operating system** based on a graphical user interface with buttons, textboxes, menus and other graphical elements. The various versions released throughout the 1990s were intended for personal computing. Microsoft entered the corporate operating systems market with the 1993 release of *Windows NT*.
Windows XP and Windows Vista	*Windows XP* was released in 2001 and combined Microsoft's corporate and consumer operating-system lines. At the time of this writing, it still holds more than 10% of the operating-systems market (`https://www.netmarketshare.com/operating-system-market-share.aspx`). *Windows Vista*, released in 2007, offered the attractive new Aero user interface, many powerful enhancements and new apps and enhanced security. But Vista never caught on.
Windows 7	*Windows 7* is currently the world's most widely used desktop operating system with over 47% of the operating-systems market (`https://www.netmarketshare.com/operating-system-market-share.aspx`). Windows added enhancements to the Aero user interface, faster startup times, further refinement of Vista's security features, touch-screen with multitouch support, and more.
Windows 8 for Desktops and Tablets	Windows 8, released in 2012, provided a similar platform (the underlying system on which apps run) and *user experience* across a wide range of devices including personal computers, smartphones, tablets *and* the Xbox Live online game service. Its new look-and-feel featured a Start screen with *tiles* representing each app, similar to that of *Windows Phone* (now Windows 10 Mobile)—Microsoft's smartphone operating system. Windows 8 featured *multi-touch* support for *touchpads* and *touchscreen* devices, enhanced security features and more.
Windows 8 UI (User Interface)	Windows 8 UI (previously called "Metro") introduced a clean look-and-feel with minimal distractions to the user. Windows 8 apps featured a *chromeless window* with no borders, title bars and menus. These elements were *hidden*, allowing apps to fill the *entire* screen—particularly helpful on smaller screens such as tablets and smartphones. The interface elements were displayed in the *app bar* when the user *swiped* the top or bottom of the screen by holding down the mouse button, moving the mouse in the swipe direction and releasing the mouse button; or using a *finger swipe* on a touchscreen device.

Fig. 1.2 | The evolution of the Windows operating system. (Part 1 of 2.)

Version	Description
Windows 10 and the Universal Windows Platform	Windows 10, released in 2015, is the current version of Windows and currently holds a 15% (and growing) share of the operating-systems market (`https://www.netmarketshare.com/operating-system-market-share.aspx`). In addition to many user-interface and other updates, Windows 10 introduced the **Universal Windows Platform (UWP)**, which is designed to provide a common platform (the underlying system on which apps run) and user experience across all Windows devices including personal computers, smartphones, tablets, Xbox and even Microsoft's new HoloLens augmented reality holographic headset—all using nearly identical code.

Fig. 1.2 | The evolution of the Windows operating system. (Part 2 of 2.)

Windows Store

You can sell apps or offer them for free in the Windows Store. At the time of this writing, the fee to become a registered developer is $19 for individuals and $99 for companies. Microsoft retains 30% of the purchase price (more in some markets). See the App Developer Agreement for more information:

```
https://msdn.microsoft.com/en-us/library/windows/apps/hh694058.aspx
```

The Windows Store offers several business models for monetizing your app. You can charge full price for your app before download, with prices starting at $1.49. You also can offer a time-limited trial or feature-limited trial that allows users to try the app before purchasing the full version, sell virtual goods (such as additional app features) using in-app purchases and more. To learn more about the Windows Store and monetizing your apps, visit

```
https://msdn.microsoft.com/windows/uwp/monetize/index
```

1.6 Visual Studio Integrated Development Environment

C# programs can be created using Microsoft's Visual Studio—a collection of software tools called an **Integrated Development Environment (IDE)**. The **Visual Studio Community** edition IDE enables you to *write*, *run*, *test* and *debug* C# programs quickly and conveniently. It also supports Microsoft's Visual Basic, Visual C++ and F# programming languages and many more. Most of this book's examples were built using *Visual Studio Community*, which runs on Windows 7, 8 and 10. A few of the book's examples require Windows 10.

1.7 Painter Test-Drive in Visual Studio Community

You'll now use *Visual Studio Community* to "test-drive" an existing app that enables you to draw on the screen using the mouse. The **Painter** app allows you to choose among several brush sizes and colors. The elements and functionality you see in this app are typical of what you'll learn to program in this text. The following steps walk you through test-driving the app. For this test drive, we assume that you placed the book's examples in your user account's `Documents` folder in a subfolder named `examples`.

Step 1: Checking Your Setup

Confirm that you've set up your computer and the software properly by reading the book's Before You Begin section that follows the Preface.

Step 2: Locating the Painter *App's Directory*

Open a **File Explorer** (Windows 8 and 10) or **Windows Explorer** (Windows 7) window and navigate to

`C:\Users\`*yourUserName*`\Documents\examples\ch01`

Double click the `Painter` folder to view its contents (Fig. 1.3), then double click the `Painter.sln` file to open the app's solution in Visual Studio. An app's *solution* contains all of the app's *code files*, *supporting files* (such as *images*, *videos*, *data files*, etc.) and configuration information. We'll discuss the contents of a solution in more detail in the next chapter.

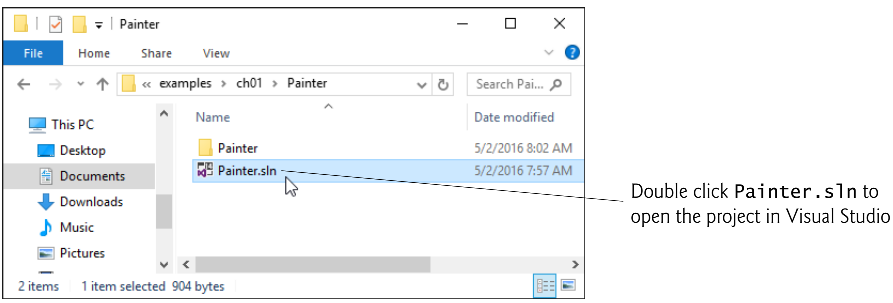


Fig. 1.3 | Contents of `C:\examples\ch01\Painter`.

Depending on your system configuration, **File Explorer** or **Windows Explorer** might display `Painter.sln` simply as `Painter`, without the filename extension `.sln`. To display the filename extensions in Windows 8 and higher:

1. Open **File Explorer**.
2. Click the **View** tab, then ensure that the **File name extensions** checkbox is checked.

To display them in Windows 7:

1. Open **Windows Explorer**.
2. Press *Alt* to display the menu bar, then select **Folder Options...** from **Windows Explorer**'s **Tools** menu.
3. In the dialog that appears, select the **View** tab.
4. In the **Advanced settings:** pane, uncheck the box to the left of the text **Hide extensions for known file types**. [*Note*: If this item is already unchecked, no action needs to be taken.]
5. Click **OK** to apply the setting and close the dialog.

Step 3: Running the Painter App

To see the running **Painter** app, click the **Start** button (Fig. 1.4)

or press the *F5* key.

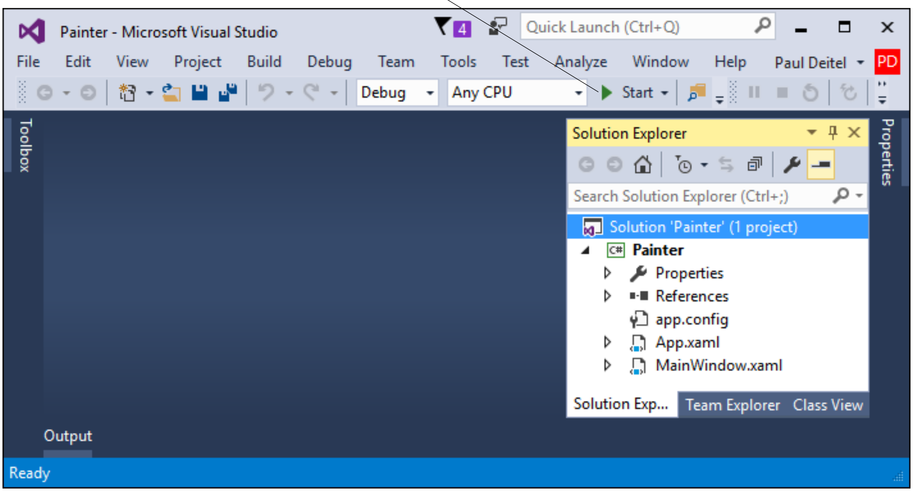


Fig. 1.4 | Running the **Painter** app.

Figure 1.5 shows the running app and labels several of the app's graphical elements—called **controls**. These include `GroupBox`es, `RadioButton`s, `Button`s and a `Panel`. These controls and many others are discussed throughout the text. The app allows you to draw with a **Red**, **Blue**, **Green** or **Black** brush of **Small**, **Medium** or **Large** size. As you drag the mouse on the white `Panel`, the app draws circles of the specified color and size at the mouse pointer's current position. The slower you drag the mouse, the closer the circles will be. Thus, dragging slowly draws a continuous line (as in Fig. 1.6) and dragging quickly draws individual circles with space in between. You also can **Undo** your previous operation or **Clear** the drawing to start from scratch by pressing the `Button`s below the `RadioButton`s in the GUI. By using existing *controls*—which are *objects*—you can create powerful apps much faster than if you had to write all the code yourself. This is a key benefit of *software reuse*.

The brush's properties, selected in the `RadioButton`s labeled **Black** and **Medium**, are *default settings*—the initial settings you see when you first run the app. Programmers include default settings to provide *reasonable* choices that the app will use if the user *does not* change the settings. Default settings also provide visual cues for users to choose their own settings. Now you'll choose your own settings as a user of this app.

Step 4: Changing the Brush Color

Click the `RadioButton` labeled **Red** to change the brush color, then click the `RadioButton` labeled **Small** to change the brush size. Position the mouse over the white `Panel`, then drag the mouse to draw with the brush. Draw flower petals, as shown in Fig. 1.6.

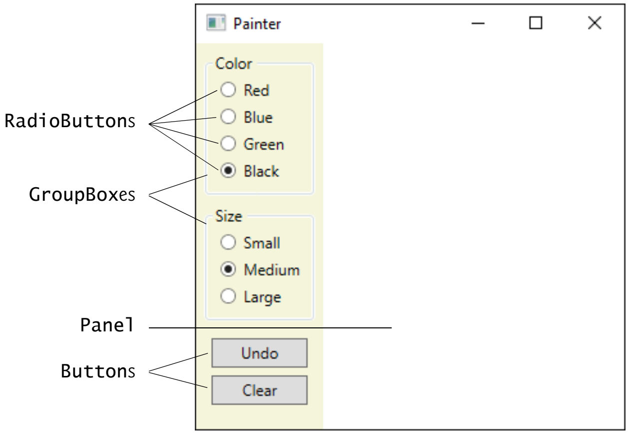


Fig. 1.5 | **Painter** app running in Windows 10.

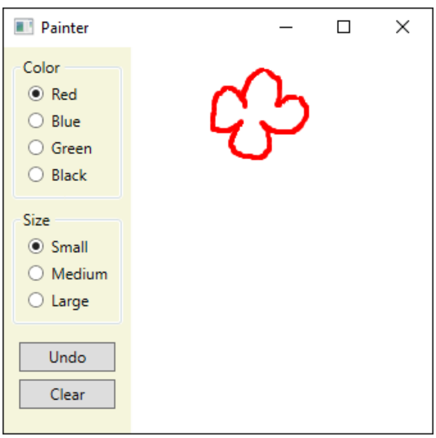


Fig. 1.6 | Drawing flower petals with a small red brush.

Step 5: Changing the Brush Color and Size

Click the **Green** `RadioButton` to change the brush color. Then, click the **Large** `RadioButton` to change the brush size. Draw grass and a flower stem, as shown in Fig. 1.7.

Step 6: Finishing the Drawing

Click the **Blue** and **Medium** `RadioButton`s. Draw raindrops, as shown in Fig. 1.8, to complete the drawing.

Step 7: Stopping the App

When you run an app from Visual Studio, you can terminate it by clicking the stop button

on the Visual Studio toolbar or by clicking the close box

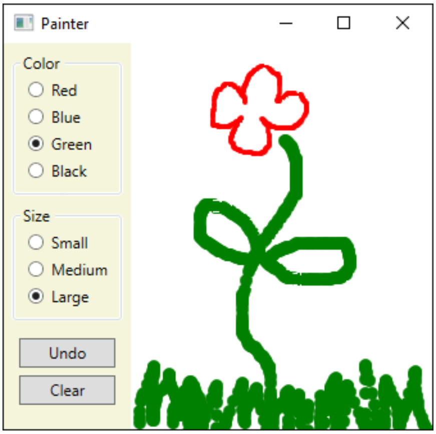


Fig. 1.7 | Drawing the flower stem and grass with a large green brush.

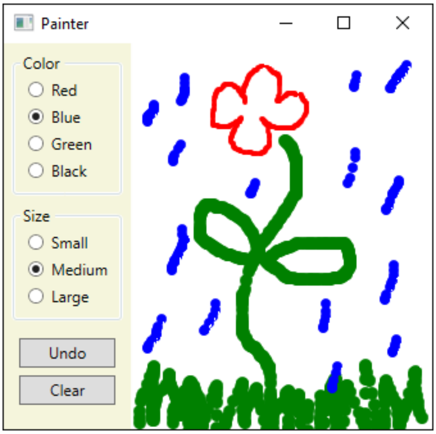


Fig. 1.8 | Drawing rain drops with a medium blue brush.

on the running app's window.

Now that you've completed the test-drive, you're ready to begin developing C# apps. In Chapter 2, Introduction to Visual Studio and Visual Programming, you'll use Visual Studio to create your first C# program using *visual programming* techniques. As you'll see, Visual Studio will generate for you the code that builds the app's GUI. In Chapter 3, Introduction to C# App Programming, you'll begin writing C# programs containing conventional program code that you write.

7

Methods: A Deeper Look

Objectives

In this chapter you'll:

- See that `static` methods and variables are associated with classes rather than objects.
- Use common `Math` class functions.
- Learn C#'s argument promotion rules for when argument types do not match parameter types exactly.
- Get a high-level overview of various namespaces from the .NET Framework Class Library.
- Use random-number generation to implement game-playing apps.
- Understand how the visibility of identifiers is limited to specific regions of programs.
- See how the method call and return mechanism is supported by the method-call stack.
- Create overloaded methods.
- Use optional and named parameters.
- Use recursive methods.
- Understand what value types and reference types are.
- Pass method arguments by value and by reference.

Outline

7.1 Introduction

In this chapter, we take a deeper look at methods. We'll discuss the difference between non-`static` and `static` methods. You'll see that the `Math` class in the .NET Framework Class Library provides many `static` methods to perform mathematical calculations. We'll also discuss `static` variables (known as class variables) and why method `Main` is declared `static`.

You'll declare a method with multiple parameters and use operator + to perform `string` concatenations. We'll discuss C#'s argument promotion rules for implicitly converting simple-type values to other types and when these rules are applied by the compiler. We'll also present several commonly used Framework Class Library namespaces.

We'll take a brief, and hopefully entertaining, diversion into simulation techniques with random-number generation and develop a version of a popular casino dice game that uses most of the programming techniques you've learned so far. You'll declare named constants with the `const` keyword and with `enum` types. We'll then present C#'s scope rules, which determine where identifiers can be referenced in an app.

We'll discuss how the method-call stack enables C# to keep track of which method is currently executing, how local variables of methods are maintained in memory and how a method knows where to return after it completes execution. You'll overload methods in a class by providing methods with the same name but different numbers and/or types of parameters, and learn how to use optional and named parameters.

We'll introduce C# 6's expression-bodied methods, which provide a concise notation for methods that simply return a value to their caller. We'll also use this expression-bodied notation for a read-only property's `get` accessor.

We'll discuss how recursive methods call themselves, breaking larger problems into smaller subproblems until eventually the original problem is solved. Finally, we'll provide more insight into how value-type and reference-type arguments are passed to methods.

7.2 Packaging Code in C#

So far, we've used properties, methods and classes to package code. We'll present additional packaging mechanisms in later chapters. C# apps are written by combining your properties, methods and classes with predefined properties, methods and classes available in the .NET Framework Class Library and in other class libraries. Related classes are often grouped into namespaces and compiled into class libraries so that they can be reused in other apps. You'll learn how to create your own namespaces and class libraries in Chapter 15. The Framework Class Library provides many *predefined* classes that contain methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, graphical user interfaces, graphics, multimedia, printing, file processing, database operations, networking operations, error checking, web-app development, accessibility (for people with disabilities) and more.

Software Engineering Observation 7.1

Don't try to "reinvent the wheel." When possible, reuse Framework Class Library classes and methods (`https://msdn.microsoft.com/library/mt472912`). This reduces app development time and errors, contributes to good performance and often enhances security.

7.3 `static` Methods, `static` Variables and Class `Math`

Although most methods are called to operate on the data of specific objects, this is not always the case. Sometimes a method performs a task that does *not* depend on the data of any object (other than the method's arguments). Such a method applies to the class in which it's declared as a whole and is known as a `static` method.

It's common for a class to contain a group of `static` methods to perform common tasks. For example, recall that we used `static` method `Pow` of class `Math` to raise a value to a power in Fig. 6.6. To declare a method as `static`, place the keyword `static` before the return type in the method's declaration. You call any `static` method by specifying the name of the class in which the method is declared, followed by the member-access operator (`.`) and the method name, as in

```
ClassName.MethodName(arguments)
```

7.3.1 Math Class Methods

Class `Math` (from the `System` namespace) provides a collection of `static` methods that enable you to perform common mathematical calculations. For example, you can calculate the square root of `900.0` with the `static` method call

```
double value = Math.Sqrt(900.0);
```

The expression `Math.Sqrt(900.0)` evaluates to `30.0`. Method `Sqrt` takes an argument of type `double` and returns a result of type `double`. The following statement displays in the console window the value of the preceding method call:

```
Console.WriteLine(Math.Sqrt(900.0));
```

Here, the value that `Sqrt` returns becomes the argument to `WriteLine`. We did not create a `Math` object before calling `Sqrt`, nor did we create a `Console` object before calling `WriteLine`. Also, *all* of `Math`'s methods are `static`—therefore, each is called by preceding the name of the method with the class name `Math` and the member-access operator (`.`).

Method arguments may be constants, variables or expressions. If `c = 13.0`, `d = 3.0` and `f = 4.0`, then the statement

```
Console.WriteLine(Math.Sqrt(c + d * f));
```

calculates and displays the square root of `13.0 + 3.0 * 4.0 = 25.0`—namely, `5.0`. Figure 7.1 summarizes several `Math` class methods. In the figure, *x* and *y* are of type `double`.

Method	Description	Example

`Abs(`*x*`)`	absolute value of *x*	`Abs(23.7)` is `23.7`
		`Abs(0.0)` is `0.0`
		`Abs(-23.7)` is `23.7`

`Ceiling(`*x*`)`	rounds *x* to the smallest integer not less than *x*	`Ceiling(9.2)` is `10.0` `Ceiling(-9.8)` is `-9.0`
`Floor(`*x*`)`	rounds *x* to the largest integer not greater than *x*	`Floor(9.2)` is `9.0` `Floor(-9.8)` is `-10.0`
`Cos(`*x*`)`	trigonometric cosine of *x* (*x* in radians)	`Cos(0.0)` is `1.0`
`Sin(`*x*`)`	trigonometric sine of *x* (*x* in radians)	`Sin(0.0)` is `0.0`
`Tan(`*x*`)`	trigonometric tangent of *x* (*x* in radians)	`Tan(0.0)` is `0.0`

Fig. 7.1 | `Math` class methods. (Part 1 of 2.)

Method	Description	Example
`Exp(x)`	exponential method e^x	`Exp(1.0)` is `2.71828` `Exp(2.0)` is `7.38906`
`Log(x)`	natural logarithm of x (base e)	`Log(Math.E)` is `1.0` `Log(Math.E * Math.E)` is `2.0`
`Max(x, y)`	larger value of x and y	`Max(2.3, 12.7)` is `12.7` `Max(-2.3, -12.7)` is `-2.3`
`Min(x, y)`	smaller value of x and y	`Min(2.3, 12.7)` is `2.3` `Min(-2.3, -12.7)` is `-12.7`
`Pow(x, y)`	x raised to the power y (i.e., x^y)	`Pow(2.0, 7.0)` is `128.0` `Pow(9.0, 0.5)` is `3.0`
`Sqrt(x)`	square root of x	`Sqrt(900.0)` is `30.0`

Fig. 7.1 | `Math` class methods. (Part 2 of 2.)

7.3.2 Math Class Constants PI and E

Each object of a class maintains its own copy of each of the class's *instance variables*. There are also variables for which each object of a class does *not* need its own separate copy (as you'll see momentarily). Such variables are declared `static` and are also known as **class variables**. When objects of a class containing `static` variables are created, all the objects of that class share *one* copy of those variables. Together a class's `static` variables and instance variables are known as its **fields**. You'll learn more about `static` fields in Section 10.9.

Class `Math` also declares two `double` constants for commonly used mathematical values:

- **`Math.PI`** (3.1415926535897931) is the ratio of a circle's circumference to its diameter, and
- **`Math.E`** (2.7182818284590451) is the base value for natural logarithms (calculated with `static Math` method `Log`).

These constants are declared in class `Math` with the modifiers `public` and `const`. Making them `public` allows other programmers to use these variables in their own classes. A constant is declared with the keyword **`const`**—its value cannot be changed after the constant is declared. Fields declared `const` are implicitly `static`, so you can access them via the class name `Math` and the member-access operator (`.`), as in `Math.PI` and `Math.E`.

Common Programming Error 7.1

Constants declared in a class, but not inside a method or property, are implicitly `static`—it's a syntax error to declare such a constant with keyword `static` explicitly.

7.3.3 Why Is Main Declared static?

Why must `Main` be declared `static`? During app *startup*, when *no objects* of the class have been created, the `Main` method must be called to begin program execution. `Main` is sometimes called the app's **entry point**. Declaring `Main` as `static` allows the execution environment to invoke `Main` without creating an instance of the class. Method `Main` is typically declared with the header:

```
static void Main()
```

but also can be declared with the header:

```
static void Main(string[] args)
```

which we'll discuss and demonstrate in Section 8.12, Shuffling and Dealing Cards. In addition, you can declare `Main` with return type `int` (instead of `void`)—this can be useful if an app is executed by another app and needs to return an indication of success or failure to that other app.

7.3.4 Additional Comments About Main

Most earlier examples have one class that contained only `Main`, and some examples had a second class that was used by `Main` to create and manipulate objects. Actually, *any* class can contain a `Main` method. In fact, each of our two-class examples could have been implemented as one class. For example, in the app in Figs. 4.11–4.12, method `Main` (lines 7–43 of Fig. 4.12) could have been moved into class `Account` (Fig. 4.11). The app results would have been identical to those of the two-class version. You can place a `Main` method in every class you declare. Some programmers take advantage of this to build a small test app into each class they declare. However, if you declare more than one `Main` method among the classes of your project, you'll need to indicate to the IDE which one you would like to be the *app's* entry point. To do so:

1. With the project open in Visual Studio, select **Project > [ProjectName] Properties...** (where **[ProjectName]** is the name of your project).
2. Select the class containing the `Main` method that should be the entry point from the **Startup object** list box.

7.4 Methods with Multiple Parameters

We now consider how to write a method with multiple parameters. Figure 7.2 defines `Maximum` method that determines and returns the largest of *three* `double` values. When the app begins execution, the `Main` method (lines 8–23) executes. Line 19 calls method `Maximum` (declared in lines 26–43) to determine and return the largest of its three `double` arguments. In Section 7.4.3, we'll discuss the use of the + operator in line 22. The sample outputs show that `Maximum` determines the largest value regardless of whether that value is the first, second or third argument.

```
1  // Fig. 7.2: MaximumFinder.cs
2  // Method Maximum with three parameters.
3  using System;
4
5  class MaximumFinder
6  {
7     // obtain three floating-point values and determine maximum value
8     static void Main()
9     {
```

Fig. 7.2 | Method `Maximum` with three parameters. (Part 1 of 2.)

```
         // prompt for and input three floating-point values
         Console.Write("Enter first floating-point value: ");
         double number1 = double.Parse(Console.ReadLine());
         Console.Write("Enter second floating-point value: ");
         double number2 = double.Parse(Console.ReadLine());
         Console.Write("Enter third floating-point value: ");
         double number3 = double.Parse(Console.ReadLine());

         // determine the maximum of three values
         double result = Maximum(number1, number2, number3);

         // display maximum value
         Console.WriteLine("Maximum is: " + result);
      }

      // returns the maximum of its three double parameters
      static double Maximum(double x, double y, double z)
      {
         double maximumValue = x; // assume x is the largest to start

         // determine whether y is greater than maximumValue
         if (y > maximumValue)
         {
            maximumValue = y;
         }

         // determine whether z is greater than maximumValue
         if (z > maximumValue)
         {
            maximumValue = z;
         }

         return maximumValue;
      }
   }
```

```
Enter first floating-point values: 3.33
Enter second floating-point values: 1.11
Enter third floating-point values: 2.22
Maximum is: 3.33
```

```
Enter first floating-point values: 2.22
Enter second floating-point values: 3.33
Enter third floating-point values: 1.11
Maximum is: 3.33
```

```
Enter first floating-point values: 2.22
Enter second floating-point values: 1.11
Enter third floating-point values: 3.33
Maximum is: 3.33
```

Fig. 7.2 | Method `Maximum` with three parameters. (Part 2 of 2.)

7.4.1 Keyword `static`

Method `Maximum`'s declaration begins with keyword `static`, which enables the `Main` method (another `static` method) to call `Maximum` as shown in line 19 without creating an object of class `MaximumFinder` and without qualifying the method name with the class name `MaximumFinder`—`static` methods in the *same* class can call each other directly.

7.4.2 Method `Maximum`

Consider the declaration of method `Maximum` (lines 26–43). Line 26 indicates that the method returns a `double` value, that the method's name is `Maximum` and that the method requires *three* `double` parameters (`x`, `y` and `z`) to accomplish its task. When a method has more than one parameter, the parameters are specified as a *comma-separated list*. When `Maximum` is called in line 19, the parameter `x` is initialized with the value of the argument `number1`, the parameter `y` is initialized with the value of the argument `number2` and the parameter `z` is initialized with the value of the argument `number3`. There must be one argument in the method call for each required parameter in the method declaration. Also, each argument must be *consistent* with the type of the corresponding parameter. For example, a parameter of type `double` can receive values like `7.35` (a `double`), `22` (an `int`) or `-0.03456` (a `double`), but not `string`s like `"hello"`. Section 7.6 discusses the argument types that can be provided in a method call for each parameter of a simple type. Note the use of type `double`'s **`Parse`** method in lines 12, 14 and 16 to convert into `double` values the `string`s typed by the user.

> **Common Programming Error 7.2**
>
> *Declaring method parameters of the same type as* `double x, y` *instead of* `double x, double y` *is a syntax error—a type is required for each parameter in the parameter list.*

Logic of Determining the Maximum Value

To determine the maximum value, we begin with the assumption that parameter `x` contains the largest value, so line 28 declares local variable `maximumValue` and initializes it with the value of parameter `x`. Of course, it's possible that parameter `y` or `z` contains the largest value, so we must compare each of these values with `maximumValue`. The `if` statement at lines 31–34 determines whether `y` is greater than `maximumValue`. If so, line 33 assigns `y` to `maximumValue`. The `if` statement at lines 37–40 determines whether `z` is greater than `maximumValue`. If so, line 39 assigns `z` to `maximumValue`. At this point, the largest of the three values resides in `maximumValue`, so line 42 returns that value to line 19 where it's assigned to the variable `result`. When program control returns to the point in the app where `Maximum` was called, `Maximum`'s parameters `x`, `y` and `z` are no longer accessible. Methods can return *at most one* value; the returned value can be a value type that contains one or more values (implemented as a `struct`; Section 10.13) or a reference to an object that contains one or more values.

7.4.3 Assembling `string`s with Concatenation

C# allows `string` objects to be created by assembling smaller `string`s into larger `string`s using operator `+` (or the compound assignment operator `+=`). This is known as **`string` concatenation**. When both operands of operator `+` are `string` objects, the `+` operator creates a *new* `string` object containing copies of the characters in its left operand followed by copies of the characters in its right operand. For example, the expression `"hello " + "there"` creates the `string` `"hello there"` without disturbing the original `string`s.

In line 22, the expression `"Maximum is: " + result` uses operator `+` with operands of types `string` and `double`. Every simple-type value has a `string` representation. When one of the `+` operator's operands is a `string`, the other is implicitly converted to a `string`, then the two `string`s are *concatenated*. So, in line 22, the `double` value is converted to its `string` representation and placed at the end of `"Maximum is: "`. If there are any trailing zeros in a `double` value, these are *discarded*. Thus, the `string` representation of 9.3500 is `"9.35"`.

Anything Can Be Converted to a `string`

If a `bool` is concatenated with a `string`, the `bool` is converted to the `string` `"True"` or `"False"` (each is capitalized). In addition, every object has a `ToString` method that returns a `string` representation of that object. When an object is concatenated with a `string`, the object's `ToString` method is called *implicitly* to obtain the `string` representation of the object. If the object is `null`, an *empty string* is written.

If a type does not define a `ToString` method, the default `ToString` implementation returns a `string` containing the type's **fully qualified name**—that is, the namespace in which the type is defined followed by a dot (`.`) and the type name (e.g., `System.Object` for the .NET class `Object`). Each type you create can declare a custom `ToString` method, as you'll do in Chapter 8 for a `Card` class that represents a playing card in a deck of cards.

Formatting `strings` *with* `string` *Interpolation*

Line 22 of Fig. 7.2, of course, could also be written using `string` interpolation as

```
Console.WriteLine($"Maximum is: {result}");
```

As with `string` concatenation, using `string` interpolation to insert an *object* into a `string` *implicitly* calls the object's `ToString` method to obtain the object's `string` representation.

7.4.4 Breaking Apart Large `string` Literals

When a large `string` literal or interpolated `string` is typed into an app's source code, you can break that `string` into several smaller `string`s and place them on multiple lines for readability. The `string`s can be reassembled using `string` concatenation. We discuss the details of `string`s in Chapter 16.

Common Programming Error 7.3

It's a syntax error to break a `string` *literal or interpolated* `string` *across multiple lines of code. If a* `string` *does not fit on one line, you can split it into several smaller* `string`*s and use concatenation to form the desired* `string`*.*

Common Programming Error 7.4

Confusing the string concatenation `+` *operator with the addition* `+` *operator can lead to strange results. The* `+` *operator is left-associative. For example, if* `y` *has the* `int` *value* `5`*, the expression* `"y + 2 = " + y + 2` *results in the* `string` `"y + 2 = 52"`*, not* `"y + 2 = 7"`*, because first the value of* `y` *(*`5`*) is concatenated with the* `string` `"y + 2 = "`*, then the value* `2` *is concatenated with the new larger* `string` `"y + 2 = 5"`*. The expression* `"y + 2 = " + (y + 2)` *produces the desired result* `"y + 2 = 7"`*. Using C# 6* `string` *interpolation eliminates this problem.*

7.4.5 When to Declare Variables as Fields

Variable `result` is a local variable in method `Main` because it's declared in the block that represents the method's body. Variables should be declared as fields of a class (i.e., as either instance variables or `static` variables) *only* if they're required for use in more than one method of the class or if the app should save their values between calls to a given method.

7.4.6 Implementing Method `Maximum` by Reusing Method `Math.Max`

Recall from Fig. 7.1 that class `Math`'s `Max` method can determine the larger of two values. The entire body of our maximum method could also be implemented with nested calls to `Math.Max`, as follows:

```
return Math.Max(x, Math.Max(y, z));
```

The leftmost `Math.Max` call has the arguments `x` and `Math.Max(y, z)`. Before any method can be called, the runtime evaluates *all* the arguments to determine their values. If an argument is a method call, the call must be performed to determine its return value. So, in the preceding statement, `Math.Max(y, z)` is evaluated first to determine the larger of `y` and `z`. Then the result is passed as the second argument to the first call to `Math.Max`, which returns the larger of its two arguments. Using `Math.Max` in this manner is a good example of software reuse—we find the largest of three values by reusing `Math.Max`, which finds the larger of two values. Note how concise this code is compared to lines 28–42 of Fig. 7.2.

7.5 Notes on Using Methods

Three Ways to Call a Method

You've seen three ways to call a method:

1. Using a method name by itself to call a method of the *same* class—as in line 19 of Fig. 7.2, which calls `Maximum(number1, number2, number3)` from `Main`.
2. Using a reference to an object, followed by the member-access operator (`.`) and the method name to call a non-`static` method of the referenced object—as in line 23 of Fig. 4.12, which called `account1.Deposit(depositAmount)` from the `Main` method of class `AccountTest`.
3. Using the class name and the member-access operator (`.`) to call a `static` method of a class—as in lines 12, 14 and 16 of Fig. 7.2, which each call `Console.ReadLine()`, or as in `Math.Sqrt(900.0)` in Section 7.3.

Three Ways to Return from a Method

You've seen three ways to return control to the statement that calls a method:

- Reaching the method-ending right brace in a method with return type `void`.
- When the following statement executes in a method with return type `void`

```
return;
```

- When a method returns a result with a statement of the following form in which the *expression* is evaluated and its result (and control) are returned to the caller:

```
return expression;
```

Common Programming Error 7.5

Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.

Common Programming Error 7.6

Redeclaring a method parameter as a local variable in the method's body is a compilation error.

Common Programming Error 7.7

Forgetting to return a value from a method that should return one is a compilation error. If a return type other than `void` is specified, the method must use a `return` statement to return a value, and that value must be consistent with the method's return type. Returning a value from a method whose return type has been declared `void` is a compilation error.

`static` *Members Can Access Only the Class's Other* `static` *Members Directly*

A `static` method or property can call *only* other `static` methods or properties of the same class directly (i.e., using the method name by itself) and can manipulate *only* `static` variables in the same class directly. To access a class's non-`static` members, a `static` method or property must use a reference to an object of that class. Recall that `static` methods relate to a class as a whole, whereas non-`static` methods are associated with a specific object (instance) of the class and may manipulate the instance variables of that object (as well as the class's `static` members).

Many objects of a class, each with its own copies of the instance variables, may exist at the same time. Suppose a `static` method were to invoke a non-`static` method directly. How would the method know which object's instance variables to manipulate? What would happen if no objects of the class existed at the time the non-`static` method was invoked?

Software Engineering Observation 7.2

A `static` method cannot access non-`static` members of the same class directly.

7.6 Argument Promotion and Casting

Another important feature of method calls is **argument promotion**—implicitly converting an argument's value to the type that the method expects to receive (if possible) in its corresponding parameter. For example, an app can call `Math` method `Sqrt` with an integer argument even though the method expects to receive a `double` argument. The statement

```
Console.WriteLine(Math.Sqrt(4));
```

correctly evaluates `Math.Sqrt(4)` and displays the value `2.0`. `Sqrt`'s parameter list causes C# to convert the `int` value `4` to the `double` value `4.0` before passing the value to `Sqrt`. Such conversions may lead to compilation errors if C#'s **promotion rules** are not satisfied. The promotion rules specify which conversions are allowed—that is, which conversions can be performed *without losing data*. In the `Sqrt` example above, an `int` is converted to a double without changing its value. However, converting a double to an int *truncates* the fractional part of the double value—thus, part of the value is lost. Also, double variables can hold values much larger (and much smaller) than int variables, so assigning a double to an int can cause a loss of information when the double value doesn't fit in the int. Converting large integer types to small integer types (e.g., long to int) also can produce incorrect results.

7.6.1 Promotion Rules

The promotion rules apply to expressions containing values of two or more simple types and to simple-type values passed as arguments to methods. Each value is promoted to the appropriate type in the expression. (Actually, the expression uses a *temporary* copy of each promoted value—the types of the original values remain unchanged.) Figure 7.3 lists the simple types alphabetically and the types to which each can be promoted. Values of all simple types also can be implicitly converted to type object. We demonstrate such implicit conversions in Chapter 19.

Type	Conversion types
bool	no possible implicit conversions to other simple types
byte	ushort, short, uint, int, ulong, long, decimal, float or double
char	ushort, int, uint, long, ulong, decimal, float or double
decimal	no possible implicit conversions to other simple types
double	no possible implicit conversions to other simple types
float	double
int	long, decimal, float or double
long	decimal, float or double
sbyte	short, int, long, decimal, float or double
short	int, long, decimal, float or double
uint	ulong, long, decimal, float or double
ulong	decimal, float or double
ushort	uint, int, ulong, long, decimal, float or double

Fig. 7.3 | Implicit conversions between simple types.

7.6.2 Sometimes Explicit Casts Are Required

By default, C# does not allow you to implicitly convert values between simple types if the target type cannot represent every value of the original type (e.g., the int value 2000000 cannot be represented as a short, and any floating-point number with nonzero digits after its decimal point cannot be represented in an integer type such as long, int or short).

To prevent a compilation error in cases where information may be lost due to an implicit conversion between simple types, the compiler requires you to use a *cast operator* to *force* the conversion. This enables you to "take control" from the compiler. You essentially say, "I know this conversion might cause loss of information, but for my purposes here, that's fine." Suppose you create a method `Square` that calculates the square of an `int` argument. To call `Square` with the whole part of a `double` argument named `doubleValue`, you'd write `Square((int) doubleValue)`. This method call explicitly casts (converts) the value of `doubleValue` to an integer for use in method `Square`. Thus, if `doubleValue`'s value is `4.5`, the method receives the value `4` and returns `16`, not `20.25`.

Common Programming Error 7.8

Converting a simple-type value to a value of another simple type may change the value if the promotion is not allowed. For example, converting a floating-point value to an integral value may introduce truncation errors (loss of the fractional part) in the result.

7.7 The .NET Framework Class Library

Many predefined classes are grouped into categories of related classes called *namespaces*. Together, these namespaces are referred to as the .NET Framework Class Library.

`using` *Directives and Namespaces*

Throughout the text, `using` directives allow us to use library classes from the Framework Class Library without specifying their namespace names. For example, an app would include the declaration

```
using System;
```

in order to use the class names from the `System` namespace without fully qualifying their names. This allows you to use the *unqualified* name `Console`, rather than the *fully qualified* name `System.Console`, in your code.

Software Engineering Observation 7.3

The C# compiler does not require `using` *declarations in a source-code file if the fully qualified class name is specified every time a class name is used. Many programmers prefer the more concise programming style enabled by* `using` *declarations.*

You might have noticed in each project containing multiple classes that in each class's source-code file we did not need additional `using` directives to use the other classes in the project. There's a special relationship between classes in a project—by default, such classes are in the same namespace and can be used by other classes in the project. Thus, a `using` declaration is not required when one class in a project uses another in the same project—such as when class `AccountTest` used class `Account` in Chapter 4's examples. Also, any classes that are not *explicitly* placed in a namespace are *implicitly* placed in the so-called **global namespace**.

.NET Namespaces

A strength of C# is the large number of classes in the namespaces of the .NET Framework Class Library. Some key Framework Class Library namespaces are described in Fig. 7.4, which represents only a small portion of the reusable classes in the .NET Framework Class Library.

Namespace	Description
`System.Windows.Forms`	Contains the classes required to create and manipulate GUIs. (Various classes in this namespace are discussed in Chapter 14, Graphical User Interfaces with Windows Forms: Part 1, and Chapter 15, Graphical User Interfaces with Windows Forms: Part 2.)

`System.Windows.Controls`
`System.Windows.Input`
`System.Windows.Media`
`System.Windows.Shapes`
	Contain the classes of the Windows Presentation Foundation for GUIs, 2-D and 3-D graphics, multimedia and animation.

`System.Linq`	Contains the classes that support Language Integrated Query (LINQ). (See Chapter 9, Introduction to LINQ and the `List` Collection, and several other chapters throughout the book.)
`System.Data.Entity`	Contains the classes for manipulating data in databases (i.e., organized collections of data), including support for LINQ to Entities. (See Chapter 20, Databases and LINQ.)
`System.IO`	Contains the classes that enable programs to input and output data. (See Chapter 17, Files and Streams.)
`System.Web`	Contains the classes used for creating and maintaining web apps, which are accessible over the Internet.
`System.Xml`	Contains the classes for creating and manipulating XML data. Data can be read from or written to XML files.
`System.Xml.Linq`	Contains the classes that support Language Integrated Query (LINQ) for XML documents. (See Chapter 21, Asynchronous Programming with `async` and `await`.)
`System.Collections` `System.Collections.Generic`	Contain the classes that define data structures for maintaining collections of data. (See Chapter 19, Generic Collections; Functional Programming with LINQ/PLINQ.)
`System.Text`	Contains classes that enable programs to manipulate characters and `string`s. (See Chapter 16, Strings and Characters: A Deeper Look.)

Fig. 7.4 | .NET Framework Class Library namespaces (a subset).

Locating Additional Information About a .NET Class's Methods

You can locate additional information about a .NET class's methods in the *.NET Framework Class Library* reference

```
https://msdn.microsoft.com/library/mt472912
```

When you visit this site, you'll see an alphabetical listing of all the namespaces in the Framework Class Library. Locate the namespace and click its link to see an alphabetical listing of all its classes, with a brief description of each. Click a class's link to see a more complete description of the class. Click the **Methods** link in the left-hand column to see a listing of the class's methods.

Good Programming Practice 7.1

The online .NET Framework documentation is easy to search and provides many details about each class. As you learn each class in this book, you should review it in the online documentation for additional information.

7.8 Case Study: Random-Number Generation

In this and the next section, we develop a nicely structured game-playing app with multiple methods. The app uses most of the control statements presented thus far in the book and introduces several new programming concepts.

There's something in the air of a casino that invigorates people—from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It's the **element of chance**, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced in an app via an object of class `Random` (of namespace `System`). Objects of class **`Random`** can produce random `byte`, `int` and `double` values. In the next several examples, we use objects of class `Random` to produce random numbers.

Secure Random Numbers

According to Microsoft's documentation for class `Random`, the random values it produces "are not completely random because a mathematical algorithm is used to select them, but they are sufficiently random for practical purposes." Such values should not be used, for example, to create randomly selected passwords. If your app requires so-called cryptographically secure random numbers, use class `RNGCryptoServiceProvider`[1] from namespace `System.Security.Cryptography`) to produce random values:

```
https://msdn.microsoft.com/library/system.security.cryptography.
   rngcryptoserviceprovider
```

7.8.1 Creating an Object of Type Random

A new random-number generator object can be created with class `Random` (from the `System` namespace) as follows:

```
Random randomNumbers = new Random();
```

The `Random` object can then be used to generate random `byte`, `int` and `double` values—we discuss only random `int` values here.

7.8.2 Generating a Random Integer

Consider the following statement:

```
int randomValue = randomNumbers.Next();
```

When called with no arguments, method **`Next`** of class `Random` generates a random `int` value in the range 0 to +2,147,483,646, inclusive. If the `Next` method truly produces values at random, then every value in that range should have an equal chance (or probability) of being chosen each time method `Next` is called. The values returned by `Next` are actually

1. Class `RNGCryptoServiceProvider` produces arrays of `byte`s. We discuss arrays in Chapter 8.

pseudorandom numbers—a sequence of values produced by a complex mathematical calculation. The calculation uses the current time of day (which, of course, changes constantly) to **seed** the random-number generator such that each execution of an app yields a different sequence of random values.

7.8.3 Scaling the Random-Number Range

The range of values produced directly by method `Next` often differs from the range of values required in a particular C# app. For example, an app that simulates coin tossing might require only 0 for "heads" and 1 for "tails." An app that simulates the rolling of a six-sided die might require random integers in the range 1–6. A video game that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon might require random integers in the range 1–4. For cases like these, class `Random` provides versions of method `Next` that accept arguments. One receives an `int` argument and returns a value from 0 up to, but not including, the argument's value. For example, you might use the statement

```
int randomValue = randomNumbers.Next(6); // 0, 1, 2, 3, 4 or 5
```

which returns 0, 1, 2, 3, 4 or 5. The argument `6`—called the **scaling factor**—represents the number of unique values that `Next` should produce (in this case, six—0, 1, 2, 3, 4 and 5). This manipulation is called **scaling** the range of values produced by `Random` method `Next`.

7.8.4 Shifting Random-Number Range

Suppose we wanted to simulate a six-sided die that has the numbers 1–6 on its faces, not 0–5. Scaling the range of values alone is not enough. So we **shift** the range of numbers produced. We could do this by adding a **shifting value**—in this case 1—to the result of method `Next`, as in

```
int face = 1 + randomNumbers.Next(6); // 1, 2, 3, 4, 5 or 6
```

The shifting value (`1`) specifies the first value in the desired set of random integers. The preceding statement assigns to `face` a random integer in the range 1–6.

7.8.5 Combining Shifting and Scaling

The third alternative of method `Next` provides a more intuitive way to express both shifting and scaling. This method receives two `int` arguments and returns a value from the first argument's value up to, but not including, the second argument's value. We could use this method to write a statement equivalent to our previous statement, as in

```
int face = randomNumbers.Next(1, 7); // 1, 2, 3, 4, 5 or 6
```

7.8.6 Rolling a Six-Sided Die

To demonstrate random numbers, let's develop an app that simulates 20 rolls of a six-sided die and displays each roll's value. Figure 7.5 shows two sample outputs, which confirm that the results of the preceding calculation are integers in the range 1–6 and that each run of the app can produce a *different* sequence of random numbers. Line 9 creates the `Random` object `randomNumbers` to produce random values. Line 15 executes 20 times in a loop to roll the die and line 16 displays the value of each roll.

```
// Fig. 7.5: RandomIntegers.cs
// Shifted and scaled random integers.
using System;

class RandomIntegers
{
   static void Main()
   {
      Random randomNumbers = new Random(); // random-number generator

      // loop 20 times
      for (int counter = 1; counter <= 20; ++counter)
      {
         // pick random integer from 1 to 6
         int face = randomNumbers.Next(1, 7);
         Console.Write($"{face}  "); // display generated value
      }

      Console.WriteLine();
   }
}
```

```
3  3  3  1  1  2  1  2  4  2  2  3  6  2  5  3  4  6  6  1
```

```
6  2  5  1  3  5  2  1  6  5  4  1  6  1  3  3  1  4  3  4
```

Fig. 7.5 | Shifted and scaled random integers.

Rolling a Six-Sided Die 60,000,000 Times

To show that the numbers produced by `Next` occur with approximately equal likelihood, let's simulate 60,000,000 rolls of a die (Fig. 7.6). Each integer from 1 to 6 should appear approximately 10,000,000 times.

```
// Fig. 7.6: RollDie.cs
// Roll a six-sided die 60,000,000 times.
using System;

class RollDie
{
   static void Main()
   {
      Random randomNumbers = new Random(); // random-number generator

      int frequency1 = 0; // count of 1s rolled
      int frequency2 = 0; // count of 2s rolled
      int frequency3 = 0; // count of 3s rolled
      int frequency4 = 0; // count of 4s rolled
      int frequency5 = 0; // count of 5s rolled
      int frequency6 = 0; // count of 6s rolled
```

Fig. 7.6 | Roll a six-sided die 60,000,000 times. (Part 1 of 2.)

```

      // summarize results of 60,000,000 rolls of a die
      for (int roll = 1; roll <= 60000000; ++roll)
      {
         int face = randomNumbers.Next(1, 7); // number from 1 to 6

         // determine roll value 1-6 and increment appropriate counter
         switch (face)
         {
            case 1:
               ++frequency1; // increment the 1s counter
               break;
            case 2:
               ++frequency2; // increment the 2s counter
               break;
            case 3:
               ++frequency3; // increment the 3s counter
               break;
            case 4:
               ++frequency4; // increment the 4s counter
               break;
            case 5:
               ++frequency5; // increment the 5s counter
               break;
            case 6:
               ++frequency6; // increment the 6s counter
               break;
         }
      }

      Console.WriteLine("Face\tFrequency"); // output headers
      Console.WriteLine($"1\t{frequency1}\n2\t{frequency2}");
      Console.WriteLine($"3\t{frequency3}\n4\t{frequency4}");
      Console.WriteLine($"5\t{frequency5}\n6\t{frequency6}");
   }
}
```

```
Face    Frequency
1       10006774
2       9993289
3       9993438
4       10006520
5       9998762
6       10001217
```

```
Face    Frequency
1       10002183
2       9997815
3       9999619
4       10006012
5       9994806
6       9999565
```

Fig. 7.6 | Roll a six-sided die 60,000,000 times. (Part 2 of 2.)

As the two sample outputs show, the values produced by method `Next` enable the app to realistically simulate rolling a six-sided die. The app uses nested control statements (the `switch` is nested inside the `for`) to determine the number of times each side of the die occurred. The `for` statement (lines 19–45) iterates 60,000,000 times. During each iteration, line 21 produces a random value from 1 to 6. This `face` value is then used as the `switch` expression (line 24). Based on the `face` value, the `switch` statement increments one of the six counter variables during each iteration of the loop. (In Section 8.4.7, we show an elegant way to replace the entire `switch` statement in this app with a single statement.) The `switch` statement has no `default` label because we have a `case` label for every possible die value that the expression in line 21 can produce. Run the app several times and observe the results. You'll see that every time you execute this apkp, it produces different results.

7.8.7 Scaling and Shifting Random Numbers

Previously, we demonstrated the statement

```
int face = randomNumbers.Next(1, 7);
```

which simulates the rolling of a six-sided die. This statement always assigns to variable `face` an integer in the range `1 ≤ face < 7`. The width of this range (i.e., the number of consecutive integers in the range) is `6`, and the starting number in the range is `1`. Referring to the preceding statement, we see that the width of the range is determined by the difference between the two integers passed to `Random` method `Next`, and the starting number of the range is the value of the first argument. We can generalize this result as

```
int number = randomNumbers.Next(shiftingValue, shiftingValue + scalingFactor);
```

where *shiftingValue* specifies the first number in the desired range of consecutive integers and *scalingFactor* specifies how many numbers are in the range.

It's also possible to choose integers at random from sets of values *other* than ranges of consecutive integers. For this purpose, it's simpler to use the version of the `Next` method that takes only *one* argument. For example, to obtain a random value from the sequence 2, 5, 8, 11 and 14, you could use the statement

```
int number = 2 + 3 * randomNumbers.Next(5);
```

In this case, `randomNumbers.Next(5)` produces values in the range 0–4. Each value produced is multiplied by 3 to produce a number in the sequence 0, 3, 6, 9 and 12. We then add 2 to that value to *shift* the range of values and obtain a value from the sequence 2, 5, 8, 11 and 14. We can generalize this result as

```
int number = shiftingValue +
   differenceBetweenValues * randomNumbers.Next(scalingFactor);
```

where *shiftingValue* specifies the first number in the desired range of values, *differenceBetweenValues* represents the difference between consecutive numbers in the sequence and *scalingFactor* specifies how many numbers are in the range.

7.8.8 Repeatability for Testing and Debugging

As we mentioned earlier in this section, the methods of class `Random` actually generate *pseudorandom* numbers based on complex mathematical calculations. Repeatedly calling any of `Random`'s methods produces a sequence of numbers that appears to be random. The calculation that produces the pseudorandom numbers uses the time of day as a **seed value** to change the sequence's starting point. Each new `Random` object seeds itself with a value based on the computer system's clock at the time the object is created, enabling each execution of an app to produce a *different* sequence of random numbers.

When debugging an app, it's sometimes useful to repeat the *same* sequence of pseudorandom numbers during each execution of the app. This repeatability enables you to prove that your app is working for a specific sequence of random numbers before you test the app with different sequences of random numbers. When repeatability is important, you can create a `Random` object as follows:

```
Random randomNumbers = new Random(seedValue);
```

The `seedValue` argument (an `int`) seeds the random-number calculation—using the *same* `seedValue` every time produces the *same* sequence of random numbers. Different seed values, of course, produce *different* sequences of random numbers.

7.9 Case Study: A Game of Chance; Introducing Enumerations

One popular game of chance is the dice game known as "craps," which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> *You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called "craps"), you lose (i.e., "the house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.*

The app in Fig. 7.7 simulates the game of craps, using methods to define the logic of the game. The `Main` method (lines 24–80) calls the `static RollDice` method (lines 83–94) as needed to roll the two dice and compute their sum. The four sample outputs show winning on the first roll, losing on the first roll, losing on a subsequent roll and winning on a subsequent roll, respectively. Variable `randomNumbers` (line 8) is declared `static`, so it can be created once during the program's execution and used in method `RollDice`.

```
1  // Fig. 7.7: Craps.cs
2  // Craps class simulates the dice game craps.
3  using System;
4
5  class Craps
6  {
7     // create random-number generator for use in method RollDice
8     private static Random randomNumbers = new Random();
9
10    // enumeration with constants that represent the game status
11    private enum Status {Continue, Won, Lost}
```

Fig. 7.7 | `Craps` class simulates the dice game craps. (Part 1 of 4.)

```
   
   // enumeration with constants that represent common rolls of the dice
   private enum DiceNames
   {
      SnakeEyes = 2,
      Trey = 3,
      Seven = 7,
      YoLeven = 11,
      BoxCars = 12
   }

   // plays one game of craps
   static void Main()
   {
      // gameStatus can contain Continue, Won or Lost
      Status gameStatus = Status.Continue;
      int myPoint = 0; // point if no win or loss on first roll

      int sumOfDice = RollDice(); // first roll of the dice

      // determine game status and point based on first roll
      switch ((DiceNames) sumOfDice)
      {
         case DiceNames.Seven: // win with 7 on first roll
         case DiceNames.YoLeven: // win with 11 on first roll
            gameStatus = Status.Won;
            break;
         case DiceNames.SnakeEyes: // lose with 2 on first roll
         case DiceNames.Trey: // lose with 3 on first roll
         case DiceNames.BoxCars: // lose with 12 on first roll
            gameStatus = Status.Lost;
            break;
         default: // did not win or lose, so remember point
            gameStatus = Status.Continue; // game is not over
            myPoint = sumOfDice; // remember the point
            Console.WriteLine($"Point is {myPoint}");
            break;
      }

      // while game is not complete
      while (gameStatus == Status.Continue) // game not Won or Lost
      {
         sumOfDice = RollDice(); // roll dice again

         // determine game status
         if (sumOfDice == myPoint) // win by making point
         {
            gameStatus = Status.Won;
         }
         else
         {
```

Fig. 7.7 | `Craps` class simulates the dice game craps. (Part 2 of 4.)

```
            // lose by rolling 7 before point
            if (sumOfDice == (int) DiceNames.Seven)
            {
               gameStatus = Status.Lost;
            }
         }
      }

      // display won or lost message
      if (gameStatus == Status.Won)
      {
         Console.WriteLine("Player wins");
      }
      else
      {
         Console.WriteLine("Player loses");
      }
   }

   // roll dice, calculate sum and display results
   static int RollDice()
   {
      // pick random die values
      int die1 = randomNumbers.Next(1, 7); // first die roll
      int die2 = randomNumbers.Next(1, 7); // second die roll

      int sum = die1 + die2; // sum of die values

      // display results of this roll
      Console.WriteLine($"Player rolled {die1} + {die2} = {sum}");
      return sum; // return sum of dice
   }
}
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 2 + 1 = 3
Player loses
```

```
Player rolled 2 + 4 = 6
Point is 6
Player rolled 3 + 1 = 4
Player rolled 5 + 5 = 10
Player rolled 6 + 1 = 7
Player loses
```

Fig. 7.7 | Craps class simulates the dice game craps. (Part 3 of 4.)

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 1 + 3 = 4
Player rolled 1 + 3 = 4
Player rolled 2 + 3 = 5
Player rolled 4 + 4 = 8
Player rolled 6 + 6 = 12
Player rolled 4 + 4 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 6 = 8
Player rolled 6 + 6 = 12
Player rolled 6 + 4 = 10
Player wins
```

Fig. 7.7 | `Craps` class simulates the dice game craps. (Part 4 of 4.)

7.9.1 Method `RollDice`

In the rules of the game, the player must roll two dice on the first roll and must do the same on all subsequent rolls. We declare method `RollDice` (lines 83–94) to roll the dice and compute and display their sum. Method `RollDice` is declared once, but it's called from two places (lines 30 and 54) in method `Main`, which contains the logic for one complete game of craps. Method `RollDice` takes no arguments, so it has an empty parameter list. Each time it's called, `RollDice` returns the sum of the dice as an `int`. Although lines 86 and 87 look the same (except for the die names), they do not necessarily produce the same result. Each of these statements produces a random value in the range 1–6. Variable `randomNumbers` (used in lines 86–87) is *not* declared in the method. Rather it's declared as a `private static` variable of the class and initialized in line 8. This enables us to create *one* `Random` object that's reused in each call to `RollDice`.

7.9.2 Method `Main`'s Local Variables

The game is reasonably involved. The player may win or lose on the first roll or may win or lose on any subsequent roll. Method `Main` (lines 24–80) uses local variable `gameStatus` (line 27) to keep track of the overall game status, local variable `myPoint` (line 28) to store the "point" if the player does not win or lose on the first roll and local variable `sumOfDice` (line 30) to maintain the sum of the dice for the most recent roll. Variable `myPoint` is initialized to `0` to ensure that the app will compile. If you do not initialize `myPoint`, the compiler issues an error, because `myPoint` is not assigned a value in every `case` of the `switch` statement—thus, the app could try to use `myPoint` before it's definitely assigned a value. By contrast, `gameStatus` does not require initialization because it's assigned a value in every branch of the `switch` statement—thus, it's guaranteed to be initialized before it's used. However, as good practice, we initialize it anyway.

7.9.3 `enum` Type `Status`

Local variable `gameStatus` (line 27) is declared to be of a new type called `Status`, which we declared in line 11. `Status` is a user-defined type called an **enumeration**, which declares a set of *constants* represented by identifiers. An enumeration is introduced by the keyword **`enum`** and a type name (in this case, `Status`). As with a class, braces (`{` and `}`) delimit the body of an `enum` declaration. Inside the braces is a comma-separated list of **enumeration constants**—by default, the first constant has the value `0` and each subsequent constant's value is incremented by `1`. The `enum` constant names must be *unique*, but the value associated with each constant need not be. Type `Status` is declared as a `private` member of class `Craps`, because `Status` is used only in that class.

Variables of type `Status` should be assigned only one of the three constants declared in the enumeration. When the game is won, the app sets local variable `gameStatus` to `Status.Won` (lines 37 and 59). When the game is lost, the app sets `gameStatus` to `Status.Lost` (lines 42 and 66). Otherwise, the app sets `gameStatus` to `Status.Continue` (line 45) to indicate that the dice must be rolled again.

Good Programming Practice 7.2

Using enumeration constants (like `Status.Won`, `Status.Lost` and `Status.Continue`) rather than literal integer values (such as 0, 1 and 2) can make code easier to read and maintain.

7.9.4 The First Roll

Line 30 in method `Main` calls `RollDice`, which picks two random values from 1 to 6, displays the value of the first die, the value of the second die and the sum of the dice, and returns the sum of the dice. Method `Main` next enters the `switch` statement at lines 33–49, which uses the `sumOfDice` value to determine whether the game has been won or lost, or whether it should continue with another roll.

7.9.5 enum Type DiceNames

The sums of the dice that would result in a win or loss on the first roll are declared in the `DiceNames` enumeration in lines 14–21. These are used in the `switch` statement's `case`s. The identifier names use casino parlance for these sums. In the `DiceNames` enumeration, we assign a value explicitly to each identifier name. When the `enum` is declared, each constant in the `enum` declaration is a constant value of type `int`. If you do not assign a value to an identifier in the `enum` declaration, the compiler will do so. If the first `enum` constant is unassigned, the compiler gives it the value `0`. If any other `enum` constant is unassigned, the compiler gives it a value one higher than that of the preceding `enum` constant. For example, in the `Status` enumeration, the compiler implicitly assigns `0` to `Status.Continue`, `1` to `Status.Won` and `2` to `Status.Lost`.

7.9.6 Underlying Type of an enum

You could also declare an `enum`'s underlying type to be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong` by writing

```
private enum MyEnum : typeName {Constant1, Constant2, ...}
```

where *typeName* represents one of the integral simple types.

7.9.7 Comparing Integers and enum Constants

If you need to compare a simple integral type value to the underlying value of an enumeration constant, you must use a cast operator to make the two types match—there are no implicit conversions between `enum` and integral types. In the `switch` expression (line 33), we use the cast operator to convert the `int` value in `sumOfDice` to type `DiceNames` and compare it to each of the constants in `DiceNames`. Lines 35–36 determine whether the player won on the first roll with `Seven` (7) or `YoLeven` (11). Lines 39–41 determine whether the player lost on the first roll with `SnakeEyes` (2), `Trey` (3) or `BoxCars` (12). After the first roll, if the game is not over, the `default` case (lines 44–48) saves `sumOfDice` in `myPoint` (line 46) and displays the point (line 47).

Additional Rolls of the Dice

If we're still trying to "make our point" (i.e., the game is continuing from a prior roll), the loop in lines 52–69 executes. Line 54 rolls the dice again. If `sumOfDice` matches `myPoint` in line 57, line 59 sets `gameStatus` to `Status.Won`, and the loop terminates because the game is complete. In line 64, we use the cast operator `(int)` to obtain the underlying value of `DiceNames.Seven` so that we can compare it to `sumOfDice`. If `sumOfDice` is equal to `Seven` (7), line 66 sets `gameStatus` to `Status.Lost`, and the loop terminates because the game is over. When the game completes, lines 72–79 display a message indicating whether the player won or lost, and the app terminates.

Control Statements in the `Craps` Example

Note the use of the various program-control mechanisms we've discussed. The `Craps` class uses two methods—`Main` and `RollDice` (called twice from `Main`)—and the `switch`, `while`, `if...else` and nested `if` control statements. Also, notice that we use multiple `case` labels in the `switch` statement to execute the same statements for sums of `Seven` and `YoLeven` (lines 35–36) and for sums of `SnakeEyes`, `Trey` and `BoxCars` (lines 39–41).

Code Snippets for Auto-Implemented Properties

Visual Studio has a feature called **code snippets** that allows you to insert *predefined code templates* into your source code. One such snippet enables you to easily create a `switch` statement with `case`s for all possible values for an `enum` type. Type `switch` in the C# code then press *Tab* twice. If you specify a variable of an `enum` type in the `switch` statement's expression and press *Enter*, a `case` for each `enum` constant will be generated automatically.

To get a list of all available code snippets, type *Ctrl* + *k*, *Ctrl* + *x*. This displays the **Insert Snippet** window in the code editor. You can navigate through the Visual C# snippet folders with the mouse to see the snippets. This feature also can be accessed by *right clicking* in the source code editor and selecting the **Insert Snippet…** menu item.

7.10 Scope of Declarations

You've seen declarations of C# entities, such as classes, methods, properties, variables and parameters. Declarations introduce names that can be used to refer to such C# entities. The **scope** of a declaration is the portion of the app that can refer to the declared entity by its unqualified name. Such an entity is said to be "in scope" for that portion of the app. This section introduces several important scope issues. The basic scope rules are as follows:

1. The scope of a parameter declaration is the body of the method in which the declaration appears.
2. The scope of a local-variable declaration is from the point at which the declaration appears to the end of the block containing the declaration.

3. The scope of a local-variable declaration that appears in the initialization section of a `for` statement's header is the body of the `for` statement and the other expressions in the header.
4. The scope of a method, property or field of a class is the entire body of the class. This enables non-`static` methods and properties of a class to use any of the class's fields, methods and properties, regardless of the order in which they're declared. Similarly, `static` methods and properties can use any of the `static` members of the class.

Any block may contain variable declarations. If a local variable or parameter in a method has the same name as a field, the field is hidden until the block terminates—in Chapter 10, we discuss how to access hidden fields. A compilation error occurs if a *nested block* in a method contains a variable with the same name as a local variable in an *outer block* of the method. The app in Fig. 7.8 demonstrates scoping issues with fields and local variables.

Error-Prevention Tip 7.1

Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method hides a field of the same name in the class.

```
// Fig. 7.8: Scope.cs
// Scope class demonstrates static- and local-variable scopes.
using System;

class Scope
{
   // static variable that's accessible to all methods of this class
   private static int x = 1;

   // Main creates and initializes local variable x
   // and calls methods UseLocalVariable and UseStaticVariable
   static void Main()
   {
      int x = 5; // method's local variable x hides static variable x

      Console.WriteLine($"local x in method Main is {x}");

      // UseLocalVariable has its own local x
      UseLocalVariable();

      // UseStaticVariable uses class Scope's static variable x
      UseStaticVariable();

      // UseLocalVariable reinitializes its own local x
      UseLocalVariable();

      // class Scope's static variable x retains its value
      UseStaticVariable();
```

Fig. 7.8 | `Scope` class demonstrates static- and local-variable scopes. (Part 1 of 2.)

```
29
30          Console.WriteLine($"\nlocal x in method Main is {x}");
31       }
32
33       // create and initialize local variable x during each call
34       static void UseLocalVariable()
35       {
36          int x = 25; // initialized each time UseLocalVariable is called
37
38          Console.WriteLine(
39             $"\nlocal x on entering method UseLocalVariable is {x}");
40          ++x; // modifies this method's local variable x
41          Console.WriteLine(
42             $"local x before exiting method UseLocalVariable is {x}");
43       }
44
45       // modify class Scope's static variable x during each call
46       static void UseStaticVariable()
47       {
48          Console.WriteLine("\nstatic variable x on entering method " +
49             $"UseStaticVariable is {x}");
50          x *= 10; // modifies class Scope's static variable x
51          Console.WriteLine("static variable x before exiting " +
52             $"method UseStaticVariable is {x}");
53       }
54    }
```

```
local x in method Main is 5

local x on entering method UseLocalVariable is 25
local x before exiting method UseLocalVariable is 26

static variable x on entering method UseStaticVariable is 1
static variable x before exiting method UseStaticVariable is 10

local x on entering method UseLocalVariable is 25
local x before exiting method UseLocalVariable is 26

static variable x on entering method UseStaticVariable is 10
static variable x before exiting method UseStaticVariable is 100

local x in method Main is 5
```

Fig. 7.8 | `Scope` class demonstrates static- and local-variable scopes. (Part 2 of 2.)

Line 8 declares and initializes the `static` variable `x` to `1`. This `static` variable is *hidden* in any block (or method) that declares a local variable named `x`. Method `Main` (lines 12–31) declares local variable `x` (line 14) and initializes it to `5`. This local variable's value is output to show that `static` variable `x` (whose value is `1`) is hidden in method `Main`. The app declares two other methods—`UseLocalVariable` (lines 34–43) and `UseStaticVariable` (lines 46–53)—that each take no arguments and do not return results. Method `Main` calls each method twice (lines 19–28). Method `UseLocalVariable` declares local variable `x` (line 36). When `UseLocalVariable` is first called (line 19), it creates local variable `x` and initializes it to `25` (line 36), outputs the value of `x` (lines 38–39), increments `x` (line 40) and outputs the value of `x` again (lines 41–42). When `UseLocalVariable` is called a second time (line 25), it re-creates local variable `x` and reinitializes it to `25`, so the output of each call to `UseLocalVariable` is identical.

Method `UseStaticVariable` does not declare any local variables. Therefore, when it refers to `x`, `static` variable `x` (line 8) of the class is used. When method `UseStaticVariable` is first called (line 22), it outputs the value (`1`) of `static` variable `x` (lines 48–49), multiplies the `static` variable `x` by `10` (line 50) and outputs the value (`10`) of `static` variable `x` again (lines 51–52) before returning. The next time method `UseStaticVariable` is called (line 28), the `static` variable has its modified value, `10`, so the method outputs `10`, then `100`. Finally, in method `Main`, the app outputs the value of local variable `x` again (line 30) to show that none of the method calls modified `Main`'s local variable `x`, because the methods all referred to variables named `x` in other scopes.

7.11 Method-Call Stack and Activation Records

To understand how C# performs method calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**. Think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's placed at the *top*—referred to as **pushing** the dish onto the stack. Similarly, when a dish is removed from the pile, it's removed from the top—referred to as **popping** the dish off the stack. Stacks are known as **last-in, first-out (LIFO) data structures**—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

7.11.1 Method-Call Stack

The **method-call stack** (sometimes referred to as the **program-execution stack**) is a data structure that works behind the scenes to support the method call/return mechanism. It also supports the creation, maintenance and destruction of each called method's local variables. As we'll see in Figs. 7.10–7.12, the stack's last-in, first-out (LIFO) behavior is *exactly* what a method needs in order to return to the method that called it.

7.11.2 Stack Frames

As each method is called, it may, in turn, call other methods, which may, in turn, call other methods—all *before* any of the methods return. Each method eventually must return control to the method that called it. So, somehow, the system must keep track of the *return addresses* that each method needs in order to return control to the method that called it. The method-call stack is the perfect data structure for handling this information. Each time a method calls another method, an entry is *pushed* onto the stack. This entry, called a **stack frame** or an **activation record**, contains the *return address* that the called method needs in order to return to the calling method. It also contains some additional information we'll soon discuss. If the called method returns instead of calling another method before returning, the stack frame for the method call is *popped*, and control transfers to the return address in the popped stack frame. The same techniques apply when a method accesses a property or when a property calls a method.

The beauty of the call stack is that each called method *always* finds the information it needs to return to its caller at the *top* of the call stack. And, if a method makes a call to another method, a stack frame for the new method call is simply *pushed* onto the call stack. Thus, the return address required by the newly called method to return to its caller is now located at the *top* of the stack.

7.11.3 Local Variables and Stack Frames

The stack frames have another important responsibility. Most methods have local variables—parameters and any local variables the method declares. Local variables need to exist while a method is executing. They need to remain active if the method makes calls to other methods. But when a called method returns to its caller, the called method's local variables need to "go away." The called method's stack frame is a perfect place to reserve the memory for the called method's local variables. That stack frame exists as long as the called method is active. When that method returns—and no longer needs its local variables—its stack frame is *popped* from the stack, and those local variables no longer exist.

7.11.4 Stack Overflow

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the method-call stack. If more method calls occur than can have their activation records stored on the method-call stack, a fatal error known as **stack overflow** occurs[2]—typically caused by infinite recursion (Section 7.16).

7.11.5 Method-Call Stack in Action

Now let's consider how the call stack supports the operation of a `Square` method (lines 15–18 of Fig. 7.9) called by `Main` (lines 8–12).

```
1   // Fig. 7.9: SquareTest.cs
2   // Square method used to demonstrate the method
3   // call stack and activation records.
4   using System;
5
6   class Program
7   {
8      static void Main()
9      {
10        int x = 10; // value to square (local variable in main)
11        Console.WriteLine($"x squared: {Square(x)}");
12     }
13
14     // returns the square of an integer
15     static int Square(int y) // y is a local variable
16     {
17        return y * y; // calculate square of y and return result
18     }
19  }
```

Fig. 7.9 | `Square` method used to demonstrate the method-call stack and activation records. (Part 1 of 2.)

2. This is how the website `stackoverflow.com` got its name. This is a popular website for getting answers to your programming questions.

```
x squared: 100
```

Fig. 7.9 | `Square` method used to demonstrate the method-call stack and activation records. (Part 2 of 2.)

First, the operating system calls `Main`—this *pushes* an activation record onto the stack (Fig. 7.10). This tells `Main` how to return to the operating system (i.e., transfer to return address `R1`) and contains the space for `Main`'s local variable `x`, which is initialized to `10`.

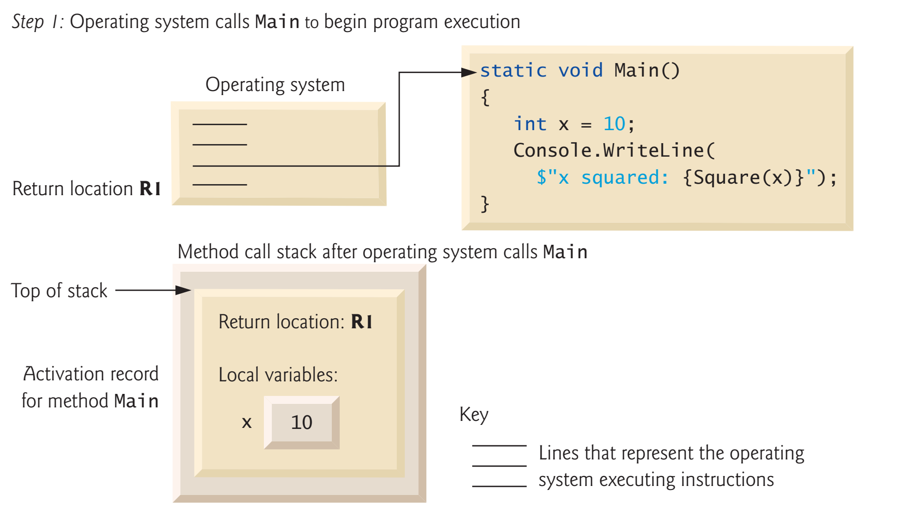


Fig. 7.10 | Method-call stack after the operating system calls `main` to execute the program.

Method `Main`—before returning to the operating system—calls method `Square` in line 11 of Fig. 7.9. This causes a stack frame for `Square` (lines 15–18) to be pushed onto the method-call stack (Fig. 7.11). This stack frame contains the return address that `Square` needs to return to `Main` (i.e., `R2`) and the memory for `Square`'s local variable `y`.

After `Square` performs its calculation, it needs to return to `Main`—and no longer needs the memory for `y`. So `Square`'s stack frame is *popped* from the stack—giving `Square` the return location in `Main` (i.e., `R2`) and losing `Square`'s local variable (*Step 3*). Figure 7.12 shows the method-call stack *after* `Square`'s activation record has been *popped*.

Method `Main` now displays the result of calling `Square` (Fig. 7.9, line 11). Reaching the closing right brace of `Main` causes its stack frame to be *popped* from the stack, giving `Main` the address it needs to return to the operating system (i.e., `R1` in Fig. 7.10)—at this point, `Main`'s local variable `x` no longer exists.

You've now seen how valuable the stack data structure is in implementing a key mechanism that supports program execution. There's a significant omission in the sequence of illustrations in this section. See if you can spot it before reading the next sentence. The call to the method `Console.Writeln`, of course, also involves the stack, which should be reflected in this section's illustrations and discussion.

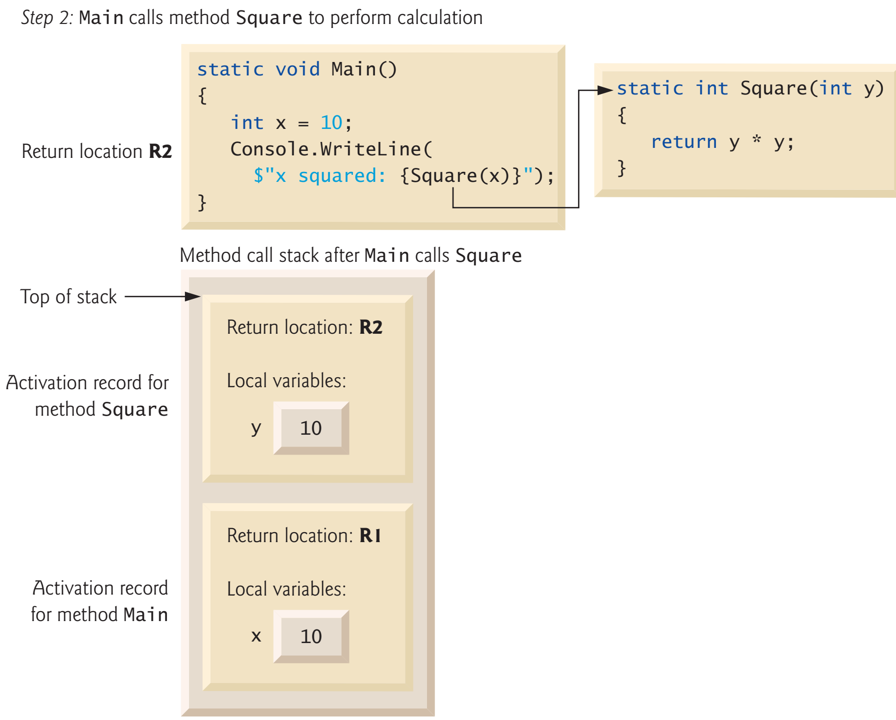


Fig. 7.11 | Method-call stack after `Main` calls `square` to perform the calculation.

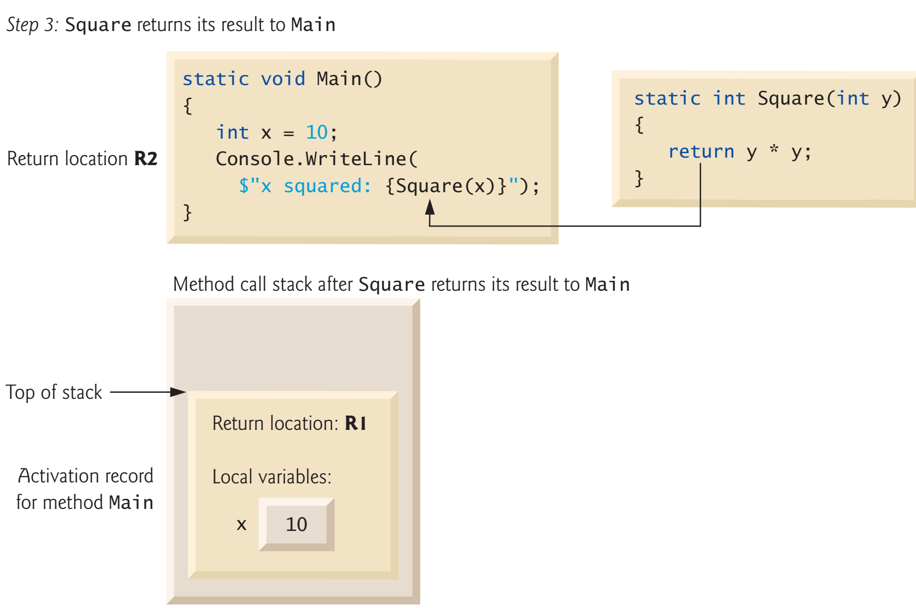


Fig. 7.12 | Method-call stack after method `square` returns to `Main`.

7.12 Method Overloading

Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters). This is called **method overloading**. When an **overloaded method** is called, the C# compiler selects the appropriate method by examining the number, types and order of the arguments in the call. Method overloading is commonly used to create several methods with the *same name* that perform the same or similar tasks, but on *different types* or *different numbers of arguments*. For example, `Random` method `Next` (Section 7.8) has overloads that accept different numbers of arguments, and `Math` method `Max` has overloads that accept different types of arguments (`int`s vs. `double`s). These find the minimum and maximum, respectively, of two values of each of the numeric simple types. Our next example demonstrates declaring and invoking overloaded methods. You'll see examples of overloaded constructors in Chapter 10.

7.12.1 Declaring Overloaded Methods

In class `MethodOverload` (Fig. 7.13), we include two `Square` methods—one that calculates the square of an `int` (and returns an `int`) and one that calculates the square of a `double` (and returns a `double`). Although these methods have the same name and similar parameter lists and bodies, you can think of them simply as *different* methods. It may help to think of the method names as "`Square` of `int`" and "`Square` of `double`," respectively.

```
// Fig. 7.13: MethodOverload.cs
// Overloaded method declarations.
using System;

class MethodOverload
{
   // test overloaded square methods
   static void Main()
   {
      Console.WriteLine($"Square of integer 7 is {Square(7)}");
      Console.WriteLine($"Square of double 7.5 is {Square(7.5)}");
   }

   // square method with int argument
   static int Square(int intValue)
   {
      Console.WriteLine($"Called square with int argument: {intValue}");
      return intValue * intValue;
   }

   // square method with double argument
   static double Square(double doubleValue)
   {
      Console.WriteLine(
         $"Called square with double argument: {doubleValue}");
      return doubleValue * doubleValue;
   }
}
```

Fig. 7.13 | Overloaded method declarations. (Part 1 of 2.)

```
Called square with int argument: 7
Square of integer 7 is 49
Called square with double argument: 7.5
Square of double 7.5 is 56.25
```

Fig. 7.13 | Overloaded method declarations. (Part 2 of 2.)

Line 10 in `Main` invokes method `Square` with the argument `7`. Literal integer values are treated as type `int`, so the method call in line 10 invokes the version of `Square` at lines 15–19 that specifies an `int` parameter. Similarly, line 11 invokes method `Square` with the argument `7.5`. Literal real-number values are treated as type `double`, so the method call in line 11 invokes the version of `Square` at lines 22–27 that specifies a `double` parameter. Each method first outputs a line of text to prove that the proper method was called in each case.

The overloaded methods in Fig. 7.13 perform the same calculation, but with two different types. C#'s generics feature provides a mechanism for writing a single "generic method" that can perform the same tasks as an entire set of overloaded methods. We discuss generic methods in Chapter 18.

7.12.2 Distinguishing Between Overloaded Methods

The compiler distinguishes overloaded methods by their **signature**—a combination of the method's name and the number, types and order of its parameters. The signature also includes the way those parameters are passed, which can be modified by the `ref` and `out` keywords (discussed in Section 7.18). If the compiler looked only at method names during compilation, the code in Fig. 7.13 would be *ambiguous*—the compiler would not know how to distinguish between the `Square` methods (lines 15–19 and 22–27). Internally, the compiler uses signatures to determine whether a class's methods are unique in that class.

For example, in Fig. 7.13, the compiler will use the method signatures to distinguish between the "`Square` of `int`" method (the `Square` method that specifies an `int` parameter) and the "`Square` of `double`" method (the `Square` method that specifies a `double` parameter). As another example, if `Method1`'s declaration begins as

```
void Method1(int a, float b)
```

then that method will have a different signature than a method that begins with

```
void Method1(float a, int b)
```

The *order* of the parameter types is important—the compiler considers the preceding two `Method1` headers to be distinct.

7.12.3 Return Types of Overloaded Methods

In discussing the logical names of methods used by the compiler, we did not mention the methods' return types. Methods *cannot* be distinguished by return type. If in a class named `MethodOverloadError` you define overloaded methods with the following headers:

```
int Square(int x)
double Square(int x)
```

which each have the *same* signature but *different* return types, the compiler generates the following error for the second `Square` method:

```
Type 'MethodOverloadError' already defines a member called 'Square'
with the same parameter types
```

Overloaded methods can have the *same* or *different* return types if the parameter lists are *different*. Also, overloaded methods need not have the same number of parameters.

Common Programming Error 7.9

Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.

7.13 Optional Parameters

Methods can have **optional parameters** that allow the calling method to *vary the number of arguments* to pass. An optional parameter specifies a **default value** that's assigned to the parameter if the optional argument is omitted. You can create methods with one or more optional parameters. *All optional parameters must be placed to the right of the method's non-optional parameters*—that is, at the end of the parameter list.

Common Programming Error 7.10

Declaring a non-optional parameter to the right of an optional one is a compilation error.

When a parameter has a default value, the caller has the *option* of passing that particular argument. For example, the method header

```
static int Power(int baseValue, int exponentValue = 2)
```

specifies an optional second parameter. Each call to `Power` must pass at least a `baseValue` argument, or a compilation error occurs. Optionally, a second argument (for the `exponentValue` parameter) can be passed to `Power`. Each optional parameter must specify a default value by using an equal (`=`) sign followed by the value. For example, the header for `Power` sets `2` as `exponentValue`'s default value. Consider the following calls to `Power`:

- `Power()`—This call generates a compilation error because this method requires a minimum of one argument.
- `Power(10)`—This call is valid because one argument (`10`) is being passed. The optional `exponentValue` is not specified in the method call, so the compiler uses `2` for the `exponentValue`, as specified in the method header.
- `Power(10, 3)`—This call is also valid because `10` is passed as the required argument and `3` is passed as the optional argument.

Figure 7.14 demonstrates an optional parameter. The program calculates the result of raising a base value to an exponent. Method `Power` (lines 15–25) specifies that its second parameter is optional. In `Main`, lines 10–11 call method `Power`. Line 10 calls the method without the optional second argument. In this case, the compiler provides the second argument, `2`, using the default value of the optional argument, which is not visible to you in the call.

```
// Fig. 7.14: CalculatePowers.cs
// Optional parameter demonstration with method Power.
using System;

class CalculatePowers
{
   // call Power with and without optional arguments
   static void Main()
   {
      Console.WriteLine($"Power(10) = {Power(10)}") ;
      Console.WriteLine($"Power(2, 10) = {Power(2, 10)}");
   }

   // use iteration to calculate power
   static int Power(int baseValue, int exponentValue = 2)
   {
      int result = 1;

      for (int i = 1; i <= exponentValue; ++i)
      {
         result *= baseValue;
      }

      return result;
   }
}
```

```
Power(10) = 100
Power(2, 10) = 1024
```

Fig. 7.14 | Optional parameter demonstration with method `Power`.

7.14 Named Parameters

Normally, when calling a method, the argument values—in order—are assigned to the parameters from left to right in the parameter list. Consider a `Time` class that stores the time of day in 24-hour clock format as `int` values representing the hour (0–23), minute (0–59) and second (0–59). Such a class might provide a `SetTime` method with optional parameters like

```
public void SetTime(int hour = 0, int minute = 0, int second = 0)
```

In the preceding method header, all three of `SetTime`'s parameters are optional. Assuming that we have a `Time` object named `t`, consider the following calls to `SetTime`:

- `t.SetTime()`—This call specifies no arguments, so the compiler assigns the default value `0` to each parameter. The resulting time is 12:00:00 AM.
- `t.SetTime(12)`—This call specifies the argument `12` for the first parameter, `hour`, and the compiler assigns the default value `0` to the `minute` and `second` parameters. The resulting time is 12:00:00 PM.
- `t.SetTime(12, 30)`—This call specifies the arguments `12` and `30` for the parameters `hour` and `minute`, respectively, and the compiler assigns the default value `0` to the parameter `second`. The resulting time is 12:30:00 PM.

- `t.SetTime(12, 30, 22)`—This call specifies the arguments `12`, `30` and `22` for the parameters `hour`, `minute` and `second`, respectively, so the compiler does not provide any default values. The resulting time is 12:30:22 PM.

What if you wanted to specify only arguments for the `hour` and `second`? You might think that you could call the method as follows:

```
t.SetTime(12, , 22); // COMPILATION ERROR
```

C# doesn't allow you to skip an argument as shown above. C# provides a feature called **named parameters**, which enable you to call methods that receive optional parameters by providing *only* the optional arguments you wish to specify. To do so, you explicitly specify the parameter's name and value—separated by a colon (`:`)—in the argument list of the method call. For example, the preceding statement can be written as follows:

```
t.SetTime(hour: 12, second: 22); // sets the time to 12:00:22
```

In this case, the compiler assigns parameter `hour` the argument `12` and parameter `second` the argument `22`. The parameter `minute` is not specified, so the compiler assigns it the default value `0`. It's also possible to specify the arguments *out of order* when using named parameters. The arguments for the required parameters must always be supplied. The *argumentName*`:` *value* syntax may be used with any method's required parameters.

6 7.15 C# 6 Expression-Bodied Methods and Properties

C# 6 introduces a new concise syntax for:

- methods that contain only a `return` statement that returns a value
- read-only properties in which the `get` accessor contains only a `return` statement
- methods that contain single statement bodies.

Consider the following `Cube` method:

```
static int Cube(int x)
{
   return x * x * x;
}
```

In C# 6, this can be expressed with an **expression-bodied method** as

```
static int Cube(int x) => x * x * x;
```

The value of `x * x * x` is returned to `Cube`'s caller implicitly. The symbol `=>` follows the method's parameter list and introduces the method's body—no braces or `return` statement are required and this can be used with `static` and non-`static` methods alike. If the expression to the right of `=>` does not have a value (e.g., a call to a method that returns `void`), the expression-bodied method must return `void`. Similarly, a read-only property can be implemented as an **expression-bodied property**. The following reimplements the `IsNoFaultState` property in Fig. 6.11 to return the result of a logical expression:

```
public bool IsNoFaultState =>
   State == "MA" || State == "NJ" || State == "NY" || State == "PA";
```

7.16 Recursion

The apps we've discussed thus far are generally structured as methods that call one another in a disciplined, hierarchical manner. For some problems, however, it's useful to have a method call itself. A **recursive method** is a method that calls itself, either *directly* or *indirectly through another method*. We consider recursion conceptually first. Then we examine an app containing a recursive method.

7.16.1 Base Cases and Recursive Calls

Recursive problem-solving approaches have a number of elements in common. When a recursive method is called to solve a problem, it actually is capable of solving *only* the simplest case(s), or **base case(s)**. If the method is called with a base case, it returns a result. If the method is called with a more complex problem, it divides the problem into two conceptual pieces (often called *divide and conquer*): a piece that the method knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of it. Because this new problem looks like the original problem, the method calls a fresh copy (or several fresh copies) of itself to work on the smaller problem; this is referred to as a **recursive call** and is also called the **recursion step**. The recursion step normally includes a `return` statement, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the method is still active (i.e., while it has not finished executing). The recursion step can result in many more recursive calls, as the method divides each new subproblem into two conceptual pieces. For the recursion to *terminate* eventually, each time the method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case(s). At that point, the method recognizes the base case and returns a result to the previous copy of the method. A sequence of returns ensues until the original method call returns the result to the caller. This process sounds complex compared with the conventional problem solving we've performed to this point.

7.16.2 Recursive Factorial Calculations

Let's write a recursive app to perform a popular mathematical calculation. The factorial of a nonnegative integer *n*, written *n!* (and pronounced "*n* factorial"), is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

1! is equal to 1 and 0! is defined to be 1. For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.

The factorial of an integer, `number`, greater than or equal to 0 can be calculated iteratively (nonrecursively) using the `for` statement as follows:

```
long factorial = 1;

for (long counter = number; counter >= 1; --counter)
{
   factorial *= counter;
}
```

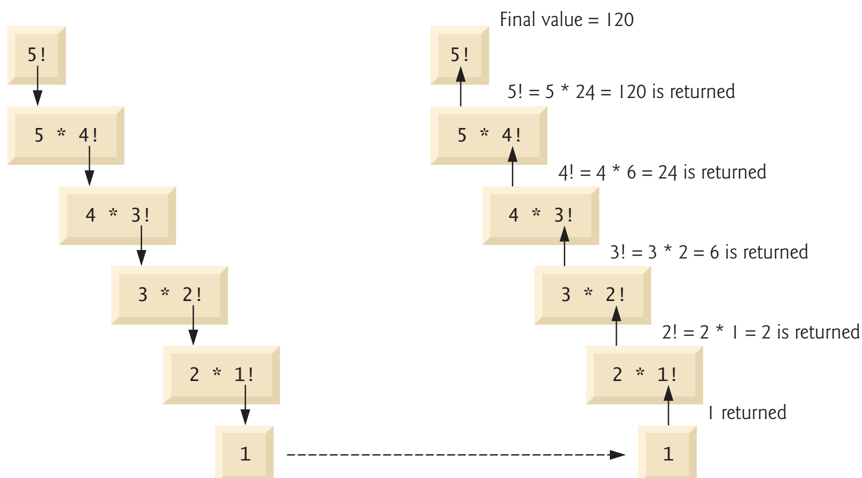
A recursive declaration of the factorial method is arrived at by observing the following relationship:

$$n! = n \cdot (n - 1)!$$

For example, 5! is clearly equal to 5 · 4!, as is shown by the following equations:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in Fig. 7.15. Figure 7.15(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion. Figure 7.15(b) shows the values returned from each recursive call to its caller until the value is calculated and returned.



(a) Sequence of recursive calls (b) Values returned from each recursive call

Fig. 7.15 | Recursive evaluation of 5!.

7.16.3 Implementing Factorial Recursively

Figure 7.16 uses recursion to calculate and display the factorials of the integers from 0 to 10. The recursive method `Factorial` (lines 17–28) first tests to determine whether a terminating condition (line 20) is `true`. If `number` is less than or equal to `1` (the base case), `Factorial` returns `1` and no further recursion is necessary. If `number` is greater than `1`, line 26 expresses the problem as the product of `number` and a recursive call to `Factorial` evaluating the factorial of `number - 1`, which is a slightly simpler problem than the original calculation, `Factorial(number)`.

```
1  // Fig. 7.16: FactorialTest.cs
2  // Recursive Factorial method.
3  using System;
```

Fig. 7.16 | Recursive `Factorial` method. (Part 1 of 2.)

```
 4
 5  class FactorialTest
 6  {
 7     static void Main()
 8     {
 9        // calculate the factorials of 0 through 10
10        for (long counter = 0; counter <= 10; ++counter)
11        {
12           Console.WriteLine($"{counter}! = {Factorial(counter)}");
13        }
14     }
15
16     // recursive declaration of method Factorial
17     static long Factorial(long number)
18     {
19        // base case
20        if (number <= 1)
21        {
22           return 1;
23        }
24        else // recursion step
25        {
26           return number * Factorial(number - 1);
27        }
28     }
29  }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Fig. 7.16 | Recursive `Factorial` method. (Part 2 of 2.)

Method `Factorial` (lines 17–28) receives a parameter of type `long` and returns a result of type `long`. As you can see in Fig. 7.16, factorial values become large quickly. We chose type `long` (which can represent relatively large integers) so that the app could calculate factorials greater than 20!. Unfortunately, the `Factorial` method produces large values so quickly that factorial values soon exceed even the maximum value that can be stored in a `long` variable. Due to the restrictions on the integral types, variables of type `float`, `double` or `decimal` might ultimately be needed to calculate factorials of larger numbers. This situation points to a weakness in some programming languages—the languages are *not easily extended* to handle the unique requirements of various apps. As you know, C# allows you to create new types. For example, you could create a type `HugeInteger` for arbitrarily large integers. This class would enable an app to calculate the factorials of larger numbers. In fact, the .NET Framework's `BigInteger` type (from namespace `System.Numerics`) supports arbitrarily large integers.

Common Programming Error 7.11

*Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause **infinite recursion**, eventually exhausting memory. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.*

7.17 Value Types vs. Reference Types

Types in C# are divided into two categories—*value types* and *reference types*.

Value Types

C#'s simple types (like `int`, `double` and `decimal`) are all **value types**. A variable of a value type simply contains a *value* of that type. For example, Fig. 7.17 shows an `int` variable named `count` that contains the value `7`.

```
int count = 7;
```

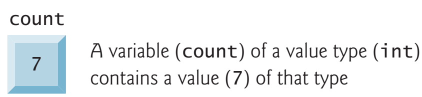


Fig. 7.17 | Value-type variable.

Reference Types

By contrast, a variable of a **reference type** (also called a **reference**) contains the *location* where the data referred to by that variable is stored. Such a variable is said to **refer to an object** in the program. For example, the statement

```
Account myAccount = new Account();
```

creates an object of our class `Account` (presented in Chapter 4), places it in memory and stores the object's reference in variable `myAccount` of type `Account`, as shown in Fig. 7.18. The `Account` object is shown with its `name` instance variable.

```
Account myAccount = new Account();
```

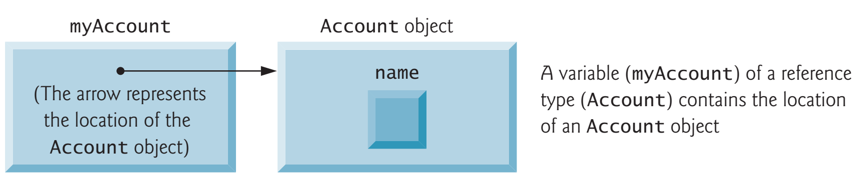


Fig. 7.18 | Reference-type variable.

Reference-Type Instance Variables Are Initialized to `null` by Default

Reference-type instance variables (such as `myAccount` in Fig. 7.18) are initialized by default to `null`. The type `string` is a reference type. For this reason, `string` instance variable `name` is shown in Fig. 7.18 with an empty box representing the `null`-valued variable. A `string` variable with the value `null` is *not* an empty `string`, which is represented by `""` or **`string.Empty`**. Rather, the value `null` represents a reference that does *not* refer to an object, whereas the empty `string` is a `string` object that does not contain any characters. In Section 7.18, we discuss value types and reference types in more detail.

Software Engineering Observation 7.4

A variable's declared type (e.g., `int` or `Account`) indicates whether the variable is of a value type or a reference type. If a variable's type is one of the simple types (Appendix), an `enum` type or a `struct` type (which we introduce in Section 10.13), then it's a value type. Classes like `Account` are reference types.

7.18 Passing Arguments By Value and By Reference

Two ways to pass arguments to methods in many programming languages are **pass-by-value** and **pass-by-reference**. When an argument is passed by *value* (the default in C#), a *copy* of its value is made and passed to the called method. Changes to the copy do *not* affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument that's been passed in the programs so far has been passed by value. When an argument is passed by *reference*, the caller gives the method the ability to access and modify the caller's original variable—no copy is passed.

To pass an object by reference into a method, simply provide as an argument in the method call the variable that refers to the object. Then, in the method body, reference the object using the corresponding parameter name. The parameter refers to the original object in memory, so the called method can access the original object directly.

In the previous section, we began discussing the differences between *value types* and *reference types*. A major difference is that:

- *value-type variables store values*, so specifying a value-type variable in a method call passes a *copy* of that variable's value to the method, whereas
- *reference-type variables store references to objects*, so specifying a reference-type variable as an argument passes the method a *copy of the reference* that refers to the object.

Even though the reference itself is passed by value, the method can still use the reference it receives to interact with—and possibly modify—the original object. Similarly, when returning information from a method via a `return` statement, the method returns a copy of the value stored in a value-type variable or a copy of the reference stored in a reference-type variable. When a reference is returned, the calling method can use that reference to interact with the referenced object.

Performance Tip 7.1

A disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

Performance Tip 7.2

Pass-by-reference improves performance by eliminating the pass-by-value overhead of copying large objects.

Software Engineering Observation 7.5

Pass-by-reference can weaken security; the called method can corrupt the caller's data.

7.18.1 `ref` and `out` Parameters

What if you would like to pass a variable by reference so the called method can modify the variable's value in the caller? To do this, C# provides keywords **`ref`** and **`out`**.

`ref` Parameters

Applying the `ref` keyword to a parameter declaration allows you to pass a variable to a method by reference—the method will be able to modify the original variable in the caller. Keyword `ref` is used for variables that already have been initialized in the calling method.

Common Programming Error 7.12

When a method call contains an uninitialized variable as an argument to a `ref` parameter, the compiler generates an error.

`out` Parameters

Preceding a parameter with keyword `out` creates an **output parameter**. This indicates to the compiler that the argument will be passed into the called method by reference and that the called method will assign a value to the original variable in the caller. This also prevents the compiler from generating an error message for an uninitialized variable that's passed as an argument to a method.

Common Programming Error 7.13

If the method does not assign a value to the `out` parameter in every possible path of execution, the compiler generates an error. Also, reading an `out` parameter before it's assigned a value is also a compilation error.

Software Engineering Observation 7.6

A method can return only one value to its caller via a return statement, but can return many values by specifying multiple output (`ref` and/or `out`) parameters.

Passing Reference-Type Variables by Reference

You also can pass a reference-type variable by reference, which allows you to modify it so that it refers to a new object. Passing a reference by reference is a tricky but powerful technique that we discuss in Section 8.13.

Software Engineering Observation 7.7

By default, C# does not allow you to choose whether to pass each argument by value or by reference. Value types are passed by value. Objects are not passed to methods; rather, references to objects are passed—the references themselves are passed by value. When a method receives a reference to an object, the method can manipulate the object directly, but the reference value cannot be changed to refer to a new object.

7.18.2 Demonstrating `ref`, `out` and Value Parameters

The app in Fig. 7.19 uses the `ref` and `out` keywords to manipulate integer values. The class contains three methods that calculate the square of an integer.

```
// Fig. 7.19: ReferenceAndOutputParameters.cs
// Reference, output and value parameters.
using System;

class ReferenceAndOutputParameters
{
   // call methods with reference, output and value parameters
   static void Main()
   {
      int y = 5; // initialize y to 5
      int z; // declares z, but does not initialize it

      // display original values of y and z
      Console.WriteLine($"Original value of y: {y}");
      Console.WriteLine("Original value of z: uninitialized\n");

      // pass y and z by reference
      SquareRef(ref y); // must use keyword ref
      SquareOut(out z); // must use keyword out

      // display values of y and z after they’re modified by
      // methods SquareRef and SquareOut, respectively
      Console.WriteLine($"Value of y after SquareRef: {y}");
      Console.WriteLine($"Value of z after SquareOut: {z}\n");

      // pass y and z by value
      Square(y);
      Square(z);

      // display values of y and z after they’re passed to method Square
      // to demonstrate that arguments passed by value are not modified
      Console.WriteLine($"Value of y after Square: {y}");
      Console.WriteLine($"Value of z after Square: {z}");
   }

   // uses reference parameter x to modify caller's variable
   static void SquareRef(ref int x)
   {
      x = x * x; // squares value of caller's variable
   }

   // uses output parameter x to assign a value
   // to an uninitialized variable
   static void SquareOut(out int x)
   {
      x = 6; // assigns a value to caller's variable
      x = x * x; // squares value of caller's variable
   }
```

Fig. 7.19 | Reference, output and value parameters. (Part 1 of 2.)

```
49
50      // parameter x receives a copy of the value passed as an argument,
51      // so this method cannot modify the caller's variable
52      static void Square(int x)
53      {
54         x = x * x;
55      }
56   }
```

```
Original value of y: 5
Original value of z: uninitialized

Value of y after SquareRef: 25
Value of z after SquareOut: 36

Value of y after Square: 25
Value of z after Square: 36
```

Fig. 7.19 | Reference, output and value parameters. (Part 2 of 2.)

Method `SquareRef` (lines 37–40) multiplies its parameter `x` by itself and assigns the new value to `x`. `SquareRef`'s parameter is declared as `ref int`, which indicates that the argument passed to this method must be an integer that's passed by reference. Because the argument is passed by reference, the assignment at line 39 modifies the original argument's value in the caller.

Method `SquareOut` (lines 44–48) assigns its parameter the value `6` (line 46), then squares that value. `SquareOut`'s parameter is declared as `out int`, which indicates that the argument passed to this method must be an integer that's passed by reference and that the argument does *not* need to be initialized in advance.

Method `Square` (lines 52–55) multiplies its parameter `x` by itself and assigns the new value to `x`. When this method is called, a *copy* of the argument is passed to the parameter `x`. Thus, even though parameter `x` is modified in the method, the original value in the caller is *not* modified.

Method `Main` (lines 8–34) invokes methods `SquareRef`, `SquareOut` and `Square`. We begin by initializing variable `y` to `5` and declaring, but *not* initializing, variable `z`. Lines 18–19 call methods `SquareRef` and `SquareOut`. Notice that when you pass a variable to a method with a reference parameter, you must precede the argument with the same keyword (`ref` or `out`) that was used to declare the reference parameter. Lines 23–24 display the values of `y` and `z` after the calls to `SquareRef` and `SquareOut`. Notice that `y` has been changed to `25` and `z` has been set to `36`.

Lines 27–28 call method `Square` with `y` and `z` as arguments. In this case, both variables are passed by *value*—only *copies* of their values are passed to `Square`. As a result, the values of `y` and `z` remain `25` and `36`, respectively. Lines 32–33 output the values of `y` and `z` to show that they were *not* modified.

Common Programming Error 7.14

The `ref` and `out` arguments in a method call must match the `ref` and `out` parameters specified in the method declaration; otherwise, a compilation error occurs.

7.19 Wrap-Up

In this chapter, we discussed the difference between non-`static` and `static` methods, and we showed how to call `static` methods by preceding the method name with the name of the class in which it appears and the member-access operator (`.`). You saw that the `Math` class in the .NET Framework Class Library provides many `static` methods to perform mathematical calculations. We also discussed `static` class members and why method `Main` is declared `static`.

We presented several commonly used Framework Class Library namespaces. You learned how to use operator `+` to perform `string` concatenations. You also learned how to declare constants with the `const` keyword and how to define sets of named constants with `enum` types. We demonstrated simulation techniques and used class `Random` to generate sets of random numbers. We discussed the scope of fields and local variables in a class. You saw how to overload methods in a class by providing methods with the same name but different signatures. You learned how to use optional and named parameters.

We showed the concise notation of C# 6's expression-bodied methods and read-only properties for implementing methods and read-only property `get` accessors that contain only a `return` statement. We discussed how recursive methods call themselves, breaking larger problems into smaller subproblems until eventually the original problem is solved. You learned the differences between value types and reference types with respect to how they're passed to methods, and how to use the `ref` and `out` keywords to pass arguments by reference.

In Chapter 8, you'll maintain lists and tables of data in arrays. You'll see a more elegant implementation of the app that rolls a die 60,000,000 times and two versions of a `GradeBook` case study. You'll also access an app's command-line arguments that are passed to method `Main` when a console app begins execution.

Index

Symbols

A

B

C

D

F

H

I

J

K

L

N

U

V

W