

# Microsoft Visual C# Step by Step

Tenth Edition



Professional



John Sharp

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



# Microsoft Visual C# Step by Step

Tenth Edition

John Sharp

MICROSOFT VISUAL C# STEP BY STEP, TENTH EDITION

Published with the authorization of Microsoft Corporation by:  
Pearson Education, Inc.

Copyright © 2022 by Agylia Group Ltd.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearson.com/permissions](http://www.pearson.com/permissions).

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-13-761983-2

ISBN-10: 0-13-761983-9

Library of Congress Control Number: 2022930224

ScoutAutomatedPrintCode

#### TRADEMARKS

Microsoft and the trademarks listed at <http://www.microsoft.com> on the "Trademarks" webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

#### WARNING AND DISCLAIMER

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author, the publisher, and Microsoft Corporation shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs accompanying it.

#### SPECIAL SALES

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

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

**Editor-in-Chief:** Brett Bartow

**Executive Editor:** Loretta Yates

**Sponsoring Editor:** Charvi Arora

**Development Editor:** Kate Shoup

**Managing Editor:** Sandra Schroeder

**Senior Project Editor:** Tracey Croom

**Production Editor:** Dan Foster

**Copy Editor:** Dan Foster

**Indexer:** Valerie Haynes Perry

**Proofreader:** Kim Wimpsett

**Technical Editor:** David Franson

**Editorial Assistant:** Cindy Teeters

**Cover Designer:** Twist Creative, Seattle

**Compositor:** Danielle Foster

## Pearson's Commitment to Diversity, Equity, and Inclusion

---

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at <https://www.pearson.com/report-bias.html>.



# Contents at a Glance

<i>Acknowledgments</i>	<i>xxi</i>
<i>About the author</i>	<i>xxiii</i>
<i>Introduction</i>	<i>xxv</i>

## **PART I**    **INTRODUCING MICROSOFT VISUAL C# AND MICROSOFT VISUAL STUDIO 2022**

---

CHAPTER 1	Welcome to C#	3
CHAPTER 2	Working with variables, operators, and expressions	45
CHAPTER 3	Writing methods and applying scope	73
CHAPTER 4	Using decision statements	111
CHAPTER 5	Using compound assignment and iteration statements	133
CHAPTER 6	Managing errors and exceptions	153

## **PART II**    **UNDERSTANDING THE C# OBJECT MODEL**

---

CHAPTER 7	Creating and managing classes and objects	181
CHAPTER 8	Understanding values and references	205
CHAPTER 9	Creating value types with enumerations and structures	231
CHAPTER 10	Using arrays	251
CHAPTER 11	Understanding parameter arrays	277
CHAPTER 12	Working with inheritance	289
CHAPTER 13	Creating interfaces and defining abstract classes	311
CHAPTER 14	Using garbage collection and resource management	339

## **PART III**    **UNDERSTANDING THE C# OBJECT MODEL**

---

CHAPTER 15	Implementing properties to access fields	365
CHAPTER 16	Handling binary data and using indexers	395
CHAPTER 17	Introducing generics	413
CHAPTER 18	Using collections	445
CHAPTER 19	Enumerating collections	469
CHAPTER 20	Decoupling application logic and handling events	483

CHAPTER 21	Querying in-memory data by using query expressions	513
CHAPTER 22	Operator overloading	537
<b>PART IV</b>	<b>BUILDING UNIVERSAL WINDOWS PLATFORM APPLICATIONS WITH C#</b>	
<hr/>		
CHAPTER 23	Improving throughput by using tasks	559
CHAPTER 24	Improving response time by performing asynchronous operations	599
CHAPTER 25	Implementing the user interface for a Universal Windows Platform app	641
CHAPTER 26	Displaying and searching for data in a Universal Windows Platform app	687
CHAPTER 27	Accessing a remote database from a Universal Windows Platform app	717
	<i>Index</i>	771

# Contents

<i>Acknowledgments</i> .....	xxi
<i>About the author</i> .....	xxiii
<i>Introduction</i> .....	xxv

## **PART I    INTRODUCING MICROSOFT VISUAL C# AND MICROSOFT VISUAL STUDIO 2022**

---

<b>Chapter 1</b>	<b>Welcome to C#</b>	<b>3</b>
	Writing your first C# program .....	3
	Beginning programming with the Visual Studio 2022 environment .....	9
	Writing your first program using Visual Studio 2022 .....	14
	Using namespaces .....	21
	Namespaces and assemblies .....	23
	Commenting code .....	24
	Creating a graphical application .....	24
	Examining the Universal Windows Platform app .....	37
	Adding code to the graphical application .....	40
	Summary .....	43
	Quick Reference .....	44
<b>Chapter 2</b>	<b>Working with variables, operators, and expressions</b>	<b>45</b>
	Understanding statements .....	45
	Using identifiers .....	46
	Identifying keywords .....	46
	Using variables .....	48
	Naming variables .....	48
	Declaring variables .....	48
	Specifying numeric values .....	49



Working with primitive data types . . . . .	50
Unassigned local variables . . . . .	51
Displaying primitive data type values . . . . .	51
Using arithmetic operators . . . . .	59
Operators and types . . . . .	59
Examining arithmetic operators . . . . .	61
Controlling precedence . . . . .	67
Using associativity to evaluate expressions . . . . .	68
Associativity and the assignment operator . . . . .	68
Incrementing and decrementing variables . . . . .	69
Prefix and postfix . . . . .	70
Declaring implicitly typed local variables . . . . .	70
Summary . . . . .	71
Quick Reference . . . . .	72

### **Chapter 3 Writing methods and applying scope 73**

Creating methods . . . . .	73
Declaring a method . . . . .	74
Returning data from a method . . . . .	75
Using expression-bodied methods . . . . .	76
Calling methods . . . . .	77
Specifying the method call syntax . . . . .	77
Returning multiple values from a method . . . . .	80
Applying scope . . . . .	82
Defining local scope . . . . .	83
Defining class scope . . . . .	83
Overloading methods . . . . .	84
Writing methods . . . . .	85
Using the Visual Studio Debugger to step through methods . . . . .	89
Refactoring code . . . . .	96
Nesting methods . . . . .	97

Using optional parameters and named arguments .....	100
Defining optional parameters .....	102
Passing named arguments .....	102
Resolving ambiguities with optional parameters and named arguments .....	103
Summary .....	108
Quick reference .....	109
<b>Chapter 4 Using decision statements</b>	<b>111</b>
Declaring Boolean variables .....	111
Using Boolean operators .....	112
Understanding equality and relational operators .....	112
Understanding conditional logical operators .....	113
Short-circuiting .....	114
Summarizing operator precedence and associativity .....	114
Pattern matching .....	115
Using if statements to make decisions .....	116
Understanding if statement syntax .....	116
Using blocks to group statements .....	117
Cascading if statements .....	118
Using switch statements .....	124
Understanding switch statement syntax .....	124
Following the switch statement rules .....	125
Using switch expressions with pattern matching .....	129
Summary .....	131
Quick reference .....	132
<b>Chapter 5 Using compound assignment and iteration statements</b>	<b>133</b>
Using compound assignment operators .....	133
Writing while statements .....	135
Writing for statements .....	140

Writing do statements .....	142
Summary .....	151
Quick reference .....	151

## **Chapter 6 Managing errors and exceptions 153**

Trying code and catching exceptions .....	154
Unhandled exceptions .....	155
Using multiple catch handlers .....	156
Catching multiple exceptions .....	157
Filtering exceptions .....	158
Propagating exceptions .....	163
Using checked and unchecked integer arithmetic .....	165
Writing checked statements .....	166
Writing checked expressions .....	166
Throwing exceptions .....	170
Using throw expressions .....	174
Using a finally block .....	175
Summary .....	176
Quick reference .....	177

## **PART II UNDERSTANDING THE C# OBJECT MODEL**

---

### **Chapter 7 Creating and managing classes and objects 181**

Understanding classification .....	181
The purpose of encapsulation .....	182
Defining and using a class .....	182
Controlling accessibility .....	184
Working with constructors .....	186
Overloading constructors .....	187
Deconstructing an object .....	196

Understanding static methods and data .....	197
Creating a shared field .....	198
Creating a static field by using the const keyword .....	199
Understanding static classes .....	199
Static using statements .....	200
Anonymous classes .....	202
Summary .....	203
Quick reference .....	204

## **Chapter 8 Understanding values and references 205**

Copying value type variables and classes .....	205
Understanding null values and nullable types .....	211
The null-conditional and null-coalescing operators .....	212
Using nullable types .....	213
Understanding the properties of nullable types .....	214
Using ref and out parameters .....	215
Creating ref parameters .....	216
Creating out parameters .....	216
How computer memory is organized .....	219
Using the stack and the heap .....	220
The System.Object class .....	221
Boxing .....	222
Unboxing .....	222
Casting data safely .....	224
The is operator .....	224
The as operator .....	225
The switch statement revisited .....	225
Summary .....	229
Quick reference .....	229

<b>Chapter 9</b>	<b>Creating value types with enumerations and structures</b>	<b>231</b>
	Working with enumerations . . . . .	231
	Declaring an enumeration . . . . .	231
	Using an enumeration . . . . .	232
	Choosing enumeration literal values . . . . .	233
	Choosing an enumeration's underlying type . . . . .	233
	Working with structures . . . . .	236
	Declaring a structure . . . . .	237
	Understanding differences between structures and classes . . . . .	238
	Declaring structure variables . . . . .	240
	Understanding structure initialization . . . . .	240
	Copying structure variables . . . . .	245
	Summary . . . . .	248
	Quick reference . . . . .	249
<b>Chapter 10</b>	<b>Using arrays</b>	<b>251</b>
	Declaring array variables . . . . .	252
	Creating an array instance . . . . .	252
	Populating and using an array . . . . .	253
	Creating an implicitly typed array . . . . .	254
	Accessing an individual array element . . . . .	255
	Accessing a series of array elements . . . . .	255
	Iterating through an array . . . . .	256
	Passing arrays as parameters or return values for a method . . . . .	257
	Copying arrays . . . . .	259
	Using multidimensional arrays . . . . .	260
	Creating jagged arrays . . . . .	261
	Accessing arrays that contain value types . . . . .	272
	Summary . . . . .	275
	Quick reference . . . . .	275

<b>Chapter 11</b>	<b>Understanding parameter arrays</b>	<b>277</b>
	Overloading: a recap .....	277
	Using array arguments .....	278
	Declaring a params array .....	279
	Using params object[ ] .....	281
	Using a params array .....	283
	Comparing parameter arrays and optional parameters .....	286
	Summary .....	288
	Quick reference .....	288
<b>Chapter 12</b>	<b>Working with inheritance</b>	<b>289</b>
	What is inheritance? .....	289
	Using inheritance .....	290
	The System.Object class revisited .....	292
	Calling base-class constructors .....	292
	Assigning classes .....	293
	Declaring new methods .....	295
	Declaring virtual methods .....	296
	Declaring override methods .....	297
	Understanding protected access .....	300
	Creating extension methods .....	305
	Summary .....	309
	Quick reference .....	310
<b>Chapter 13</b>	<b>Creating interfaces and defining abstract classes</b>	<b>311</b>
	Understanding interfaces .....	311
	Defining an interface .....	312
	Implementing an interface .....	313
	Referencing a class through its interface .....	314
	Working with multiple interfaces .....	315
	Explicitly implementing an interface .....	316
	Handling versioning with interfaces .....	318
	Interface restrictions .....	319
	Defining and using interfaces .....	320

Abstract classes .....	329
Abstract methods .....	330
Sealed classes .....	330
Sealed methods .....	331
Implementing and using an abstract class .....	331
Summary .....	337
Quick reference .....	338

## **Chapter 14 Using garbage collection and resource management 339**

The life and times of an object .....	339
Writing finalizers .....	340
Why use the garbage collector? .....	343
How does the garbage collector work? .....	344
Recommendations .....	345
Resource management .....	345
Disposal methods .....	346
Exception-safe disposal .....	346
The using statement and the IDisposable interface .....	347
Calling the Dispose method from a finalizer .....	349
Implementing exception-safe disposal .....	351
Handling asynchronous disposal .....	359
Summary .....	361
Quick reference .....	361

## **PART III UNDERSTANDING THE C# OBJECT MODEL**

---

### **Chapter 15 Implementing properties to access fields 365**

Implementing encapsulation by using methods .....	366
What are properties? .....	367
Using properties .....	370
Read-only properties .....	371
Write-only properties .....	371
Property accessibility .....	372

Understanding property restrictions. . . . .	373
Declaring interface properties. . . . .	375
Replacing methods with properties. . . . .	376
Pattern-matching with properties . . . . .	380
Generating automatic properties. . . . .	381
Initializing objects by using properties. . . . .	383
Automatic properties and immutability. . . . .	385
Using records with properties to implement lightweight structures . . . . .	388
Summary . . . . .	392
Quick reference . . . . .	393
<b>Chapter 16 Handling binary data and using indexers</b>	<b>395</b>
What is an indexer? . . . . .	395
Storing binary values . . . . .	396
Displaying binary values . . . . .	397
Manipulating binary values . . . . .	397
Solving the same problems using indexers . . . . .	398
Understanding indexer accessors. . . . .	400
Comparing indexers and arrays . . . . .	401
Indexers in interfaces . . . . .	403
Using indexers in a Windows application. . . . .	404
Summary . . . . .	410
Quick reference . . . . .	410
<b>Chapter 17 Introducing generics</b>	<b>413</b>
The problem: Issues with the object type . . . . .	413
The generics solution . . . . .	417
Generics vs. generalized classes . . . . .	419
Generics and constraints . . . . .	419
Creating a generic class . . . . .	420
The theory of binary trees. . . . .	420
Building a binary tree class by using generics. . . . .	423



Creating a generic method.....	433
Defining a generic method to build a binary tree.....	434
Variance and generic interfaces.....	438
Covariant interfaces.....	440
Contravariant interfaces.....	441
Summary.....	444
Quick reference.....	444

## **Chapter 18 Using collections 445**

What are collection classes?.....	445
The List<T> collection class.....	447
The LinkedList<T> collection class.....	449
The Queue<T> collection class.....	450
The PriorityQueue<TElement, TPriority> collection class.....	451
The Stack<T> collection class.....	452
The Dictionary<TKey, TValue> collection class.....	453
The SortedList<TKey, TValue> collection class.....	454
The HashSet<T> collection class.....	455
Using collection initializers.....	457
Find methods, predicates, and lambda expressions.....	458
The forms of lambda expressions.....	460
Lambda expressions and anonymous methods.....	461
Comparing arrays and collections.....	462
Summary.....	466
Quick reference.....	467

## **Chapter 19 Enumerating collections 469**

Enumerating the elements in a collection.....	469
Manually implementing an enumerator.....	470
Implementing the IEnumerable interface.....	475
Implementing an enumerator by using an iterator.....	477
A simple iterator.....	478
Defining an enumerator for the Tree<TItem> class by using an iterator.....	479

Summary .....	482
Quick reference .....	482
<b>Chapter 20 Decoupling application logic and handling events</b>	<b>483</b>
Understanding delegates .....	484
Examples of delegates in the .NET class library .....	485
The automated factory scenario .....	487
Declaring and using delegates .....	490
Lambda expressions and delegates .....	499
Enabling notifications by using events .....	500
Declaring an event .....	501
Subscribing to an event .....	502
Unsubscribing from an event .....	502
Raising an event .....	502
Understanding user-interface events .....	503
Using events .....	504
Summary .....	510
Quick reference .....	511
<b>Chapter 21 Querying in-memory data by using query expressions</b>	<b>513</b>
What is LINQ? .....	513
Using LINQ in a C# application .....	514
Selecting data .....	516
Filtering data .....	518
Ordering, grouping, and aggregating data .....	519
Joining data .....	521
Using query operators .....	522
Querying data in Tree<TItem> objects .....	525
LINQ and deferred evaluation .....	530
Summary .....	534
Quick reference .....	534

<b>Chapter 22</b>	<b>Operator overloading</b>	<b>537</b>
	Understanding operators .....	537
	Operator constraints .....	538
	Overloaded operators .....	539
	Creating symmetric operators .....	540
	Understanding compound assignment evaluation .....	541
	Declaring increment and decrement operators .....	542
	Comparing operators in structures and classes .....	542
	Defining operator pairs .....	543
	Implementing operators .....	544
	Overriding the equality operators .....	547
	Understanding conversion operators .....	550
	Providing built-in conversions .....	551
	Implementing user-defined conversion operators .....	552
	Creating symmetric operators, revisited .....	553
	Writing conversion operators .....	553
	Summary .....	555
	Quick reference .....	556

---

**PART IV**      **BUILDING UNIVERSAL WINDOWS PLATFORM  
APPLICATIONS WITH C#**

---

<b>Chapter 23</b>	<b>Improving throughput by using tasks</b>	<b>559</b>
	Why perform multitasking by using parallel processing? .....	559
	The rise of the multicore processor .....	560
	Implementing multitasking by using Microsoft .NET .....	561
	Tasks, threads, and the ThreadPool .....	562
	Creating, running, and controlling tasks .....	563
	Using the Task class to implement parallelism .....	566
	Abstracting tasks by using the Parallel class .....	576
	When not to use the Parallel class .....	580

Canceling tasks and handling exceptions . . . . .	582
The mechanics of cooperative cancellation. . . . .	582
Handling task exceptions by using the AggregateException class. . . . .	594
Using continuations with canceled and faulted tasks . . . . .	596
Summary . . . . .	596
Quick reference . . . . .	597

**Chapter 24 Improving response time by performing asynchronous operations 599**

Implementing asynchronous methods. . . . .	600
Defining asynchronous methods: the problem . . . . .	600
Defining asynchronous methods: the solution. . . . .	603
Defining asynchronous methods that return values. . . . .	609
Asynchronous method pitfalls. . . . .	610
Asynchronous methods and the Windows Runtime APIs. . . . .	611
Tasks, memory allocation, and efficiency. . . . .	613
Using PLINQ to parallelize declarative data access. . . . .	616
Using PLINQ to improve performance while iterating through a collection . . . . .	616
Canceling a PLINQ query . . . . .	621
Synchronizing concurrent access to data. . . . .	621
Locking data . . . . .	625
Synchronization primitives for coordinating tasks . . . . .	625
Canceling synchronization . . . . .	627
The concurrent collection classes. . . . .	628
Using a concurrent collection and a lock to implement thread-safe data access . . . . .	629
Summary . . . . .	639
Quick reference . . . . .	639

**Chapter 25 Implementing the user interface for a Universal Windows Platform app 641**

Features of a Universal Windows Platform app . . . . .	643
Using the Blank App template to build a Universal Windows Platform app . . . . .	645

Implementing a scalable user interface .....	649
Implementing a tabular layout by using a Grid control .....	659
Adapting the layout by using the Visual State Manager .....	667
Applying styles to a UI .....	674
Summary .....	685
Quick reference .....	686
<b>Chapter 26 Displaying and searching for data in a Universal Windows Platform app</b>	<b>687</b>
Implementing the Model-View-ViewModel pattern .....	687
Displaying data by using data binding .....	688
Modifying data by using data binding .....	694
Using data binding with a ComboBox control .....	699
Creating a ViewModel .....	702
Adding commands to a ViewModel .....	706
Summary .....	715
Quick reference .....	716
<b>Chapter 27 Accessing a remote database from a Universal Windows Platform app</b>	<b>717</b>
Retrieving data from a database .....	717
Creating an entity model .....	728
Creating and using a REST web service .....	735
Updating the UWP application to use the web service .....	752
Searching for data in the Customers app .....	761
Inserting, updating, and deleting data through a REST web service .....	767
Summary .....	769
Quick reference .....	770
<i>Index</i> .....	771

# Acknowledgments

Ho boy! Welcome to the 10<sup>th</sup> edition. In the acknowledgments to previous editions, I have made references to painting the Forth Railway Bridge and Sisyphus pushing the rock as never-ending tasks. In the future, maybe the role of updating *Microsoft C# Step By Step* will be added to this legendary list. That said, writing and updating books is far more rewarding than wielding a brush or rolling a stone up a hill forever and a day, with the added bonus that I can retire at some point.

Despite the fact that my name is on the cover, authoring a book such as this is far from a one-man project. I'd like to thank the following people who have provided unstinting support and assistance throughout this endeavor.

First, Loretta Yates at Pearson Education, who took on the role of prodding me into action and ever-so-gently tying me down to well-defined deliverables and hand-off dates. Without her initial impetus and cajoling, this project would not have gotten off the ground.

Next, Charvi Arora and her tireless team of editors, especially Kate Shoup and Dan Foster, who ensured that my grammar remained at least semi-acceptable and picked up on the missing words and nonsense phrases in the text. Also, David Fransen, who had the unenviable task of reviewing and testing the code and exercises. I know from experience that this can be a thankless and frustrating task, but the hours spent and the resulting feedback can only make for a better book. Of course, any errors that remain are entirely my responsibility, and I am happy to listen to feedback from any reader.

As ever, I must also thank Diana, my better half, who keeps me sane, fed, and watered. During Covid-19 lockdown, she felt that our house wasn't crowded enough, so she brought two rather manic kittens into the family. The dogs are now terrified, but we have endless hours of fun putting the curtains back up and playing "hunt the mouse/frog/spider or whatever they have captured and brought indoors." I wouldn't have home-life any other way.

And lastly, to James and Frankie, who have both now flown the nest. James has spent the last couple of years working for the British government in Manila (he says). Judging by the photos, it seems more like he has been on a touring holiday of the beaches of Southeast Asia. Frankie has remained closer to home so she can pop in and catch the mice/frogs/spiders from time to time. By the way, to those developers she manages at her place of work, it's time for you to make her a cup of tea!



# About the author

**JOHN SHARP** is a principal technologist for CM Group Ltd, part of the Civica Group, a software development and consultancy company in the United Kingdom. He is well versed as a software consultant, developer, author, and trainer, with more than 35 years of experience, ranging from Pascal programming on CP/M and C/Oracle application development on various flavors of UNIX to the design of C# and JavaScript distributed applications and development on Windows 11 and Microsoft Azure. He also spends much of his time writing courseware for Microsoft, focusing on areas such as data science using R and Python, big data processing with Spark and CosmosDB, SQL Server, NoSQL, web services, Blazor, cross-platform development with frameworks such as Xamarin and MAUI, and scalable application architectures with Azure.





# Introduction

A lot has changed in the last 20 years. For a laugh, I sometimes retrieve my copy of *Microsoft C# Step By Step*, first edition, released in 2001, and wonder at my naive innocence back in those days. Surely, C# was the peak of programming language perfection at that time. C# and the .NET Framework hit the world of development with a bang, and the reverberations continue to this day. However, rather than dying away, they rumble through software development with increased significance. Rather than being a single-platform approach as the naysayers of 2001 originally screamed, C# and .NET have shown themselves to be a complete multiplatform solution, whether you're building applications for Windows, macOS, Linux, or Android. Additionally, C# and .NET have proved themselves the runtime of choice for many cloud-based systems. Where would Azure be without them?

In the past, most common programming languages went through occasional updates, often spread several years apart. For example, if you look at Fortran, you will see standards named Fortran 66, Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, and Fortran 2018. That's seven updates in the last 55 years. While this relatively slow cycle of change promotes stability, it can also lead to stagnation. The issue is that the nature of problems that developers must address changes rapidly, and the tools they depend on should ideally keep pace so that they can develop effective solutions. Microsoft .NET provides a continually evolving framework, and C# undergoes frequent updates to make the best use of the platform. So, in contrast to Fortran, C# has undergone a rapid evolution since it was first released—six versions in the last five years alone, with another update due in 2022. The C# language still supports code written 20+ years ago, but these days the additions and enhancements to the language enable you to create solutions using more elegant code and concise constructs. For this reason, I make periodic updates to this book; this is now the 10<sup>th</sup> edition!

If you're interested, the following list contains a brief history of C#:

- C# 1.0 made its public debut in 2001.
- C# 2.0, with Visual Studio 2005, provided several important new features, including generics, iterators, and anonymous methods.
- C# 3.0, which was released with Visual Studio 2008, added extension methods, lambda expressions, and, most famously of all, the Language-Integrated Query (LINQ) facility.

- C# 4.0, released in 2010, provided further enhancements that improved its interoperability with other languages and technologies. These features included support for named and optional arguments and the dynamic type, which indicates that the language runtime should implement late binding for an object. Important additions to the .NET Framework, released concurrently with C# 4.0, were the classes and types that constitute the Task Parallel Library (TPL). Using the TPL, you can build highly scalable applications that can take full advantage of multicore processors.
- C# 5.0 added native support for asynchronous task-based processing through the `async` method modifier and the `await` operator.
- C# 6.0 was an incremental upgrade with features designed to make life simpler for developers. These features included items such as string interpolation (you need never use `String.Format` again!), enhancements to the ways in which properties are implemented, expression-bodied methods, and others.
- C# 7.0 through 7.3 added further enhancements to aid productivity and remove some of the minor anachronisms of C#. For example, these versions enabled you to implement property accessors as expression-bodied members, methods can return multiple values in the form of tuples, the use of `out` parameters was simplified, and `switch` statements were extended to support pattern- and type-matching. These versions of the language also included many other smaller tweaks to address concerns that many developers had, such as allowing the `Main` method to be asynchronous.
- C# 8.0, C# 9.0, and C# 10.0 continue this theme of enhancing the language to improve readability and aid developer productivity. Some major additions included records, which you can use to build immutable reference types; extensions to pattern matching, enabling you to use this feature throughout the language and not just in `switch` statements; top-level statements, which enable you to use C# as a scripting language (you don't always need to write a `Main` method); default interface methods; static local functions; asynchronous disposable types; and many other features, which are covered in this book.

It goes without saying that Microsoft Windows is an important platform for running C# applications, but now you can also run code developed by using C# on other operating systems, such as Linux, through the .NET runtime. This opens up possibilities for writing code that can run in multiple environments. Additionally, Windows supports highly interactive applications that can share data and collaborate as well as connect to services running in the cloud. The key notion in Windows is Universal Windows Platform (UWP) apps—applications designed to run on any Windows 10 or Windows 11 device, whether a full-fledged desktop system, a laptop, a tablet, or even an Internet of Things (IoT) device with limited resources. Once you've mastered the core features of C#, gaining the skills to build applications that can run on all these platforms is critical.

The cloud has become such an important element in the architecture of many systems—ranging from large-scale enterprise applications to mobile apps running on portable devices—that I decided to focus on this aspect of development in the final chapter of the book.

The development environment provided by Visual Studio makes these features easy to use, and the many new wizards and enhancements included in the latest version of Visual Studio can greatly improve your productivity as a developer. I hope you have as much fun working through this book as I had writing it!

## Who should read this book

---

This book assumes that you are a developer who wants to learn the fundamentals of programming with C# by using Visual Studio and the .NET version 6 or later. By the time you complete this book, you will have a thorough understanding of C# and will have used it to build responsive and scalable applications that can run on the Windows operating system.

## Who should not read this book

---

This book is aimed at developers new to C# but not completely new to programming. As such, it concentrates primarily on the C# language. This book is not intended to provide detailed coverage of the multitude of technologies available for building enterprise-level and global applications for Windows, such as ADO.NET, ASP.NET, Azure, or Windows Communication Foundation. If you require more information on any of these items, you might consider reading some of the other titles available from Microsoft Press.

## Finding your best starting point in this book

---

This book is designed to help you build skills in several essential areas. You can use this book if you're new to programming or if you're switching from another programming language such as C, C++, Java, or Visual Basic. Use the following table to find your best starting point.

If you are	Follow these steps
New to object-oriented programming	<ol style="list-style-type: none"> <li>1. Install the practice files as described in the upcoming section, "Code samples."</li> <li>2. Work through Chapters 1 to 22 sequentially.</li> <li>3. Complete Chapters 23 to 27 as your level of experience and interest dictates.</li> </ol>
Familiar with procedural programming languages, such as C, but new to C#	<ol style="list-style-type: none"> <li>1. Install the practice files as described in the upcoming section, "Code samples."</li> <li>2. Skim the first five chapters to get an overview of C# and Visual Studio 2022, and then concentrate on Chapters 6 through 22.</li> <li>3. Complete Chapters 23 to 27 as your level of experience and interest dictates.</li> </ol>
Migrating from an object-oriented language such as C++ or Java	<ol style="list-style-type: none"> <li>1. Install the practice files as described in the upcoming section, "Code samples."</li> <li>2. Skim the first seven chapters to get an overview of C# and Visual Studio 2022, and then concentrate on Chapters 8 through 22.</li> <li>3. For information about building Universal Windows Platform applications, read Chapters 23 to 27.</li> </ol>
Switching from Visual Basic to C#	<ol style="list-style-type: none"> <li>1. Install the practice files as described in the upcoming section, "Code samples."</li> <li>2. Work through Chapters 1 to 22 sequentially.</li> <li>3. For information about building Universal Windows Platform applications, read Chapters 23 to 27.</li> <li>4. Read the "Quick reference" sections at the end of the chapters for information about specific C# and Visual Studio 2022 constructs.</li> </ol>
Referencing the book after working through the exercises	<ol style="list-style-type: none"> <li>1. Use the index or the table of contents to find information about particular subjects.</li> <li>2. Read the "Quick reference" sections at the end of each chapter to find a brief review of the syntax and techniques presented in the chapter.</li> </ol>

Most of the book's chapters include hands-on samples that let you try out the concepts you just learned. No matter which sections you choose to focus on, be sure to download and install the sample applications on your system.

## Conventions and features in this book

This book presents information by using conventions designed to make the information readable and easy to follow.

- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.
- Boxed elements with labels such as "Note," "Tip," "Important," and "More Info" provide additional information or alternative methods for completing a step successfully.

- Text that you type (apart from code blocks) and screen elements you select appear in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, “Press Alt+Tab” means that you hold down the Alt key while you press the Tab key.

## System requirements

---

You will need the following hardware and software to complete the practice exercises in this book:

- Windows 10 (Home, Professional, Education, or Enterprise) or Windows 11 (Home, Professional, Education, or Enterprise).
- The most recent build of Visual Studio Community 2022, Visual Studio Professional 2022, or Visual Studio Enterprise 2022. (Make sure that you have installed any updates.) As a minimum, you should select the following workloads when installing Visual Studio 2022:
  - Universal Windows Platform development
  - .NET desktop development
  - ASP.NET and web development
  - Azure development
  - Data storage and processing
  - .NET Core cross-platform development



**Note** All the exercises and code samples in this book have been developed and tested using Visual Studio Community 2022. They should all work, unchanged, in Visual Studio Professional 2022 and Visual Studio Enterprise 2022.

- 1.8 GHz or faster 64-bit processor; quad-core or better recommended. ARM processors are not supported.
- 4 GB of RAM.
- Hard disk space: minimum of 850 MB up to 210 GB of available space, depending on features installed; typical installations require 20 to 50 GB of free space.

- Video card that supports a minimum display resolution of 720p (1280 by 720); Visual Studio will work best at a resolution of WXGA (1366 by 768) or higher.
- Internet connection to download software or chapter examples.

Depending on your Windows configuration, you might require local administrator rights to install or configure Visual Studio.

You also need to enable developer mode on your computer to be able to create and run UWP apps. For details on how to do this, see “Enable Your Device for Development,” at <https://msdn.microsoft.com/library/windows/apps/dn706236.aspx>.

## Code samples

---

Most of the chapters in this book include exercises with which you can interactively try out new material learned in the main text. You can download all the sample projects, in both their pre-exercise and post-exercise formats, from the following page:

*[MicrosoftPressStore.com/VisualCsharp10e/downloads](https://MicrosoftPressStore.com/VisualCsharp10e/downloads)*

## Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book:

1. Unzip the CSharpSBS.zip file that you downloaded from the book’s website, extracting the files into your Documents folder.
2. If prompted, review the end-user license agreement. If you accept the terms, select the Accept option and then click Next.

**Note** If the license agreement doesn’t appear, you can access it from the same webpage where you downloaded the CSharpSBS.zip file.

## Using the code samples

Each chapter in this book explains when and how to use the code samples for that chapter. When it’s time to use a code sample, the book will list the instructions for how to open the files.

**Important** Many of the code samples depend on NuGet packages that are not included with the code. These packages are downloaded automatically the first time you build a project. As a result, if you open a project and examine the code before doing a build, Visual Studio might report a large number of errors for unresolved references. Building the project will resolve these references, and the errors should disappear.

If you'd like to know all the details, here's a list of the sample Visual Studio projects and solutions, grouped by the folders in which you can find them. In many cases, the exercises provide starter files and completed versions of the same projects that you can use as a reference. The completed projects for each chapter are stored in folders with the suffix "- Complete."

Project/Solution	Description
<b>Chapter 1</b>	
HelloWorld	This project gets you started. It steps through the creation of a simple program using a text editor. The program displays a text-based greeting.
HelloWorld2	This project demonstrates how to use the .NET Command Level Interface (CLI) to build and run a simple C# application.
TestHello	This is a Visual Studio project that displays a greeting.
HelloUWP	This project opens a window that prompts the user for his or her name and then displays a greeting.
<b>Chapter 2</b>	
PrimitiveDataTypes	This project demonstrates how to declare variables by using each of the primitive types, how to assign values to these variables, and how to display their values in a window.
MathOperators	This program introduces the arithmetic operators (+ - * / %).
<b>Chapter 3</b>	
Methods	In this project, you'll reexamine the code in the MathOperators project and investigate how it uses methods to structure the code.
DailyRate	This project walks you through writing your own methods, running the methods, and stepping through the method calls by using the Visual Studio 2015 debugger.
DailyRate Using Optional Parameters	This project shows you how to define a method that takes optional parameters and call the method by using named arguments.
Factorial	This project demonstrates a recursive method that calculates the factorial of a number.



<b>Project/Solution</b>	<b>Description</b>
<b>Chapter 4</b>	
Selection	This project shows you how to use a cascading if statement to implement complex logic, such as comparing the equivalence of two dates.
SwitchStatement	This simple program uses a switch statement to convert characters into their XML representations.
SwitchStatement using Pattern Matching	This is an amended version of the SwitchStatement project that uses pattern matching to simplify the logic in the switch statement.
<b>Chapter 5</b>	
WhileStatement	This project demonstrates a while statement that reads the contents of a source file one line at a time and displays each line in a text box on a form.
DoStatement	This project uses a do statement to convert a decimal number to its octal representation.
<b>Chapter 6</b>	
MathOperators	This project revisits the MathOperators project from Chapter 2 and shows how various unhandled exceptions can make the program fail. The try and catch keywords then make the application more robust so that it no longer fails.
<b>Chapter 7</b>	
Classes	This project covers the basics of defining your own classes, complete with public constructors, methods, and private fields. It also shows how to create class instances by using the new keyword and how to define static methods and fields.
<b>Chapter 8</b>	
Parameters	This program investigates the difference between value parameters and reference parameters. It demonstrates how to use the ref and out keywords.
<b>Chapter 9</b>	
StructsAndEnums	This project defines a struct type to represent a calendar date.
<b>Chapter 10</b>	
Cards	This project shows how to use arrays to model hands of cards in a card game.
<b>Chapter 11</b>	
ParamsArray	This project demonstrates how to use the params keyword to create a single method that can accept any number of int arguments.
<b>Chapter 12</b>	
Vehicles	This project creates a simple hierarchy of vehicle classes by using inheritance. It also demonstrates how to define a virtual method.
ExtensionMethod	This project shows how to create an extension method for the int type, providing a method that converts an integer value from base 10 to a different number base.

<b>Project/Solution</b>	<b>Description</b>
<b>Chapter 13</b>	
Drawing	This project implements part of a graphical drawing package. The project uses interfaces to define the methods that drawing shapes expose and implement.
<b>Chapter 14</b>	
GarbageCollectionDemo	This project shows how to implement exception-safe disposal of resources by using the Dispose pattern.
<b>Chapter 15</b>	
Drawing Using Properties	This project extends the application in the Drawing project developed in Chapter 13 to encapsulate data in a class by using properties.
AutomaticProperties	This project shows how to create automatic properties for a class and use them to initialize instances of the class.
Student enrollment	This project demonstrates how to use records to model structured immutable types.
<b>Chapter 16</b>	
Indexers	This project uses two indexers: one to look up a person's phone number when given a name and the other to look up a person's name when given a phone number.
<b>Chapter 17</b>	
BinaryTree	This solution shows you how to use generics to build a type-safe structure that can contain elements of any type.
BuildTree	This project demonstrates how to use generics to implement a type-safe method that can take parameters of any type.
<b>Chapter 18</b>	
Cards	This project updates the code from Chapter 10 to show how to use collections to model hands of cards in a card game.
<b>Chapter 19</b>	
BinaryTree	This project shows you how to implement the generic <code>IEnumerator&lt;T&gt;</code> interface to create an enumerator for the generic <code>Tree</code> class.
IteratorBinaryTree	This solution uses an iterator to generate an enumerator for the generic <code>Tree</code> class.
<b>Chapter 20</b>	
Delegates	This project shows how to decouple a method from the application logic that invokes it by using a delegate. The project is then extended to show how to use an event to alert an object to a significant occurrence, and how to catch an event and perform any processing required.
<b>Chapter 21</b>	
QueryBinaryTree	This project shows how to use LINQ queries to retrieve data from a binary tree object.

<b>Project/Solution</b>	<b>Description</b>
<b>Chapter 22</b>	
ComplexNumbers	This project defines a new type that models complex numbers and implements common operators for this type.
<b>Chapter 23</b>	
GraphDemo	This project generates and displays a complex graph on a UWP form. It uses a single thread to perform the calculations.
Parallel GraphDemo	This version of the GraphDemo project uses the Parallel class to abstract out the process of creating and managing tasks.
GraphDemo with Cancellation	This project shows how to implement cancellation to halt tasks in a controlled manner before they have completed.
ParallelLoop	This application provides an example showing when you should not use the Parallel class to create and run tasks.
<b>Chapter 24</b>	
GraphDemo	This is a version of the GraphDemo project from Chapter 23 that uses the async keyword and the await operator to perform the calculations that generate the graph data asynchronously.
PLINQ	This project shows some examples of using PLINQ to query data by using parallel tasks.
CalculatePI	This project uses a statistical sampling algorithm to calculate an approximation for pi. It uses parallel tasks.
ParallelTest	This program illustrates the dangers of allowing uncontrolled data access to shared data by parallel threads.
<b>Chapter 25</b>	
Customers	This project implements a scalable user interface that can adapt to different device layouts and form factors. The user interface applies XAML styling to change the fonts and background image displayed by the application.
<b>Chapter 26</b>	
DataBinding	This is a version of the Customers project that uses data binding to display customer information retrieved from a data source in the user interface. It also shows how to implement the INotifyPropertyChanged interface so that the user interface can update customer information and send these changes back to the data source.
ViewModel	This version of the Customers project separates the user interface from the logic that accesses the data source by implementing the Model-View-ViewModel pattern.

Project/Solution	Description
<b>Chapter 27</b>	
Web Service	This solution includes a web application that provides a REST web service that the Customers application uses to retrieve customer information and modify data held in a SQL Server database. The web service uses the Entity Framework to access the database. The database and the web service run using Azure.
Customers with insert and update features	This solution contains an updated version of the Customers project that uses the REST web service to create new customers and modify the details of existing customers.

## Errata and book support

---

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at:

*[MicrosoftPressStore.com/VisualCsharp10e/errata](http://MicrosoftPressStore.com/VisualCsharp10e/errata)*

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit:

*[MicrosoftPressStore.com/Support](http://MicrosoftPressStore.com/Support)*

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to:

*<http://support.microsoft.com>*

## Stay in touch

---

Let's keep the conversation going! We're on Twitter: *<http://twitter.com/MicrosoftPress>*.



# Understanding values and references

## After completing this chapter, you will be able to:

- Explain the differences between a value type and a reference type.
- Understand `null` values and how nullable types work.
- Modify how arguments are passed as method parameters by using the `ref` and `out` keywords.
- Describe how computer memory is organized to support value types and reference types.
- Convert a value into a reference by using boxing.
- Convert a reference back to a value by using unboxing and casting.

Chapter 7, “Creating and managing classes and objects,” demonstrated how to declare your own classes and how to create objects by using the `new` keyword. That chapter also showed you how to initialize an object by using a constructor. In this chapter, you’ll learn how the characteristics of the primitive types such as `int`, `double`, and `char` differ from the characteristics of class types.

## Copying value type variables and classes

---

Most of the primitive types built into C#, such as `int`, `float`, `double`, and `char` (but not `string`, for reasons that will be covered shortly), are collectively called *value types*. These types have a fixed size, and when you declare a variable as a value type, the compiler generates code that allocates a block of memory big enough to hold a corresponding value. For example, declaring an `int` variable causes the compiler to allocate 4 bytes of memory (32 bits) to hold the integer value. A statement that assigns a value (such as 42) to the `int` causes the value to be copied into this block of memory.

Class types such as `Circle` (described in Chapter 7) are handled differently. When you declare a `Circle` variable, the compiler *does not* generate code that allocates a block of memory big enough to hold a `Circle` object. All it does is allot a small piece of memory that can potentially hold the address of (or a reference to) another block of memory containing a `Circle` object. (An address specifies the location of an item in memory.) The memory for the actual `Circle` object is allocated only when the `new` keyword is used to create the object.

A class is an example of a *reference type*. Reference types hold references to blocks of memory. To write effective C# code, you must understand the difference between value types and reference types.



**Note** The `string` type in C# is actually a class. This is because there is no standard size for a string (different strings can contain different numbers of characters), and allocating memory for a string dynamically when the program runs is far more efficient than doing so statically at compile time. The description in this chapter of reference types such as classes applies to the `string` type as well. In fact, the `string` keyword in C# is just an alias for the `System.String` class.

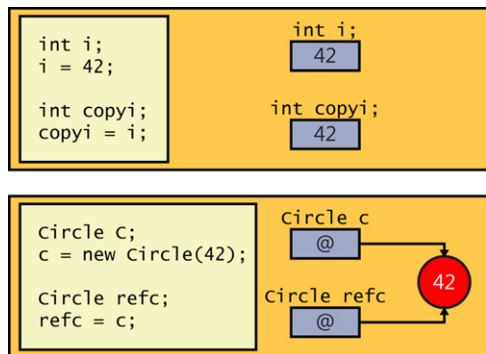
Consider a situation in which you declare a variable named `i` as an `int` and assign it the value 42. If you declare another variable called `copyi` as an `int` and then assign `i` to `copyi`, `copyi` will hold the same value as `i` (42). However, even though `copyi` and `i` happen to hold the same value, two blocks of memory contain the value 42: one block for `i` and the other block for `copyi`. If you modify the value of `i`, the value of `copyi` does not change. Let's see this in code:

```
int i = 42; // declare and initialize i
int copyi = i; /* copyi contains a copy of the data in i:
                i and copyi both contain the value 42 */
i++;          /* incrementing i has no effect on copyi;
                i now contains 43, but copyi still contains 42 */
```

The effect of declaring a variable `c` as a class type, such as `Circle`, is very different. When you declare `c` as a `Circle`, `c` can refer to a `Circle` object; the actual value held by `c` is the address of a `Circle` object in memory. If you declare an additional variable named `refc` (also as a `Circle` object) and you assign `c` to `refc`, `refc` will have a copy of the same address as `c`. In other words, there's only one `Circle` object, and both `refc` and `c` now refer to it. Here's an example in code:

```
var c = new Circle(42);
Circle refc = c;
```

The following illustration shows both examples. The at sign (@) in the `Circle` objects represents a reference holding an address in memory.



This difference is very important. It means that the behavior of method parameters depends on whether they are value types or reference types. You'll explore this difference in the next exercise.

## Copying reference types and data privacy

If you actually want to copy the contents of a `Circle` object, `c`, into a different `Circle` object, `refc`, instead of just copying the reference, you must make `refc` refer to a new instance of the `Circle` class and then copy the data, field by field, from `c` into `refc`, like this:

```
var refc = new Circle();
refc.radius = c.radius; // Don't try this
```

However, if any members of the `Circle` class are private (like the `radius` field), you won't be able to copy this data. Instead, you can make the data in the private fields accessible by exposing them as properties and then use these properties to read the data from `c` and copy it into `refc`. You'll learn how to do this in Chapter 15, "Implementing properties to access fields."

Alternatively, a class could provide a `Clone` method that returns another instance of the same class but populated with the same data. The `Clone` method would have access to the private data in an object and could copy this data directly to another instance of the same class. For example, the `Clone` method for the `Circle` class could be defined as shown here:

```
class Circle
{
    private int radius;
    // Constructors and other methods omitted
    ...
    public Circle Clone()
    {
        // Create a new Circle object
        Circle clone = new Circle();

        // Copy private data from this to clone
        clone.radius = this.radius;

        // Return the new Circle object containing the copied data
        return clone;
    }
}
```

This approach is straightforward if all the private data consists of values, but if one or more fields are themselves reference types (for example, if the `Circle` class is extended to contain a `Point` object from Chapter 7, indicating the position of the `Circle` on a graph), these reference types also need to provide a `Clone` method; otherwise, the `Clone` method of the `Circle` class will simply copy a reference to these fields. This process is known as a *deep copy*. The alternative approach, wherein the `Clone` method simply copies references, is known as a *shallow copy*.

The preceding code example also poses an interesting question: How private is private data? Previously, you saw that the `private` keyword renders a field or method inaccessible from outside a class. However, this does not mean it can be accessed by only a single object. If you create two objects of the same class, they can each access the private data of the other within the code for that class.



This sounds curious, but in fact, methods such as `Clone` depend on this feature. For example, the statement `clone.radius = this.radius;` works only because the private `radius` field in the `clone` object is accessible from within the current instance of the `Circle` class. So, private actually means private to the class rather than private to an object. Don't confuse private with static, however. If you simply declare a field as private, each instance of the class gets its own data. If a field is declared as static, each instance of the class shares the same data.

## To use value parameters and reference parameters

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Parameters** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 8\Parameters** folder in your **Documents** folder.

The project contains three C# code files: `Pass.cs`, `Program.cs`, and `WrappedInt.cs`.

3. Display the `Pass.cs` file in the Code and Text Editor window.

This file defines a class called `Pass` that is currently empty apart from a `// TODO: comment`.



**Tip** You can use the Task List window to locate all `// TODO: comment` in a solution.

4. Add a public static method called `Value` to the `Pass` class, replacing the `// TODO: comment`. This method should accept a single `int` parameter (a value type) called `param` and have the return type `void`. The body of the `Value` method should simply assign the value 42 to `param`, as shown in bold type in the following code example:

```
namespace Parameters
{
    class Pass
    {
        public static void Value(int param)
        {
            param = 42;
        }
    }
}
```



**Note** You're defining this method using the `static` keyword to keep the exercise simple. You can call the `Value` method directly on the `Pass` class without first creating a new `Pass` object. The principles illustrated in this exercise apply in the same manner to instance methods.

5. Display the Program.cs file in the Code and Text Editor window and then locate the doWork method of the Program class.

The doWork method is called by the Main method when the program starts running. As explained in Chapter 7, the method call is wrapped in a try block and followed by a catch handler.

6. Add four statements to the doWork method to perform the following tasks:

- Declare a local int variable called i and initialize it to 0.
- Write the value of i to the console by using Console.WriteLine.
- Call Pass.Value, passing i as an argument.
- Write the value of i to the console again.

By running Console.WriteLine before and after the call to Pass.Value, you can see whether the Pass.Value method actually modifies the value of i. The completed doWork method should look exactly like this:

```
static void doWork()
{
    int i = 0;
    Console.WriteLine(i);
    Pass.Value(i);
    Console.WriteLine(i);
}
```

7. On the **Debug** menu, select **Start Without Debugging** to build and run the program.
8. Confirm that the value 0 is written to the console window twice.

The assignment statement inside the Pass.Value method that updates the parameter and sets it to 42 uses a copy of the argument passed in, and the original argument i is completely unaffected.

9. Press **Enter** to close the application.

You'll now see what happens when you pass an int parameter that's wrapped within a class.

10. Display the WrappedInt.cs file in the Code and Text Editor window. This file contains the WrappedInt class, which is empty apart from a // TODO: comment.
11. Add a public instance field called Number of type int to the WrappedInt class, as shown in bold type in the following code:

```
namespace Parameters
{
    class WrappedInt
    {
        public int Number;
    }
}
```

12. Display the `Pass.cs` file in the Code and Text Editor window. Add a public static method called `Reference` to the `Pass` class. This method should accept a single `WrappedInt` parameter called `param` and have the return type `void`. The body of the `Reference` method should assign 42 to `param.Number`, as shown here:

```
public static void Reference(WrappedInt param)
{
    param.Number = 42;
}
```

13. Display the `Program.cs` file in the Code and Text Editor window. Comment out the existing code in the `doWork` method and add four more statements to perform the following tasks:

- Declare a local `WrappedInt` variable called `wi` and initialize it to a new `WrappedInt` object by calling the default constructor.
- Write the value of `wi.Number` to the console.
- Call the `Pass.Reference` method, passing `wi` as an argument.
- Write the value of `wi.Number` to the console again.

As before, with the calls to `Console.WriteLine`, you can see whether the call to `Pass.Reference` modifies the value of `wi.Number`. The `doWork` method should now look exactly like this (the new statements are in bold):

```
static void doWork()
{
    // int i = 0;
    // Console.WriteLine(i);
    // Pass.Value(i);
    // Console.WriteLine(i);
    var wi = new WrappedInt();
    Console.WriteLine(wi.Number);
    Pass.Reference(wi);
    Console.WriteLine(wi.Number);
}
```

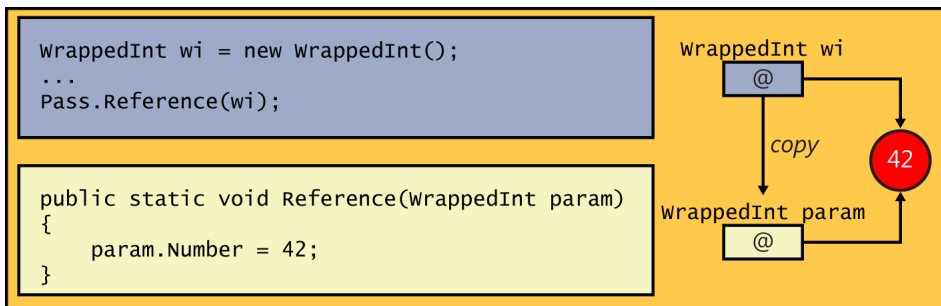
14. On the **Debug** menu, select **Start Without Debugging** to build and run the application.

This time, the two values displayed in the console window correspond to the value of `wi.Number` before and after the call to the `Pass.Reference` method. You should see that the values 0 and 42 are displayed.

15. Press **Enter** to close the application and return to Visual Studio 2022.

To explain what the previous exercise shows, the value of `wi.Number` is initialized to 0 by the compiler-generated default constructor. The `wi` variable contains a reference to the newly created `WrappedInt` object (which contains an `int`). The `wi` variable is then copied as an argument to the `Pass.Reference` method. Because `WrappedInt` is a class (a reference type), `wi` and `param` both refer to the same `WrappedInt` object. Any changes made to the contents of the object through the `param` variable in the `Pass.Reference` method are visible by using the `wi` variable when the method

completes. The following diagram illustrates what happens when a `WrappedInt` object is passed as an argument to the `Pass.Reference` method.



## Understanding null values and nullable types

When you declare a variable, it's always a good idea to initialize it. With value types, it's common to see code such as this:

```
int i = 0;
double d = 0.0;
```

To initialize a reference variable such as a class, you can create a new instance of the class and assign the reference variable to the new object, like this:

```
var c = new Circle(42);
```

This is all very well, but what if you don't actually want to create a new object? Perhaps the purpose of the variable is simply to store a reference to an existing object at some later point in your program. In the following code example, the `Circle` variable `copy` is initialized, but later it is assigned a reference to another instance of the `Circle` class:

```
var c = new Circle(42);
var copy = new Circle(99); // Some random value for initializing copy
...
copy = c; // copy and c refer to the same object
```

After assigning `c` to `copy`, what happens to the original `Circle` object with a radius of 99 that you used to initialize `copy`? Nothing refers to it anymore. In this situation, the runtime can reclaim the memory by performing an operation known as *garbage collection*, which you'll learn more about in Chapter 14, "Using garbage collection and resource management." The important thing to understand for now is that garbage collection is a potentially time-consuming operation, and that you should not create objects that are never used because doing so is a waste of time and resources.

You could argue that if a variable will be assigned a reference to another object at some point in a program, there's no point to initializing it. This is poor programming practice, however, and can lead to problems in your code. For example, you will inevitably find yourself in a situation in which you want to

refer a variable to an object only if that variable does not already contain a reference, as shown in the following code example:

```
var c = new Circle(42);
Circle copy;           // Uninitialized !!!
...
if (copy == // only assign to copy if it is uninitialized, but what goes here?)
{
    copy = c; ;        // copy and c refer to the same object
    ...
}
```

The purpose of the `if` statement is to test the `copy` variable to see whether it is initialized, but to which value should you compare this variable? The answer is to use a special value called `null`.

In C#, you can assign the `null` value to any reference variable. The `null` value simply means that the variable does not refer to an object in memory. You can use it like this:

```
Circle c = new Circle(42);
Circle copy = null; // Initialized
...
if (copy is null)
{
    copy = c; // copy and c refer to the same object
    ...
}
```



**Note** You can also use `== null` to check for a null reference. However, `is null` reads more naturally. Similarly, you can use `is not null` as well as `!= null` to check for a non-null reference.

## The null-conditional and null-coalescing operators

The null-conditional operator enables you to test for `null` values very succinctly. To use the null-conditional operator, you append a question mark (?) to the name of your variable.

For example, suppose you attempt to call the `Area` method on a `Circle` object when the `Circle` object has a `null` value:

```
Circle c = null;
Console.WriteLine($"The area of circle c is {c.Area()}");
```

In this case, the `Circle.Area` method throws a `NullReferenceException`, which makes sense because you cannot calculate the area of a circle that does not exist.

To avoid this exception, you could test whether the `Circle` object is `null` before you attempt to call the `Circle.Area` method:

```
if (c is not null)
{
    Console.WriteLine($"The area of circle c is {c.Area()}");
}
```

In this case, if `c` is `null`, nothing is written to the command window. Alternatively, you could use the null-conditional operator on the `Circle` object before you attempt to call the `Circle.Area` method:

```
Console.WriteLine($"The area of circle c is {c?.Area()}");
```

The null-conditional operator tells the C# runtime to ignore the current statement if the variable you have applied the operator to is `null`. In this case, the command window would display the following text:

```
The area of circle c is
```

Both approaches are valid and might meet your needs in different scenarios. The null-conditional operator can help you keep your code concise, particularly when you deal with complex properties with nested reference types that could all be `null` valued.

Alongside the null-conditional operator, C# provides two null-coalescing operators. The first of these, `??`, is a binary operator that returns the value of the operand on the left if it isn't `null`; otherwise, it returns the value of the operand on the right. In the following example, variable `c2` is assigned a reference to `c` if `c` isn't `null`; otherwise, it is assigned a reference to a new `Circle` object:

```
Circle c = ...; // might be null, might be a new Circle object
...
var c2 = c ?? new Circle(42);
```

The null-coalescing assignment operator, `??=`, assigns the value of the operand on the right to the operand on the left only if the left operand is `null`. If the left operand references some other value, it is unchanged.

```
Circle c = ...; // might be null, might be a new Circle object
Circle c3 = ...; // might be null, might be a new Circle object
...
var c3 ??= c; // Only assign c3 if it is null, otherwise leave unchanged;
```

## Using nullable types

The `null` value is very useful for initializing reference types. Sometimes, though, you need an equivalent value for value types. `null` is itself a reference, so you cannot assign it to a value type. The following statement is therefore illegal in C#:

```
int i = null; // illegal
```

However, C# defines a modifier that you can use to declare that a variable is a *nullable* value type. A nullable value type behaves similarly to the original value type, but you can assign the `null` value to it. You use the question mark (`?`) to indicate that a value type is nullable, like this:

```
int? i = null; // legal
```

You can ascertain whether a nullable variable contains `null` by testing it in the same way as you test a reference type.

```
if (i is null)
    ...
```

You can assign an expression of the appropriate value type directly to a nullable variable. The following examples are all legal:

```
int? i = null;
int j = 99;
i = 100; // Copy a value type constant to a nullable type
i = j; // Copy a value type variable to a nullable type
```

You should note that the converse is not true. You cannot assign a nullable variable to an ordinary value type variable. So, given the definitions of variables `i` and `j` from the preceding example, the following statement is not allowed:

```
j = i; // illegal
```

This makes sense when you consider that the variable `i` might contain `null`, and `j` is a value type that cannot contain `null`. This also means that you cannot use a nullable variable as a parameter to a method that expects an ordinary value type. If you recall, the `Pass.Value` method from the preceding exercise expects an ordinary `int` parameter, so the following method call will not compile:

```
int? i = 99;
Pass.Value(i); // Compiler error
```



**Note** Take care not to confuse nullable types with the null-conditional operator. Nullable types are indicated by appending a question mark to the type name, whereas the null-conditional operator is appended to the variable name.

## Understanding the properties of nullable types

A nullable type exposes a pair of properties that you can use to determine whether the type actually has a non-null value and what this value is. The `HasValue` property indicates whether a nullable type contains a value or is `null`. You can retrieve the value of a non-null nullable type by reading the `Value` property, like this:

```
int? i = null;
...
if (!i.HasValue)
{
    // If i is null, then assign it the value 99
    i = 99;
}
else
{
    // If i is not null, then display its value
    Console.WriteLine(i.Value);
}
```

In Chapter 4, “Using decision statements,” you saw that the NOT operator (!) negates a Boolean value. The preceding code fragment tests the nullable variable `i`, and if it does not have a value (it is `null`), it assigns it the value 99; otherwise, it displays the value of the variable. In this example, using the `HasValue` property does not provide any benefit over testing for a `null` value directly. Additionally, reading the `Value` property is a long-winded way of reading the contents of the variable. However, these apparent shortcomings are caused by the fact that `int?` is a very simple nullable type. You can create more complex value types and use them to declare nullable variables where the advantages of using the `HasValue` and `Value` properties become more apparent. You’ll see some examples in Chapter 9, “Creating value types with enumerations and structures.”



**Note** The `Value` property of a nullable type is read-only. You can use this property to read the value of a variable but not to modify it. To update a nullable variable, use an ordinary assignment statement.

## Using ref and out parameters

---

Ordinarily, when you pass an argument to a method, the corresponding parameter is initialized with a copy of the argument. This is true regardless of whether the parameter is a value type (such as an `int`), a nullable type (such as `int?`), or a reference type (such as a `WrappedInt`). This arrangement means that it’s impossible for any change to the parameter to affect the value of the argument passed in. For example, in the following code, the value output to the console is 42, not 43. The `doIncrement` method increments a copy of the argument (`arg`) and *not* the original argument, as demonstrated here:

```
static void doIncrement(int param)
{
    param++;
}

static void Main()
{
    int arg = 42;
    doIncrement(arg);
    Console.WriteLine(arg); // writes 42, not 43
}
```

In the preceding exercise, you saw that if the parameter to a method is a reference type, any changes made by using that parameter change the data referenced by the argument passed in. The key point is this: although the data that was referenced changed, the argument passed in as the parameter did not. It still references the same object. In other words, although it’s possible to modify the object that the argument refers to through the parameter, it’s not possible to modify the argument itself—for example, to set it to refer to a completely different object. Most of the time, this guarantee is very useful and can help reduce the number of bugs in a program. Occasionally, however, you might want to write a method that actually needs to modify an argument. C# provides the `ref` and `out` keywords so that you can do this.



## Creating ref parameters

If you prefix a parameter with the `ref` keyword, the C# compiler generates code that passes a reference to the actual argument rather than a copy of the argument. When using a `ref` parameter, anything you do to the parameter you also do to the original argument because the parameter and the argument both reference the same data.

When you pass an argument as a `ref` parameter, you must also prefix the argument with the `ref` keyword. This syntax provides a useful visual cue to the programmer that the argument might change. Here's the preceding example again, this time modified to use the `ref` keyword:

```
static void doIncrement(ref int param) // using ref
{
    param++;
}

static void Main()
{
    int arg = 42;
    doIncrement(ref arg); // using ref
    Console.WriteLine(arg); // writes 43
}
```

This time, the `doIncrement` method receives a reference to the original argument rather than a copy, so any changes the method makes by using this reference actually change the original value. That's why the value 43 is displayed on the console.

Remember that C# enforces the rule that you must assign a value to a variable before you can read it. This rule also applies to method arguments; you cannot pass an uninitialized value as an argument to a method even if an argument is defined as a `ref` argument. For example, in the following example, `arg` is not initialized, so this code will not compile. This failure occurs because the statement `param++`; within the `doIncrement` method is really an alias for the statement `arg++`; and this operation is allowed only if `arg` has a defined value:

```
static void doIncrement(ref int param)
{
    param++;
}

static void Main()
{
    int arg; // not initialized
    doIncrement(ref arg);
    Console.WriteLine(arg);
}
```

## Creating out parameters

The compiler checks whether a `ref` parameter has been assigned a value before calling the method. However, there might be times when you want the method itself to initialize the parameter. You can do this with the `out` keyword.

The `out` keyword is syntactically similar to the `ref` keyword. You can prefix a parameter with the `out` keyword so that the parameter becomes an alias for the argument. As when using `ref`, anything you do to the parameter, you also do to the original argument. When you pass an argument to an `out` parameter, you must also prefix the argument with the `out` keyword.

The keyword `out` is short for *output*. When you pass an `out` parameter to a method, the method *must* assign a value to it before it finishes or returns, as shown in the following example:

```
static void doInitialize(out int param)
{
    param = 42; // Initialize param before finishing
}
```

The following example does not compile because `doInitialize` does not assign a value to `param`:

```
static void doInitialize(out int param)
{
    // Do nothing
}
```

Because an `out` parameter must be assigned a value by the method, you're allowed to call the method without initializing its argument. For example, the following code calls `doInitialize` to initialize the variable `arg`, which is then displayed on the console:

```
static void doInitialize(out int param)
{
    param = 42;
}

static void Main()
{
    int arg; // not initialized
    doInitialize(out arg); // legal
    Console.WriteLine(arg); // writes 42
}
```



**Note** You can combine the declaration of an `out` variable with its use as a parameter rather than performing these tasks separately. For example, you could replace the first two statements in the `Main` method in the previous example with this single line of code:

```
doInitialize(out int arg);
```

In the next exercise, you'll practice using `ref` parameters.

### To use `ref` parameters

1. Return to the Parameters project in Visual Studio 2022.
2. Display the `Pass.cs` file in the Code and Text Editor window.

3. Edit the `Value` method to accept its parameter as a `ref` parameter.

The `Value` method should look like this:

```
class Pass
{
    public static void Value(ref int param)
    {
        param = 42;
    }
    ...
}
```

4. Display the `Program.cs` file in the Code and Text Editor window.
5. Uncomment the first four statements.

Notice that the third statement of the `doWork` method, `Pass.Value(i)`, indicates an error. The error occurs because the `Value` method now expects a `ref` parameter.

6. Edit this statement so that the `Pass.Value` method call passes its argument as a `ref` parameter.



**Note** Leave the four statements that create and test the `WrappedInt` object as they are.

The `doWork` method should now look like this:

```
class Program
{
    static void doWork()
    {
        int i = 0;
        Console.WriteLine(i);
        Pass.Value(ref i);
        Console.WriteLine(i);
        ...
    }
}
```

7. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

This time, the first two values written to the console window are 0 and 42. This result shows that the call to the `Pass.Value` method has successfully modified the argument `i`.

8. Press **Enter** to close the application and return to Visual Studio 2022.



**Note** You can use the `ref` and `out` modifiers on reference type parameters as well as on value type parameters. The effect is the same: the parameter becomes an alias for the argument.

# How computer memory is organized

Computers use memory to hold programs that are being executed and the data that those programs use. To understand the differences between value and reference types, it's helpful to understand how data is organized in memory.

Operating systems and language runtimes such as those used by C# frequently divide the memory used for holding data into two separate areas, each of which is managed in a distinct manner. These two areas of memory are traditionally called the *stack* and the *heap*. The stack and the heap serve different purposes:

- When you call a method, the memory required for its parameters and its local variables is acquired from the stack. When the method finishes (because it either returns or throws an exception), the memory acquired for the parameters and local variables is automatically released back to the stack to be made available again when another method is called. Method parameters and local variables on the stack have a well-defined lifespan: They come into existence when the method starts, and they disappear as soon as the method completes.

The same lifespan applies to variables defined in any block of code enclosed by opening and closing braces. In the following code example, the variable `i` is created when the body of the `while` loop starts, but it disappears when the `while` loop finishes, and execution continues after the closing brace:

```
while (...)  
{  
    int i = ...; // i is created on the stack here  
    ...  
}  
// i disappears from the stack here
```

- When you create an object (an instance of a class) by using the `new` keyword, the memory required to build the object is acquired from the heap. You've seen that the same object can be referenced from several places by using reference variables. When the last reference to an object disappears, the memory used by the object becomes available again (although it might not be reclaimed immediately). Objects created on the heap therefore have a more indeterminate lifespan; an object is created by using the `new` keyword, but it disappears only sometime after the last reference to the object is removed. Chapter 14 includes a more detailed discussion of how heap memory is reclaimed.



**Note** All value types are created on the stack. By default, reference types (objects) are created on the heap, although the reference itself is on the stack. (There are some exceptions to this rule, which you'll learn about in later chapters.) Nullable objects are actually reference types, and they are created on the heap.

The names *stack* and *heap* come from the way in which the runtime manages the memory:

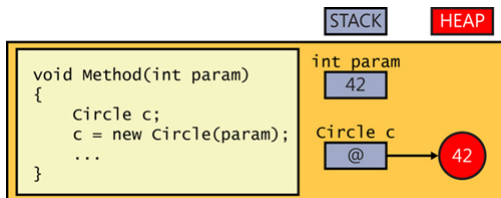
- Stack memory is organized like boxes stacked neatly on top of one another. When a method is called, each parameter is placed in a box that is added to the top of the stack. Each local variable is likewise assigned a box, which is placed on top of the boxes already on the stack. When a method finishes, think of it as being like a box being removed from the stack.
- Heap memory is like a large pile of boxes strewn around a room rather than stacked neatly on top of one another. Each box has a label indicating whether it is in use. When a new object is created, the runtime searches for an empty box and allocates it to the object. The reference to the object is stored in a local variable on the stack. The runtime keeps track of the number of references to each box. (Remember: two variables can refer to the same object.) When the last reference disappears, the runtime marks the box as not in use; at some point in the future, it empties the box and makes it available.

## Using the stack and the heap

Now let's examine what happens when a method named `Method` is called:

```
void Method(int param)
{
    Circle c;
    c = new Circle(param);
    ...
}
```

Suppose the argument passed into `param` is the value 42. When the method is called, a block of memory (just enough for an `int`) is allocated from the stack and initialized with the value 42. As execution moves inside the method, another block of memory big enough to hold a reference (a memory address) is also allocated from the stack, but left uninitialized. This is for the `Circle` variable, `c`. Next, another piece of memory big enough for a `Circle` object is allocated from the heap. This is what the `new` keyword does. The `Circle` constructor runs to convert this raw heap memory to a `Circle` object. A reference to this `Circle` object is stored in the variable `c`. The following illustration shows this process:



At this point, you should note two things:

- Although the object is stored on the heap, the reference to the object (the variable `c`) is stored on the stack.
- Heap memory is not infinite. If heap memory is exhausted, the `new` operator will throw an `OutOfMemoryException` exception, and the object will not be created.



**Note** The `Circle` constructor could also throw an exception. If it does, the memory allocated to the `Circle` object will be reclaimed, and the value returned by the constructor will be `null`.

When the method ends, the parameters and local variables go out of scope. The memory acquired for `c` and `param` is automatically released back to the stack. The runtime notes that the `Circle` object is no longer referenced and at some point in the future will arrange for its memory to be reclaimed by the heap. (See Chapter 14.)

## The `System.Object` class

One of the most important reference types in .NET is the `Object` class in the `System` namespace. To fully appreciate the significance of the `System.Object` class, you must understand inheritance, which is described in Chapter 12, “Working with inheritance.” For now, simply accept that all classes are specialized types of `System.Object` and that you can use `System.Object` to create a variable that can refer to any reference type. `System.Object` is such an important class that C# provides the `object` keyword as an alias for `System.Object`. In your code, you can use `object`, or you can write `System.Object`. They mean the same thing.

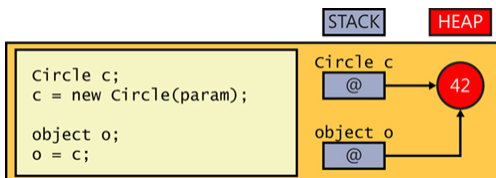


**Tip** Use the `object` keyword rather than `System.Object`. It’s more direct, and it’s consistent with other keywords that are synonyms for classes, such as `string` for `System.String` and others that are covered in Chapter 9.

In the following example, the variables `c` and `o` both refer to the same `Circle` object. The fact that the type of `c` is `Circle` and the type of `o` is `object` (the alias for `System.Object`) in effect provides two different views of the same item in memory.

```
Circle c;  
c = new Circle(42);  
object o;  
o = c;
```

The following diagram illustrates how the variables `c` and `o` refer to the same item on the heap:

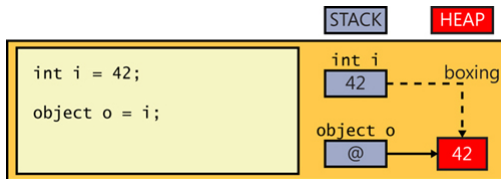


## Boxing

As you have just seen, variables of type `object` can refer to any item of any reference type. However, variables of type `object` can also refer to a value type. For example, the following two statements initialize the variable `i` (of type `int`, a value type) to 42 and then initialize the variable `o` (of type `object`, a reference type) to `i`:

```
int i = 42;
object o = i;
```

The second statement requires a little explanation to appreciate what's actually happening. Remember that `i` is a value type and that it lives on the stack. If the reference inside `o` referred directly to `i`, the reference would refer to the stack. However, references should refer to objects on the heap. Creating uncontrolled references to items on the stack could seriously compromise the robustness of the runtime and potentially create a security flaw, so it is not allowed. Therefore, the runtime allocates a piece of memory from the heap, copies the value of integer `i` to this piece of memory, and then refers the object `o` to this copy. This automatic copying of an item from the stack to the heap is called *boxing*. The following diagram shows the result:



**Important** If you modify the original value of the variable `i`, the value on the heap referenced through `o` will not change. Likewise, if you modify the value on the heap, the original value of the variable will not change.

## Unboxing

Because a variable of type `object` can refer to a boxed copy of a value, it's only reasonable to allow you to get at that boxed value through the variable. You might expect to be able to access the boxed `int` value that a variable `o` refers to by using a simple assignment statement such as this:

```
int i = o;
```

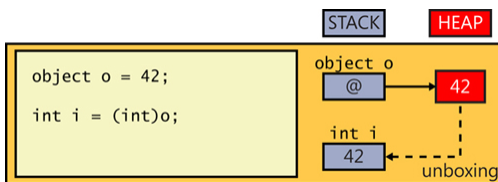
However, if you try this syntax, you'll get a compile-time error. If you think about it, it's fairly sensible that you can't use the `int i = o;` syntax. After all, `o` could be referencing absolutely anything and not just an `int`. Consider what would happen in the following code if this statement were allowed:

```
Circle c = new Circle();
int i = 42;
object o;
o = c; // o refers to a circle
i = o; // what is stored in i?
```

To obtain the value of the boxed copy, you must use what is known as a *cast*. This is an operation that checks whether converting an item of one type to another is safe before actually making the copy. You prefix the object variable with the name of the type in parentheses, as in this example:

```
int i = 42;
object o = i; // boxes
i = (int)o; // compiles okay
```

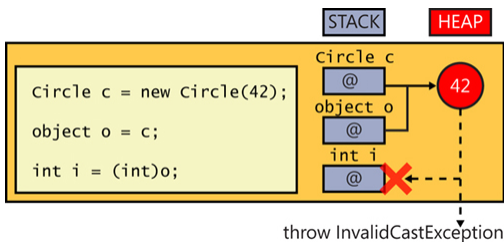
The effect of this cast is subtle. The compiler notices that you've specified the type `int` in the cast. Next, the compiler generates code to check what `o` actually refers to at runtime. It could be absolutely anything. Just because your cast says `o` refers to an `int`, that doesn't mean it actually does. If `o` really does refer to a boxed `int` and everything matches, the cast succeeds, and the compiler-generated code extracts the value from the boxed `int` and copies it to `i`. (In this example, the boxed value is then stored in `i`.) This is called *unboxing*. The following diagram shows what's happening:



On the other hand, if `o` does not refer to a boxed `int`, there is a type mismatch, causing the cast to fail. The compiler-generated code throws an `InvalidCastException` exception at runtime. Here's an example of an unboxing cast that fails:

```
Circle c = new Circle(42);
object o = c; // doesn't box because Circle is a reference variable
int i = (int)o; // compiles okay but throws an exception at runtime
```

The following diagram illustrates this case:



You'll use boxing and unboxing in later exercises. Keep in mind that boxing and unboxing are expensive operations because of the amount of checking required and the need to allocate additional heap memory. Boxing has its uses, but injudicious use can severely impair the performance of a program. You'll see an alternative to boxing in Chapter 17, "Introducing generics."



## Casting data safely

---

By using a cast, you can specify that, in your opinion, the data referenced by an object has a specific type and that it's safe to reference the object by using that type. The key phrase here is “in your opinion.” The C# compiler will not check that this is the case, but the runtime will. If the type of object in memory does not match the cast, the runtime will throw an `InvalidCastException`, as described in the preceding section. You should be prepared to catch this exception and handle it appropriately if it occurs.

However, catching an exception and attempting to recover if the type of an object is not what you expected it to be is a rather cumbersome approach. C# provides two more very useful operators that can help you perform casting in a much more elegant manner: the `is` and `as` operators.

### The `is` operator

You've seen the `is` operator before, when checking for a `null` value, but it actually enables you to check for the type of any reference object. You can use the `is` operator to verify that the type of an object is what you expect it to be, like this:

```
var wi = new WrappedInt();
...
object o = wi;
if (o is WrappedInt)
{
    WrappedInt temp = (WrappedInt)o; // This is safe; o is a WrappedInt
    ...
}
```

The `is` operator takes two operands: a reference to an object on the left, and the name of a type (or `null`) on the right. If the type of the object referenced on the heap matches the type specified by the `is` operator, `is` evaluates to `true`; otherwise, `is` evaluates to `false`. The preceding code attempts to cast the reference to the object variable `o` only if it knows that the cast will succeed.

Another form of the `is` operator enables you to abbreviate this code by combining the type check and the assignment, like this:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
...
if (o is WrappedInt temp)
{
    ... // Use temp here
}
```

In this example, if the test for the `WrappedInt` type is successful, the `is` operator creates a new reference variable (called `temp`) and assigns it a reference to the `WrappedInt` object.

## The as operator

The `as` operator fulfills a similar role to `is` but in a slightly truncated manner. You use the `as` operator like this:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
WrappedInt temp = o as WrappedInt;
if (temp is not null)
{
    ... // Cast was successful
}
```

Like the `is` operator, the `as` operator takes an object and a type as its operands. The runtime attempts to cast the object to the specified type. If the cast is successful, the result is returned and, in this example, is assigned to the `WrappedInt` variable `temp`. If the cast is unsuccessful, the `as` operator evaluates to the `null` value and assigns that to `temp` instead.



**Note** There's a little more to the `is` and `as` operators than is described here; Chapter 12 discusses them in greater detail.

## The switch statement revisited

If you need to check a reference against several types, you can use a series of `if...else` statements in conjunction with the `is` operator. The following example assumes that you have defined the `Circle`, `Square`, and `Triangle` classes. The constructors take the radius (`radius`) or side length (`side`) of the geometric shape as the parameter:

```
var c = new Circle(42);           // Circle of radius 42
var s = new Square(55);          // Square of side 55
var t = new Triangle(33);        // Equilateral triangle of side 33
...
object o = s;
...
if (o is Circle myCircle)
{
    ... // o is a Circle, a reference is available in myCircle
}
else if (o is Square mySquare)
{
    ... // o is a Square, a reference is available in mySquare
}
else if (o is Triangle myTriangle)
{
    ... // o is a Triangle, a reference is available in myTriangle
}
```

As with any lengthy set of `if...else` statements, this approach can quickly become cumbersome and difficult to read. Fortunately, you can use the `switch` statement in this situation, as follows:

```
switch (o)
{
    case Circle myCircle:
        ... // o is a Circle, a reference is available in myCircle
        break;

    case Square mySquare:
        ... // o is a Square, a reference is available in mySquare
        break;

    case Triangle myTriangle:
        ... // o is a Triangle, a reference is available in myTriangle
        break;

    default:
        throw new ArgumentException("variable is not a recognized shape");
        break;
}
```

In both examples (using the `is` operator and the `switch` statement), the scope of the variables created (`myCircle`, `mySquare`, and `myTriangle`) is limited to the code inside the corresponding `if` block or `case` block.

`case` selectors in `switch` statements also support `when` expressions, which you can use to further qualify the situation under which the case is selected. For example, the following `switch` statement shows `case` selectors that match different sizes of geometric shapes:

```
switch (o)
{
    case Circle myCircle when myCircle.Radius > 10:
        ...
        break;
    case Square mySquare when mySquare.SideLength == 100:
        ...
        break;
    ...
}
```

## Pointers and unsafe code

This sidebar is purely for your information and is aimed at developers who are familiar with C or C++. If you're new to programming, feel free to ignore this information.

If you have already written programs in languages such as C or C++, much of the discussion in this chapter concerning object references might be familiar in that both languages have a construct that provides similar functionality: a pointer. A *pointer* is a variable that holds the address of, or a reference to, an item in memory (on the heap or the stack).

A special syntax is used to identify a variable as a pointer. For example, the following statement declares the variable `pi` as a pointer to an integer:

```
int *pi;
```

Although the variable `pi` is declared as a pointer, it does not actually point anywhere until you initialize it. For example, to use `pi` to point to the integer variable `i`, you can use the following statements and the address-of operator (`&`), which returns the address of a variable:

```
int *pi;  
int i = 99;  
...  
pi = &i;
```

You can access and modify the value held in the variable `i` through the pointer variable `pi` like this:

```
*pi = 100;
```

This code updates the value of the variable `i` to 100 because `pi` points to the same memory location as the variable `i`.

One of the main problems that developers learning C and C++ encounter is understanding the syntax used by pointers. The `*` operator has at least two meanings (in addition to being the arithmetic multiplication operator), and there's often great confusion about when to use `&` rather than `*`.

The other issue with pointers is that it's easy to point somewhere invalid or to forget to point somewhere at all, and then try to reference the data pointed to. The result will be either garbage or a program that fails with an error because the operating system detects an attempt to access an illegal address in memory.

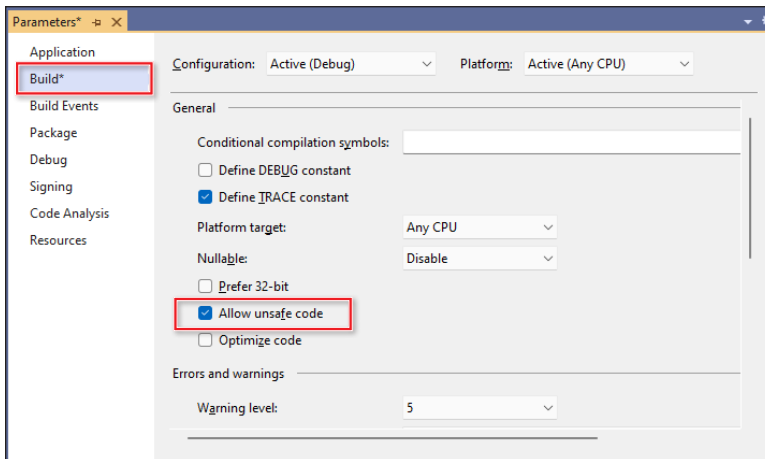
Finally, there are several security flaws in many existing systems resulting from the mismanagement of pointers. Some environments (not Windows) fail to enforce checks that a pointer does not refer to memory that belongs to another process, opening up the possibility that confidential data could be compromised.

Reference variables were added to C# to avoid all these problems. If you really want to, you can continue to use pointers in C#, but you must mark the code as unsafe. The `unsafe` keyword can be used to mark a block of code or an entire method, as shown here:

```
public static void Main(string [] args)
{
    int x = 99, y = 100;
    unsafe
    {
        swap (&x, &y);
    }
    Console.WriteLine($"x is now {x}, y is now {y}");
}

public static unsafe void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

When you compile programs containing unsafe code, you must specify the Allow Unsafe Code option when building the project. To do this, right-click the project in Solution Explorer and then select Properties. In the Properties window, select the Build tab, select Allow Unsafe Code, and then, on the File menu, select Save All.



Unsafe code also affects how memory is managed. Objects created in unsafe code are said to be unmanaged. Although situations that require you to access memory in this way are not common, you might encounter some, especially if you're writing code that needs to perform some low-level Windows operations.

You'll learn about the implications of using code that accesses unmanaged memory in more detail in Chapter 14.

# Summary

In this chapter, you learned about some important differences between value types that hold their value directly on the stack and reference types that refer indirectly to their objects on the heap. You also learned how to use the `ref` and `out` keywords on method parameters to gain access to the arguments. You saw how assigning a value (such as the `int 42`) to a variable of the `System.Object` class creates a boxed copy of the value on the heap and then causes the `System.Object` variable to refer to this boxed copy. You also saw how assigning a variable of a value type (such as an `int`) from a variable of the `System.Object` class copies (or unboxes) the value in the `System.Object` class to the memory used by the `int`.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 9.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

## Quick reference

To	Do this
Copy a value type variable	Simply make the copy. Because the variable is a value type, you will have two copies of the same value. For example: <pre>int i = 42; int copyi = i;</pre>
Copy a reference type variable	Simply make the copy. Because the variable is a reference type, you will have two references to the same object. For example: <pre>Circle c = new Circle(42); Circle refc = c;</pre>
Declare a variable that can hold a value type or the <code>null</code> value	Declare the variable by using the <code>?</code> modifier with the type. For example: <pre>int? i = null;</pre>
Pass an argument to a <code>ref</code> parameter	Prefix the argument with the <code>ref</code> keyword. This makes the parameter an alias for the actual argument rather than a copy of the argument. The method may change the value of the parameter, and this change is made to the actual argument rather than to a local copy. For example: <pre>static void Main() {     int arg = 42;     doWork(ref arg);     Console.WriteLine(arg); }</pre>
Pass an argument to an <code>out</code> parameter	Prefix the argument with the <code>out</code> keyword. This makes the parameter an alias for the actual argument rather than a copy of the argument. The method must assign a value to the parameter, and this value is made to the actual argument. For example: <pre>static void Main() {     int arg;     doWork(out arg);     Console.WriteLine(arg); }</pre>

To	Do this
Box a value	Initialize or assign a variable of type <code>object</code> with the value. For example: <code>object o = 42;</code>
Unbox a value	Cast the object reference that refers to the boxed value to the type of the value variable. For example: <code>int i = (int)o;</code>
Cast an object safely	Use the <code>is</code> operator to test whether the cast is valid. For example: <pre> WrappedInt wi = new WrappedInt(); ... object o = wi; if (o is WrappedInt temp) {     ... } </pre> Alternatively, use the <code>as</code> operator to perform the cast, and test whether the result is null. For example: <pre> WrappedInt wi = new WrappedInt(); ... object o = wi; WrappedInt temp = o as WrappedInt; if (temp != null)     ... </pre>

# Index

## SYMBOLS

&= compound assignment operator, 398  
? modifier  
    using with enumeration variables, 232  
    using with structure variables, 240  
+= operator, using with strings, 134  
== and != operators, and structs, 547  
+ (addition) operator, 60, 133  
& (AND) operator, 397–398  
\* (asterisk) after file name, Visual Studio 2022, 19  
@ (at sign), using with Circle objects, 206  
\ (backslash), 129  
{ } (curly braces)  
    matching, 17  
    terminology, 68  
-- (decrement) operator, 69, 134, 542  
/ (division) operator, 59  
\$ (dollar sign) symbol, using with strings, 60  
" (double quotation mark), 129  
[] ellipsis character, 41  
= (equal sign) assignment operator, 49, 112–113  
== (equal to) operator, 112–113  
// (forward slashes), using for comments, 24, 39  
> (greater than) operator, 113  
>= (greater than or equal to) operator, 113  
++ (increment) unary operator, 69–70, 134  
=> (lambda) operator, 459  
< (less than) operator, 113  
<= (less than or equal to) operator, 113  
<< (left-shift) operator, 397–398  
&& (logical AND) operator, 113–114  
\* (multiplication) operator, 59  
~ (NOT) operator, 397

! (NOT) operator, 112, 114, 215  
!= (not equal to) operator, 112–113  
| (OR) operator, 397  
|| (OR) operator, 113–114  
() (parentheses)  
    matching, 17  
    terminology, 68  
    wrapping expressions in, 524  
+ (plus sign), meaning in book, xxix  
; (semicolon), using with do statements, 142  
' (single quotation mark), 129  
- (subtraction) operator, 59  
[] (square brackets)  
    terminology, 68  
    using with arrays, 252  
~ (tilde), using with finalizers, 341  
// TODO: comments, 190, 208, 307–308, 617, 619–620  
^ (XOR) operator, 398

## A

abstract classes. *See also* classes  
    implementing and using, 331–335  
    and methods, 329–330, 338  
abstract keyword, 337  
accessibility, controlling, 184–186  
Action type, using with tasks, 563  
Action<T, ...> and Func<T, ...> delegate types, 486  
adapter methods, 500  
AddCardToHand method, 465  
AddItemToLocalCache helper method,  
    Cache-Aside pattern, 613  
addition (+) operator, 60, 133



- addresses and customers, LINQ example, 514–516
- AddToAccumulator method, 581–582
- addValues method, 65, 74, 77–78
- ADO.NET, 728
- Adventure Works Customers app, 645–649, 718. *See also* UWP apps
- AdventureWorks database
  - Customer table, 723
  - installing, 719–724
  - removing unneeded columns, 724–728
  - setting up, 719–724
- AdventureWorks web API project, 729–730
- AdventureWorksService
  - fetching data from, 755–759
  - modifying customer records, 768
  - NuGet Package Manager, 731
  - operations, 736–744
- AggregateException class, task exceptions, 594–596
- aggregating data, 519–521
- Alt key. *See* keyboard shortcuts
- AND (&) operator, 397–398
- AND (&&) operator, 113–114
- anonymous classes, 202–203. *See also* classes
- App.config, Solution Explorer, 14
- application logic, developing, 85
- applications
  - building, 44
  - running in debug mode, 476–476
  - running without debugging, 19, 44
- AppStyles.xaml file, displaying, 676, 679, 681–682
- App.xaml source file, Solution Explorer pane, 38–39, 677
- ArgumentException class, 279
- argumentList in method call, 78
- arguments, passing to methods, 215. *See also* named arguments and parameters
- arithmetic operators
  - assignment operator, 68–69
  - associativity, 68–69
  - combining with assignment operators, 134
  - controlling precedence, 67–68
  - evaluating expressions, 68
  - examining, 61–67
  - implementing, 544–547
  - numeric types and infinite values, 61
  - remainder, 60
  - string interpolation, 60
  - and types, 59–61
- array arguments, using, 278–279
- array elements
  - accessing, 255–256, 275
  - iterating through, 275
- array instances, creating, 252–253
- array variables, declaring, 252, 275
- arrays. *See also* parameter arrays
  - accessing elements, 255–256
  - comparing to collections, 462–466
  - containing value types, 272–275
  - copying, 259–260
  - creating instances of, 275
  - finding number of elements in, 275
  - foreach statement, 256–257
  - implicitly typed, 254–255
  - vs. indexers, 395, 401–403
  - initializing elements of, 275
  - jagged, 261–271, 276
  - multidimensional, 260–271, 276
  - naming, 252
  - parameters in Main method, 258
  - parameters or return values for methods, 257–259
  - passing as parameters, 256–257
  - populating and using, 253–254
  - properties and indexers, 402
  - as reference types, 259
  - returning from methods, 258
  - System.Array class, 469
  - using square brackets ([]) with, 252
  - using to implement card game, 262–271
  - vs. variables, 251
- The Art of Computer Programming*, 420
- as operator, 225
- ASP.NET Core SSL Certificate, 741
- ASP.NET Core web API project, creating web services in, 735, 738
- ASP.NET web API, 740–741, 767–768

assignment operators. *See also* compound assignment operators  
 and associativity, 68–69  
 combining arithmetic operators with, 134  
 equal sign (=), 49

associativity  
 and assignment operator, 68–69  
 of operators, 538  
 using to evaluate expressions, 68

assemblies, and namespaces, 23

asterisk (\*) after file name, Visual Studio 2022, 19

async keyword, 603

async modifier, using with delegates, 606

asynchronicity and scalability, 600

asynchronous disposal, handing, 359–360.  
*See also* Dispose method; exception-safe disposal

asynchronous methods. *See also* PLINQ (Parallel LINQ); threads  
 ConfigureAwait(false), 610  
 defining and calling, 603  
 defining to return values, 609  
 efficiency, 613–614  
 IAsyncResult design pattern, 614  
 implementing, 639  
 memory allocation, 5  
 naming, 607  
 pitfalls, 610–611  
 problem, 600–603  
 solution, 603–608  
 tasks, 613–614  
 and Windows Runtime APIs, 611–613

asynchronous operations, overview, 599–600

at sign (@), using with Circle objects, 206

audit-*nnnnnn.xml* file, opening, 492

Auditor class, 506

AuditOrder method, 494

AuditService project, 493

automated factory scenario, 487–490

automatic properties. *See also* properties  
 creating, 394  
 defining, 386–387  
 generating, 381–383  
 and immutability, 385–387

await keyword, 603

await operator, asynchronous operations, 360, 587, 604–605, 609

Azure account, signing up for, 719

Azure API app, REST web service published as, 770

Azure SQL Database server  
 creating, 719–724  
 Firewall settings, 723

## B

background images, applying, 679

backslash (\), 129

BankAccount class, 374

Barrier class, 627

base classes, 314

base-class constructors, calling, 292–293, 310.  
*See also* constructors

BasicCollection<T> class, 478

binary notation, specifying integer values as, 410

binary operators, 538

binary tree class, building using generics, 423, 426–433

binary trees  
 building with generic methods, 434–436  
 greater than expression, 421  
 theory of, 420–423

binary values  
 displaying, 397  
 manipulating, 397–398  
 storing, 396

BinaryTree, retrieving data from, 525–529

binding properties of controls and objects, 716

BitArray class, 446

bits variable, using, 398–399

bitwise operators, 398

Black.Hole method, 282

Blank App template, Adventure Works Customers app, 645–649

blocking wait operation, canceling, 640

blocks  
 using to group statements, 117–118  
 using with while statements, 136

- bold text, explained, xxix
- book
  - audience and starting point, xxvii–xxviii
  - conventions, xxviii–xxix
  - errata and support, xxxv
- bool data type, 50, 112
- bool keyword, type and structure, 237
- Boolean expressions
  - creating, 132
  - do statements, 142–143
  - while and for statements, 142
  - and while loops, 139
  - and while statements, 135
- Boolean operators
  - additive category, 115
  - assignment category, 115
  - conditional AND category, 115
  - conditional logical operators, 113
  - conditional OR category, 115
  - equality and relational operators, 112–113
  - equality category, 115
  - multiplicative category, 115
  - operator precedence and associativity, 114–115
  - pattern matching, 115
  - primary category, 114
  - relational category, 115
  - short-circuiting, 114
  - unary category, 114
- Boolean values
  - negating, 112, 114
  - true, 399
- Boolean variables
  - catchErrors, 158
  - declaring, 111–112, 132
- boxing and unboxing, 222–223
- boxing values, 230
- braces ({}), terminology, 68
- break
  - and continue statements, 143
  - and switch statements, 126
- breakpoints, inserting with Visual Studio Debugger, 94
- browse and search modes, ViewModel, 762–766

- Build Solution, selecting, 18
- built-in conversions, providing, 551
- Button class, 503
- Button control. *See also* More button
  - listing event names for, 40
  - separating business logic, 716
- Button element, 41
- buttons, adding to views in UWP apps, 713
- byte keyword, type and structure, 237

## C

- C#
  - case-sensitivity, 14, 46
  - as free format language, 46
  - history of, xxv–xxvi
- C# code, compiling, 18
- C# programs. *See* programs
- Cache-Aside pattern, 613–614
- cached and deferred evaluation, LINQ queries, 531–532
- calculateClick method, 79, 81, 163–164, 171
- CalculateFactorial function, 99
- CalculateInterest method, 381
- Calculator class, 352, 356
- camelCase notation, 48, 185
- CanBrowse property, 763
- Cancel method, adding, 765
- canceled tasks, using continuations with, 596.
  - See also* cooperative cancellation; tasks
- cancellation functionality, adding to
  - GraphDemo, 584–589
- cancellation of tasks, determining, 588–589, 592–594
- cancellation token, 582–584
- cancelling Parallel.For and Parallel.ForEach, 589
- CanExecute method, ViewModel, 706–708
- CanExecuteChanged method, ViewModel, 706
- CanSearch property, 763
- Card Game application, 263, 271
- card game
  - implementing using collections, 462–466
  - implementing with arrays, 262–271
- cascading if statements, 118–123

- case labels, using with switch statements, 125–126
- casting data safely, 224–226
- casting objects safely, 230
- catch block, 506
- catch handlers, using, 154–157
- catchall handler, using, 173
- catchErrors Boolean variable, 158
- catching exceptions, 50, 155, 157–158, 173–174, 177. *See also* try/catch statement block
- char data type, 50
- character pairs, matching, 17
- characters, reading from strings, 126–129
- checked expressions, writing, 166–168. *See also* expressions
- checked statements, writing, 166
- CheckoutButtonClicked method, 494, 498
- CheckoutController class, adding event to, 505–510
- CheckoutController component, creating, 495–499
- circle, calculating area of, 629–630
- Circle and Square classes
  - modifying, 334–335
  - using with interfaces, 322–328
- Circle class, 182–188, 198, 382–383
- Circle objects, using at sign (@) with, 206
- Class icon, IntelliSense, 17
- class keyword, using, 182–184
- class libraries, splitting into assemblies, 23
- class methods. *See* static methods and data
- class scope, defining, 83–84
- class types
  - declaring variables as, 206
  - handling, 205–206
- classes. *See also* abstract classes; anonymous classes; derived classes; interfaces; partial classes; sealed classes
  - and assemblies, 23
  - assigning, 293–295
  - and base classes, 314
  - copying, 205–211
  - creating hierarchy of, 300–305
  - defining, 204
  - defining and using, 182–184
  - and encapsulation, 182
  - fields in, 185
  - locating methods in, 56
  - vs. objects, 183
  - as reference types, 206
  - vs. structures, 238–240
  - using finalizer, 351–353
  - using namespaces with, 22
  - writing constructors for, 188
- classes and structures, comparing behaviors of, 246–247
- classification, understanding, 181–182
- CLI (command-line interface), 3
- Click attribute, 41
- Clone methods
  - using with arrays, 260
  - using with classes, 207–208
- cloud
  - deploying web services to, 746–751
  - importance of, xxvii
- CLR (common language runtime), 336, 600
- code
  - avoiding duplication in, 329
  - commenting out blocks of, 531
  - trying, 154–165
- Code and Text Editor window
  - opening, 54
  - using with graphical applications, 24–36
  - Visual Studio 2022, 13
- code samples, using, xxx–xxxv
- collection classes. *See also* concurrent collection classes; thread-safe collection
  - adding thread safety to methods in, 629
  - defining, 311–312
  - Dictionary<TKey, TValue>, 453–454
  - HashSet<T>, 455–457
  - LinkedList<T>, 449–450
  - List<T>, 447–449
  - making enumerable, 482
  - overview, 445–446
  - PriorityQueue<TElement, TPriority>, 451–452
  - SortedList<TKey, TValue>, 454–455
  - Stack<T>, 452–453
- collection initializers, using, 457–458

- collections. *See also* enumerating collections
  - adding items to, 467
  - comparing to arrays, 462–466
  - creating, 467
  - finding number of elements in, 467
  - iterating through elements of, 467
  - locating items in, 467
  - in memory, 522
  - removing items from, 467
  - using to implement card game, 462–466
- COM (Component Object Model), 100
- ComboBox control
  - and data binding, 699–701
  - data binding, 704–705
  - UWP app, 655–656
- command bars, resource for, 713
- Command class, implementing, 706–709
- command-line interface (CLI), 3
- commands
  - dir, 5, 7
  - dotnet run, 6
  - run, 6
- commenting code, 24. *See also* (forward slashes (//); // TODO: comments
- commenting out blocks of code, 531
- common language runtime (CLR), 336, 600
- compareClick method, 120
- CompareTo method, 312, 526
- compiling C# code, 18
- Complex struct, System.Numerics namespace, 544–547
- compound assignment operators, using, 133–134, 398, 541. *See also* assignment operators
- computer memory. *See also* memory allocation and efficiency
  - boxing, 222
  - organization of, 219–220
  - stack and heap, 220–221
  - System.Object class, 221
  - unboxing, 222–223
- concurrent collection classes. *See also* collection classes
  - adding thread safety, 629–638
  - and locks, 629–638
  - types, 628
- concurrent operations, being careful with, 622
- conditional logical operators, 113
- ConfigureAwait(false), asynchronous methods, 610
- Console Application template, Visual Studio 2022, 12–13
- console applications
  - creating using .NET CLI, 44
  - creating in Visual Studio 2022, 10–14
  - explained, 3
  - instance methods, 194–196
  - vs. UWP (Universal Windows Platform) app, 96
- Console class, 16, 277–278
- Console.Write method, 87
- Console.WriteLine method
  - data management, 521
  - parameter arrays, 277–278, 282
- Console.Write(v) statement, 94
- const field, declaring, 204
- const keyword, using, 199
- constructors. *See also* base-class constructors; deconstructor
  - calling, 201
  - declaring and calling, 204
  - deconstructing objects, 196–197
  - defaults, 186
  - defining, 384
  - overloading, 187–189
  - overview, 186–187
  - partial classes, 188
  - writing, 188–194
- continuations, using with tasks, 565, 596, 601–603
- continue and break statements, 143
- Continue button, Visual Studio Debugger, 93
- ContinueWith method, 564–566
- contravariant interfaces, defining, 441–444
- controller, adding to REST web service, 736–737
- controller classes, creating, 740–741
- controls
  - dragging on forms, 33
  - enabling for data binding, 716
  - enabling using data binding, 716

- conversion operators
  - defining, 550–555
  - writing, 553–555
- conversion operators, defining, 556. *See also* operators
- cooperative cancellation, 582–594. *See also* canceled tasks; tasks
- copying
  - arrays, 259–260
  - classes, 205–211
  - reference type variables, 229
  - reference types, 207–208
  - structure variables, 245–248
  - value type variables, 205–211, 229
- Count function, invoking, 524
- Count property, 712
- CountdownEvent class, 626
- covariant interfaces, defining, 440–441, 443–444
- CPU use, increasing for Task objects, 576
- CPU-bound code and Parallel class, 580
- .cs suffix, 5
- Ctrl key. *See* keyboard shortcuts
- curly braces ({}), matching, 17
- Current property, 704
- Customer class, INotifyPropertyChanged interface, 696–699
- Customer information
  - displaying, 689–694
  - modifying with two-way data binding, 694–699
  - ViewModel for, 702–705
- customers
  - displaying, 759–761
  - retrieving blocks of, 744–745
- customers and addresses, LINQ example, 514–516
- Customers app
  - extending, 768
  - scalable user interface, 649–659
  - searching for data in, 761–767
- Customers entity model, creating, 730–734
- Customers form
  - defining styles for, 675–685
  - Next and Previous buttons, 713–715

- CustomersController class, Scaffold wizard, 738–739

## D

- data
  - aggregating, 519–521
  - casting safely, 224–226
  - deleting through REST web services, 767–768
  - enumerating in order, 519, 534
  - filtering, 518–519, 523
  - grouping, 519–521, 523, 534
  - inserting through REST web services, 767–768
  - joining, 521–522, 535
  - locking, 625
  - ordering, 519–521, 523
  - retrieving, 519, 717–719
  - searching and sorting, 437
  - selecting for LINQ, 516–518
  - storage in LINQ, 515
  - synchronizing concurrent access to, 621–624
  - updating through REST web services, 767–768
- data access. *See* PLINQ (Parallel LINQ)
- data binding
  - ComboBox control, 704–705
  - controls and objects, 716
  - enabling controls for, 716
  - implementing for title ComboBox controls, 700–701
  - overview, 688
  - TextBox control, 704–705
  - using to modify data, 694–699
  - using with ComboBox control, 699–701
  - using to display Customer information, 689–694
  - UWP apps, 672
  - “Data Consistency Primer,” 768
- data privacy and copying reference types, 207–208
- data types. *See* primitive data types
- databases
  - removing columns from, 724–728
  - retrieving data from, 717–719

- DataTypes project, 493
- Date struct, 243, 246
- dateCompare method, 120
- dates, comparing, 123
- DbContext class, Entity Framework, 732–733
- DbSets, defining, 733
- DealCardFromPack method, 464
- Debug menu. *See also* Visual Studio Debugger
  - arithmetic operators, 547
  - automatic properties, 387
  - calculating pi, 634–635
  - cancellation functionality, 587
  - Card Game application, 263, 271
  - card game using collections, 466
  - catching unhandled exceptions, 174
  - checked expressions, 167–168
  - CheckoutController class, 509
  - CheckoutController component, 499
  - class using finalizer, 352–353
  - Complex struct, 547
  - console applications, 18
  - constructors and objects, 192
  - conversion operators, 555
  - Customers app, 658–659
  - data binding, 693–695, 701
  - deferred and cached evaluation, 532–533
  - do statements, 150
  - enumerations, 235
  - exception handling, 169
  - expression-bodied methods, 76
  - fetching data from web service, 758
  - GraphDemo single-threaded application, 569, 575, 579, 607–608
  - graphical application, 41–42
  - hierarchy of classes, 303–305
  - IDisposable interface, 355
  - if statements, 119
  - indexers in Windows application, 405, 409–410
  - inheriting from DrawingShape class, 335
  - INotifyPropertyChanged interface, 696–699
  - InsertIntoTree method, 436
  - instance methods, 196
  - locks for serializing method calls, 638
  - MathOperators project, 61–62
  - method definitions, 76
  - Next and Previous buttons, 714
  - object initializers, 387
  - objects and disposal, 356
  - OperationCanceledException exception, 593
  - optional parameters, 106–108
  - parallelized LINQ query, 618, 620
  - params array and optional parameters, 287–288
  - preventing object disposal, 357
  - primitive data type values, 52–53, 58
  - propagating exceptions, 164
  - properties, 379
  - ref parameters, 218
  - retrieving data from BinaryTree, 527–530
  - Square and Circle classes, 328
  - static using statements, 202
  - structure types, 244
  - structures and classes, 247
  - styles for Customers form, 684
  - switch expressions, 131
  - tabular layout and Grid control, 666
  - task cancellation, 593
  - task status, 590
  - thread-safe collection, 637
  - throwing exceptions, 170–172
  - Tree<TItem> class, 432
  - try/catch statement block, 159
  - unhandled exceptions, 155, 159
  - user interface, 35
  - Utils.Sum method, 285
  - value and reference parameters, 209
  - verifying disposal of objects, 359
  - ViewModel for customer information, 705
  - Visual State Manager and layout, 674
  - while statements, 136
- debug mode, running applications in, 44
- Debug toolbar
  - commands on, 148
  - displaying, 91–92, 109
- debugging in Visual Studio 2022, 19, 41–42
- decimal data type, 50
- decimal keyword, type and structure, 237

- decision statements
  - Boolean operators, 112–115
  - Boolean variables, 111–112
  - if statements, 116–123
  - switch expressions with pattern matching, 129–131
  - switch statements, 124–129
- deconstructor, implementing, 196–197. *See also* constructors
- decrement (-- operators, 69, 134, 542
- decrementing and incrementing, 134
- decrementing variables, 69–70, 72
- default keyword, using with TreeEnumerator class, 472
- deferred and cached evaluation, LINQ queries, 531–532
- Delegate icon, IntelliSense, 17
- delegate type, declaring, 511
- delegates
  - automated factory scenario, 487–490
  - CheckoutController component, 495–499
  - creating instance of, 511
  - declaring and using, 490
  - Func<T, ...> and Action<T, ...>, 486
  - vs. function pointers, 485
  - invoking, 511
  - and lambda expressions, 499–500
  - in .NET class library, 485–486
  - purpose of, 483
  - understanding, 484–485
  - using async modifier with, 606
  - Wide World Importers application, 491–495
- Delegates project, 493–494
- DELETE button, UWP apps, 744
- DELETE request, 768
- DeleteCustomer method, 740
- deleting
  - data through REST web service, 767–768
  - unneded columns from database, 727
- DeliveryService project, 493
- Dequeue method, 416
- derived classes. *See also* classes
  - creating from base classes, 310
  - methods in, 295
- Design Patterns: Elements of Reusable Object-Oriented Software*, 500
- Design View window
  - displaying forms in, 41
  - Visual Studio 2022, 27–36, 672
  - zooming in and out of, 54
- destructors vs. finalizers, 342
- developer mode, enabling, xxx
- Developer Mode, using with graphical applications, 26
- Dictionary<TKey, TValue> collection class, 446, 453–454
- dir command, running, 5, 7
- Dispatcher object, 603–604
- dispatch-nnnnn.txt file, opening, 493
- disposal methods, calling, 346–347
- Dispose method. *See also* asynchronous disposal
  - calling from finalizer, 349–350
  - purpose of, 348
  - and thread safety, 357–358
- DistanceTo method, 194
- Distinct method, 524
- divideValues method, 66
- division (/) operator, 59
- DLLs (dynamic-link libraries), 5. *See also* SystemRuntime.dll assembly
- do statements, writing, 142–150
- doAdditionalProcessing method, 623, 633
- doAuditing method, 506
- Document Outline window, displaying, 63
- doFirstLongRunningOperation method, 605
- dollar sign (\$) symbol, using with strings, 60
- doMoreAdditionalProcessing method, 623
- doShipping method, 507
- dotnet run command, using, 6
- double data type, 50
- double keyword, type and structure, 237
- double quotation mark ("), 129
- double variable, initializing, 195
- doWork method, 244, 526
- DoWorkWithData method, 101
- Drawing Pad window, opening, 328, 380
- DrawingShape abstract class, 331–335



## duplication in code, avoiding

duplication in code, avoiding, 329  
dynamic-link libraries (DLLs), 5. *See also*  
    SystemRuntime.dll assembly

## E

ellipsis character [...], 41  
else statements, 116, 123  
encapsulation  
    implementing by using methods, 366–367  
    purpose of, 182  
Entity Framework  
    DbContext class, 732  
    mapping layer, 728  
    SQL SELECT commands, 728  
    using, 770  
    using with relational databases, 718  
    version availability, 731  
entity model, creating, 728–734, 770  
Enum icon, IntelliSense, 17  
Enumerable class, System.Linq namespace,  
    522–523  
enumerable collections  
    filtering rows from, 534  
    projecting fields from, 534  
enumerating collections, 469–470. *See also*  
    collections  
enumeration variables  
    assigning to values, 249  
    declaring, 249  
    nullable versions of, 232  
enumerations  
    choosing literal values, 233  
    choosing underlying types, 233–236  
    creating and using, 234–236  
    declaring, 231, 249  
    using, 232  
enumerators. *See also* IEnumerable interface  
    adding to Tree<TItem> class, 480–481  
    defining for Tree<TItem> class, 479–480  
    implementing, 470–474, 477–482  
    as pointers, 470  
    testing, 476–477, 481  
Equal method, 548  
equal sign (=) assignment operator, 49, 112  
equal to (==) operator, 112–113  
equality and relational operators, 112–113  
equality operators, overriding, 547–550  
equi-joins, support in LINQ, 524  
Error List window, displaying, 19, 548  
errors, occurrence of, 154  
errors and exceptions, overview, 153  
Event icon, IntelliSense, 17  
EventArgs argument, 504  
events  
    adding to CheckoutController class, 505–510  
    declaring, 501, 511  
    raising, 502–503, 512  
    subscribing to, 502, 511  
    unsubscribing from, 502, 512  
    user interface, 503–504  
    using, 504–510  
    using to enable notifications, 500–503  
Example class  
    defining class scope, 83  
    and IDisposable interface, 350  
    using enumeration, 232  
Exception family, 157  
exception handling, Visual Studio debugger,  
    169  
exceptions. *See also*  
    OperationCanceledException exception  
    catching, 154–155, 157–158, 177  
    filtering, 158–163  
    propagating, 163–164  
    raised by tasks, 598  
    throwing, 170–175, 177  
exception-safe disposal. *See also* asynchronous  
    disposal  
    class using finalizer, 351–353  
    finally block, 346–347  
    implementing IDisposable interface,  
        353–355, 362  
    preventing objects from disposed, 356–358  
    verifying object disposal, 358–359  
Execute method, ViewModel, 706–708  
Exists method, 485  
explicit conversion, 551–552  
expression-bodied methods, defining,  
    76–77, 109

expressions. *See also* checked expressions  
 catching, 172–173  
 comparing values of, 132  
 evaluating using associativity, 68–69  
 Extensible Application Markup Language (XAML). *See* XAML (Extensible Application Markup Language)  
 extension methods. *See also* methods  
 creating, 305–309  
 defining for types, 310  
 icon, 17  
 and LINQ, 517  
 using with BinaryTree, 525–529  
 Extract Method Wizard, 97. *See also* methods

## F

factorials, calculating, 98–100  
 factory scenario. *See* automated factory scenario  
 failing operations, retrying, 740  
 fall-through, stopping, 126  
 faulted tasks, using continuations with, 596.  
*See also* tasks  
 FFT (Fourier transform), 633  
 field names and properties, 369  
 fields. *See also* shared fields  
 hiding using methods, 366–367  
 projecting from enumerable collection, 534  
 projecting with Select method, 518  
 and variables, 83–84  
 file I/O, 612. *See also* I/O operations  
 FileOpenPicker class, 612  
 FileProcessor class, 341–342  
 files  
 creating, 5  
 opening, 137  
 filtering data, 523  
 finalizers  
 calling Dispose method from, 349–350  
 vs. destructors, 342  
 recommendations, 345  
 using with classes, 351–353  
 finalizers, writing, 340–342, 361

finally block  
 using, 175–177, 506  
 using with exception-safe disposal, 346–347  
 Find and Replace command, 55  
 Find method, 459, 485  
 FindCustomerAsync method, 764–765  
 FindCustomers method, 745  
 FinishFolding method, 500  
 Firewall settings, Azure SQL Database server, 723  
 first-in, first-out structure, 413–414  
 float data type, 50  
 float keyword, type and structure, 237  
 folders, creating, 719–714  
 for statements, writing, 140–142  
 foreach loop, using with grouped data, 520  
 foreach statement  
 calculating pi, 633  
 retrieving data from BinaryTree, 528  
 using with arrays, 256–257, 469  
 using with iterators, 479  
 using with List<T> collection, 447  
 form terminology, using with applications, 28  
 FormatException type, 155, 157, 160  
 forms. *See also* Windows Forms app  
 displaying in Design View, 41  
 dragging controls on, 33  
 Fortran, updates of, xxv  
 forward slashes (//), using for comments, 24, 39. *See also* commenting code  
 Fourier transform (FFT), 633  
 Frame object, creating, 39  
 free format language, C# as, 46  
 Func<T, ...> and Action<T, ...> delegate types, 486  
 function pointers vs. delegates, 485  
 functions vs. methods, 4

## G

Gamma, Erich, 500  
 garbage collection. *See also* objects  
 forcing, 361  
 function of, 344–345  
 using, 343–344, 351

- GC class, 357
  - Generate Method Stub Wizard, using, 86–89, 109. *See also* methods
  - generateGraphData method, 569, 573–574, 578–579, 585, 593
  - generateGraphDataAsync method, 607–608
  - generic classes
    - binary trees, 420–423
    - Tree<TItem>, 426–433
  - generic interfaces and variance
    - contravariant interfaces, 441–443
    - covariant interfaces, 440–441
    - overview, 438–440
  - generic methods. *See also* methods
    - creating, 433–434, 444
    - InsertIntoTree, 434–436
  - generic types, using, 444
  - generics
    - and constraints, 419
    - vs. generalized classes, 419
    - issues with object type, 413–416
    - solution, 417–418
    - using to build binary tree class, 423, 426–433
  - gestures, UWP apps, 643
  - get accessor, implementing, 406–407
  - get and set keywords, using with properties, 368
  - GetCachedeValue helper method, Cache-Aside pattern, 613
  - GetCustomers method, 739–740
  - GetDataAsync method, 756–757, 765
  - GetEnumerator method, 470, 478
  - GetHashCode method, overriding, 548
  - globally unique identifier (GUID), 492
  - Grade class, 385
  - Grade struct, 388–389
  - GraphDemo single-threaded application, 566–571
    - asynchronous method, 606–608
    - cancellation functionality, 584–589
    - modifying to use Task objects, 573–576
  - graphical applications
    - adding code, 40–43
    - creating, 24–36
    - .NET MAUI (Multi-platform Application User Interface), 43
    - UWP (Universal Windows Platform) app, 37–39
    - Windows Forms app, 43
    - WinUI3, 42
    - WPF (Windows Presentation Foundation), 43
  - graphical user interface (GUI), 3
  - greater than (>) operator, 113
  - greater than or equal to (>=) operator, 113
  - Grid control
    - using to implement tabular layout, 659–667
    - using with Visual State Manager, 669–670
  - Grid.Row attribute, adding to TextBlock control, 662
  - GroupBy method, 520, 528
  - grouping data, 519–521, 523
  - GUI (graphical user interface), 3
  - GUID (globally unique identifier), 492
- ## H
- hard disk space requirement, xxix
  - HashSet<T> collection class, 455–457
  - Haskell programming language, 458
  - HeaderStyle, adding, 679
  - heap and stack, using, 219–221
  - “Hello World!” application
    - building, 3–6
    - using .NET CLI tools, 6–9
  - Hello YourName! message, displaying, 14–21
  - HelloUWP namespace, 38–39
  - HelloWorld2.csproj file, 8
  - Helm, Richard, 500
  - helper methods, 97–100
  - hexadecimal notation, specifying integer values as, 410
  - hill-climbing algorithm, Windows Runtime (WinRT), 563
  - Horse class, 329
  - Hour struct, 539–540, 542–543, 552
  - HTTP DELETE request, handling, 740
  - HTTP POST request, responding to, 740
  - HTTP PUT request, sending, 740
  - HttpClient object, creating and initializing, 756

- I
- IAsyncOperation interface, 612
- IAsyncResult design pattern, 614–615
- IColor interface, defining, 320–322
- ICommand interface, ViewModel, 706
- IComparable<T> interface, 471–472, 475, 480, 526
- IDE (interactive development environment), 3
- idempotency in REST web services, 768
- identifiers, using, 45–46
- IDisposable interface
  - implementing, 353–355
  - and using statement, 347–350
- IDraw interface, defining, 320–322
- IEnumerable interface. *See also* enumerations
  - vs. IEnumerator, 470
  - implementing, 470, 475–477
  - and LINQ, 515
- IEnumerator<TItem> interface, 473
- if statements
  - accidental assignments, 117
  - nesting, 118–123
  - using, 116–123, 132, 136
  - writing, 119–123
- immutable properties, creating, 382
- immutable types, creating and instantiating, 394
- implicit conversion, 551
- implicitly typed local variables, declaring, 70–71. *See also* variables
- Important items, explained, xxviii
- increment operators, declaring, 542
- incrementing and decrementing, 134
- incrementing variables, 69–70, 72
- index accessors, 400–401
- indexers. *See also* interface indexer
  - vs. arrays, 395, 401–403
  - binary values, 396–398
  - calling, 407–409
  - creating for classes, 410
  - creating for structures, 410
  - defining in interfaces, 410
  - features of, 400
  - implementing, 411
  - in interfaces, 403–404
  - and operators, 538
  - overview, 395
  - properties and arrays, 402
  - solving problems, 398–400
  - using in Windows applications, 404–410
  - writing, 406–407
- inequality (!=) operator, 112–113
- Infinity value, 61
- inheritance
  - assigning classes, 293–295
  - calling base-class constructors, 292–293
  - calling base-class constructors, 310
  - declaring methods, 295–296
  - declaring override methods, 297–298, 310
  - declaring virtual methods, 296–297, 310
  - extension methods, 305–310
  - hierarchy of classes, 301–305
  - and interfaces, 330
  - overview, 289–290
  - protected access, 300–305
  - System.Object class, 292
  - using, 290–292
  - virtual methods and polymorphism, 298–299
- initializing variables, 211
- INotifyPropertyChanged interface, Customer class, 696–699
- InsertIntoTree method
  - testing, 436
  - writing, 434–436
- instance methods, writing and calling, 194–196
- int, referring to minimum value of, 165
- int data type, 50
- int keyword, type and structure, 237
- int types, 396
- int variable type, 49. *See also* long integers
- Int32 method, .NET libraries, 59
- IntBits struct, 399–400
- integer arithmetic, checked and unchecked, 165–168, 177
- integer values, displaying as binary or hexadecimal, 410
- Intel, 560
- IntelliSense, 15–17, 195
- interactive development environment (IDE), 3
- Interface icon, IntelliSense, 17

## interface indexer, implementing

- interface indexer, implementing, 411.
  - See also* indexers
- interface properties. *See also* properties
  - declaring, 375–381
  - implementing, 393
- interfaces. *See also* classes
  - declaring, 338
  - declaring properties in, 393
  - defining, 312–313, 318, 320–328
  - explicitly implementing, 316–317
  - extending, 318
  - handling versioning with, 318–319
  - Hungarian notation, 313
  - IDraw and IColor, 320–322
  - implementing, 313–314, 338
  - indexers in, 403–404
  - and inheritance, 330
  - inheriting from, 314
  - overview, 311–312
  - reducing coding errors, 314
  - referencing classes through, 314–315
  - restrictions, 313, 319–320
  - Square and Circle classes, 322–328
  - working with, 315
- Internet connection requirement, xxx
- InvalidOperationException, 172
- I/O operations, waiting for completion, 600.
  - See also* file I/O
- is operator
  - using, 224
  - using in pattern matching, 115
- IsCardAlreadyDealt method, 464
- IsSearching property, 763
- IStoreWrapper<T> interface, 440
- iterators, using to implement enumerators, 477–481
- IWrapper<T> interface, 439–440

## J

- jagged arrays, creating, 261–271, 276
- Johnson, Ralph, 500
- join operator, using, 524
- joining data, 521–522

## K

- keyboard shortcuts
  - Quick Find dialog, 55
  - Visual Studio Debugger, 91–92
  - zooming in and out of Design View, 54
- keywords, identifying, 46–47
- Knuth, Donald E., 420

## L

- lambda expressions
  - and anonymous methods, 461
  - body of, 459–460
  - and delegates, 499–500
  - forms of, 460–461
- lambda (=>) operator, 459
- (Language-Integrated Query (LINQ)), 599.
  - See also* PLINQ (Parallel LINQ)
  - aggregating data, 519–521
  - commenting out blocks of code, 531
  - data storage, 515
  - and deferred evaluation, 530–533
  - equi-joins, 524
  - expressions in on clause, 524
  - filtering data, 518–519
  - grouping data, 519–521
  - joining data, 521–522
  - ordering data, 519–521
  - overview, 513–514
  - query operators, 522–524
  - querying data in Tree<TItem> objects, 525–530
  - selecting data, 516–518
  - summarizing information, 524
  - using in C# application, 514–516
- large methods, 97–100
- layout, adapting using Visual State Manager, 667–674
- left-shift (<<) operator, 397–398
- Length property, using with arrays, 256
- less than (<) operator, 113
- less than or equal to (<=) operator, 113
- lightweight structures, implementing, 388–392
- LinkedList<T> collection class, 446, 449–450

- LINQ (Language-Integrated Query), 599.
    - See also* PLINQ (Parallel LINQ)
    - aggregating data, 519–521
    - commenting out blocks of code, 531
    - data storage, 515
    - and deferred evaluation, 530–533
    - equi-joins, 524
    - expressions in on clause, 524
    - filtering data, 518–519
    - grouping data, 519–521
    - joining data, 521–522
    - ordering data, 519–521
    - overview, 513–514
    - query operators, 522–524
    - querying data in `Tree<TItem>` objects, 525–530
    - selecting data, 516–518
    - summarizing information, 524
    - using in C# application, 514–516
  - LINQ queries
    - forcing results for, 535
    - parallelizing, 639
    - parallelizing over collection, 616–618
    - parallelizing to join collections, 619–621
  - `List<T>` collection class, 446–449, 485
  - literal values, 49
  - local scope, defining, 83
  - local variables. *See* unassigned local variables
  - locking data, 625
  - locks
    - thread safety and `Dispose` method, 357–358
    - using to serialize method calls, 638
  - logical AND (&&) operator, 113–114
  - logical OR (||) operator, 113–114
  - long data type, 50
  - long integers, 50. *See also* int variable type
  - long keyword, type and structure, 237
  - loops, nesting, 135
- ## M
- Main method
    - array parameters in, 258
    - and asynchronous methods, 604
    - calling `doWork` method, 189
    - using, 9
  - MainPage class, 509
  - MainPage constructor, 704
  - MainPage form, displaying and activating, 39
  - MainPage.xaml.cs, displaying, 37
  - Mammal class, 291–299
  - ManualResetEventSlim class, 626
  - Math class, 183, 197, 199
  - MathOperators application, performing calculations in, 63–66
  - MathOperators application, 61–62, 67
  - memory allocation and efficiency, 613–614.
    - See also* computer memory
  - method calls, serializing with locks, 638
  - method definitions, viewing, 105
  - Method icon, IntelliSense, 17
  - method signature, explained, 295
  - methodName, 74, 78
  - methods. *See also* abstract classes; extension methods; Extract Method Wizard; Generate Method Stub Wizard; generic methods; override methods; statements; summary methods; virtual methods; Visual Studio Debugger
    - accepting arguments, 288
    - call syntax, 77–80
    - calling, 109
    - constructors as, 186–187
    - declaring, 74–75, 109, 295–296
    - in derived classes, 295
    - and encapsulation, 366–367
    - explained, 73
    - expression-bodied methods, 76–77
    - finding, 458–460
    - vs. functions, 4
    - locating in classes, 56
    - nesting, 97–100
    - with optional parameters, 104–108
    - overloading, 84
    - overriding vs. hiding, 297
    - passing arguments to, 215
    - replacing with properties, 376–380
    - return types and void, 74
    - returning arrays from, 258
    - returning data from, 75
    - returning values from, 80–82, 109

methods (*continued*)

- specifying call syntax, 77–80
- specifying for tasks, 597
- stepping into and out of, 109
- syntax, 74
- tuples, 80–82
- and var keyword, 74
- variables in, 185
- writing, 85–89
- Microsoft IntelliSense, Visual Studio 2022, 15–17, 195
- Microsoft .NET and multitasking. *See also* tasks overview, 561
  - tasks, threads, and ThreadPool, 562–563
- Microsoft Press, contacting, xxxv
- Microsoft Windows platform, significance of, xxvi
- Min method, using with params array, 279–280
- models and views, communication between, 702
- Model-View-ViewModel (MVVM) pattern, implementing, 687–688
- Moore, Gordon E., 560
- Moore’s Law, 560–561
- More button, adding, 759–761. *See also* Button control
- More Info items, explained, xxviii
- MoveNext method, 473–474
- multicore processor, rise of, 560–561
- multidimensional arrays, using, 260–271
- multiplication (\*) operator, 59
- multiplyValues method, 66
- multitasking and parallel processing, 559–560
- multitasking using Microsoft .NET. *See also* tasks overview, 561
  - tasks, threads, and ThreadPool, 562–563
- MVVM (Model-View-ViewModel) pattern, implementing, 687–688
- MyFileUtil application, 259

## **N**

- named arguments and parameters, 100–108. *See also* arguments
- nameof operator, 699
- Namespace icon, IntelliSense, 17

- namespaces
  - and assemblies, 23
  - using, 21–22
- NaN (not a number) value, 61
- narrowing conversion, 551
- nested loops, creating, 135
- nested methods, creating, 109
- nesting
  - if statements, 118–123
  - methods, 97–100
- .NET class library
  - collection types, 446
  - exception classes, 170
- .NET CLI (command-line interface)
  - building and running projects, 6–9
  - creating console applications, 44
- .NET Framework, 11, 23, 625
- .NET libraries
  - and features, 336
  - Int32 method, 59
- .NET MAUI (Multi-platform Application User Interface), 43
- new keyword, 337
- Next and Previous buttons, ViewModel, 713–715
- NextCustomer command, ViewModel, 710–715
- NOT (~) operator, 397
- NOT (!) operator, 112, 114, 215
- not equal to (!=) operator, 112–113
- not a number (NaN) value, 61
- Notepad, accessing, 4
- Notes, explained, xxviii
- notifications, enabling by using events, 500–503
- NuGet Package Manager,
  - AdventureWorksService, 731
- null values and nullable types, 211–215, 229
- nullable types, properties of, 214–215
- null-coalescing operator, 213
- null-conditional operator, 212–214
- NullReferenceException exception, 489
- number generator, System. Random class, 252
- numeric types and infinite values, 61
- numeric values, specifying, 49–50. *See also* values

## O

- object initializers, using, 386–387
- object keyword, 221, 237
- object types and generics, 413–416, 438
- ObjectComparer object, 442
- objects. *See also* garbage collection
  - binding and data values, 716
  - binding properties of controls to, 716
  - casting safely, 230
  - vs. classes, 183
  - creating, 189–194, 201, 339
  - creating references to, 343
  - deconstructing, 196–197
  - enabling to notify bindings, 716
  - garbage collector, 343–345
  - initializing using properties, 383–387, 394
  - instantiating using generic types, 444
  - managing lifetimes of, 343
  - preventing from being disposed, 356–358
  - recommendations, 345
  - verifying disposal of, 358–359
  - writing finalizers, 340–342
- OK button, writing code for, 40–42
- onClick method, adding to MainPage class, 40–41
- OnPropertyChanged method, 697
- opening files, 137
- OperationCanceledException exception, 592–594. *See also* exceptions
- operator pairs, defining, 543–544
- operator precedence and associativity, summarizing, 114–115
- operators. *See also* conversion operators; symmetric operators
  - associativity, 538
  - comparing in structures and classes, 542–543
  - compound assignment evaluation, 541
  - constraints, 538
  - implementing, 544–547, 556
  - increment and decrement, 542
  - and indexers, 538
  - overloading, 539–540
  - precedence, 537
  - semantics, 537

- symmetric, 540–541
  - understanding, 537–538
- optional parameters, using, 100–108
- optMethod method, 103
- OR (!) operator, 397
- OR (!!) operator, 113–114
- OrderByDescending method, 519
- ordering data, 519–521, 523
- out and ref parameters, using, 215–218
- out parameters
  - creating, 216–218
  - passing arguments to, 229
- overflow checking, Visual Studio 2022, 165
- OverflowException, occurrence of, 169, 551
- overloaded operators, 539–540
- overloading
  - methods, 84
  - and parameter arrays, 277–278
- override keyword, 337
- override methods, declaring, 297–298, 310. *See also* methods

## P

- Parallel class
  - avoiding use of, 580–581
  - using to abstract tasks, 576–580
- Parallel LINQ (PLINQ). *See also* asynchronous methods; LINQ (Language-Integrated Query)
  - cancelling queries, 621
  - and declarative data access, 616
  - performance and collections, 616–621
- parallel processing and multitasking, 559–560
- parallel tasks. *See also* tasks
  - calculating pi, 634–636
  - loop iterations, 597
  - statement sequences, 597
- Parallel.For and Parallel.ForEach, canceling, 589
- parallelism
  - implementing using Task class, 566–576
  - increasing, 580
- parallelizing
  - declarative data access, 616–621
  - LINQ queries, 639



## ParallelPI method

- ParallelPI method, 636
- ParallelTest method, 623–624
- parameter arrays. *See also* arrays
  - array arguments, 278–279
  - and Console.WriteLine method, 282
  - vs. optional parameters, 286–288
  - and overloading, 277–278
  - using, 277
- parameterList in method declaration, 74
- parameters
  - of containing types, 539
  - and named arguments, 100–109
  - naming parameters, 539
  - specifying by name, 102
- Parameters solution, opening, 208
- params array
  - declaring, 279–281
  - using, 283–285
  - writing, 283–284
- params object[], using, 281–282
- parentheses (())
  - matching, 17
  - terminology, 68
  - wrapping expressions in, 524
- partial classes, 188. *See also* classes
- Pass.Reference method, 211
- passwords, protecting, 734
- pattern matching
  - with properties, 380–381
  - using switch expressions with, 129–131
  - using, 115
- performance and PLINQ, 616–621
- Performance Profiler, GraphDemo single-threaded application, 571–573
- Person class, 272
- Person objects, array of, 272
- Person struct, 437, 458–459, 485
- Phone Book application, 405
- pi, calculating, 629–636
- PI field, Math class, 197
- PickMultipleFileAsync method, 612
- PickSingleFileAsync method, 612
- pixel (px) suffix, 32
- PlayingCard class, 264
- PLINQ (Parallel LINQ). *See also* asynchronous methods; LINQ (Language-Integrated Query)
  - cancelling queries, 621
  - and declarative data access, 616
  - performance and collections, 616–621
- PLINQ queries
  - canceling, 621
  - enabling cancellation in, 639
- plotButton\_Click method, 567, 574, 586, 606, 608
- plotXY method, 569
- plus sign (+), meaning in book, xxix
- Point class, 191–192, 196, 200
- pointers
  - enumerators as, 470
  - and memory management, 437–438
  - Span<T> type, 437–438
  - and unsafe code, 227–228
- pointsList collection, 636–637
- Polygon class, 386
- polymorphism and virtual methods, 298–299
- POST button, 744
- POST request, 768
- PostCustomer method, 740
- precedence of operators, 67–68, 72, 537–538
- predicates, finding, 458–460
- prefix and postfix, 70
- Previous and Next buttons, View Model, 713–715
- PreviousCustomer command, ViewModel, 710–715
- primitive data types
  - displaying values, 51–58
  - overview, 50, 205
  - unassigned local variables, 51
  - using in code, 53–58
- PriorityQueue<TElement, TPriority> collection class, 446, 451–452
- private and static keywords, writing, 201
- private fields, instance methods, 194
- private keyword, 337
- processor requirement, xxix
- Program class, 189
- Program.cs, Solution Explorer, 8–9, 14, 209–210

programming languages, updates of, xxv

programs

- using Visual Studio 2022, 14–21
- writing, 3–9

project file, example of, 5

projects

- adding, 495
- building and running with .NET CLI, 6–9
- locating items in, 55
- opening, 52

properties. *See also* automatic properties; interface properties; read-only properties; records with properties; write-only properties

- accessibility, 372
- arrays and indexers, 402
- declaring, 368
- declaring in interfaces, 393
- expression-bodied members, 369
- and field names, 369
- getters and setters, 367–368
- implementing, 376
- modifying set accessors, 697
- overview, 367–370
- pattern-matching with, 380–381
- and public fields, 381
- read-only, 371
- read/write, 370, 393
- replacing methods with, 376–380
- restrictions, 373
- simulating assignment (=) operator, 538
- in Solution Explorer, 14
- and types, 378
- using, 370, 374, 376–380
- using to initialize objects, 383–387, 394
- write-only, 371–372, 393

properties of controls, binding to properties of objects, 716

protected access, 300–305

protected keyword, 337

public and private keywords, using, 184–186, 300

public constructor, using, 186–187

public fields and properties, 381

public keyword, 337

PUT button, 744

PUT request, 768

PutCustomer method, 740

px (pixel) suffix, 32

Pythagoras' theorem, using, 195

## Q

QBF (Query By Forms), 761–762

query operators

- using, 522–524
- using with BinaryTree, 529–530

querying data in Tree<TItem> objects, 525–530

Queue class, 413–417

Queue<T> collection class, 446, 450–451

Quick Find dialog, displaying, 55

QuickWatch dialog, accessing, 161

## R

radius, declaring as private field, 184–185

RAM requirement, xxix

read access, sharing, 640

ReaderWriterLockSlim class, 626–627

ReadLine method, 87

read-only properties, 371. *See also* properties

read/write property, declaring, 370, 393

record types, defining and using, 390–392

records with properties, implementing

- lightweight structures, 388–392.
- See also* properties

ref and out parameters, using, 215–218

ref parameters

- creating, 216
- passing arguments to, 229
- using, 208–211, 217–218

reference type variables, copying, 229

reference types

- arrays as, 259
- classes as, 206
- copying, 207–208

reference variables, initializing, 211

References, Solution Explorer, 14

reflection and equality operators, 547

relational and equality operators, 112–113

- relational database
  - tables in, 522
  - and web services, 718
- remainder operator, 60
- remainderValues method, 66
- RenderTransform property, 681
- requestPayment method, 494
- resource management
  - disposal methods, 346
  - Dispose method and finalizer, 349–350
  - exception-safe disposal, 346–347
  - overview, 345
  - using statement and IDisposable interface, 347–348
- resources, releasing, 361
- response time, issues with, 599
- responsiveness, improving, 559
- REST web service. *See also* UWP apps; web services
  - accessing URLs, 735
  - adding controllers, 736–737
  - adding data items to, 770
  - creating and consuming, 770
  - creating and using, 735–744
  - data management, 767–768
  - deleting data, 767–768
  - and entity model, 770
  - HTTP protocol, 735
  - idempotency in, 768
  - inserting data, 767–768
  - publishing as Azure API app, 770
  - PUT requests, 768
  - remote access, 770
  - retrieving data from, 770
  - updating data, 767–768, 770
- result = clause in method call, 78
- RetrieveInOrder method, using with classes, 311
- RetrieveValue helper method, Cache-Aside pattern, 613
- return statements
  - adding to DistanceTo method, 195
  - using with methods, 74
  - and switch statements, 126
- returnType in method declaration, 74
- rows, filtering from enumerable collection, 534

- run command, using, 6
- Run To Cursor, using with do statement, 147
- running statements, 151

## S

- Save As command, using, 5
- sbyte keyword, type and structure, 237
- Scaffold wizard, CustomersController class, 738
- scalability, improving, 560
- scope
  - applying, 82
  - class scope, 83–84
  - local scope, 83
  - overloading methods, 84
  - and variables, 82
  - writing methods, 85–89
- ScreenPosition struct, 366–368, 371–372, 376
- sealed classes, creating, 330–335, 338.
  - See also* classes
- sealed keyword, 337
- search and browse modes, ViewModel, 762–766
- Search method, defining, 765
- searching
  - for data in Customers app, 761–767
  - enhancing in UWP apps, 745–746
  - and sorting data, 437
- Select method
  - and summary methods, 520–521
  - using with LINQ, 516–518
- semantics, defined, 46
- SemaphoreSlim class, 626
- semicolon (;), using with do statements, 142
- SerialPI method, 631–633, 636
- SerialTest method, 623–624
- server firewall, setting for AdventureWorks database, 722
- set accessors, using with properties, 379, 697
- set and get keywords, using with
  - properties, 368
- shared fields, creating, 198–199. *See also* fields
- short keyword, type and structure, 237
- short-circuiting, 114
- Show Steps, using with do statement, 147

- ShowAsync() method, 41
- showStepsClick method, 145
- single quotation mark ('), 129
- .sln suffix, 52
- slowMethod method, 601, 604
- Solution Explorer pane, Visual Studio 2022, 13–14
- Solution 'TestHello,' Solution Explorer, 13
- solutions, loading, 52
- SortedList<TKey, TValue> collection class, 446, 454–455
- sorting and searching data, 437
- Span<T> type, 437–438
- SQL (Structured Query Language), 514
- SQL SELECT commands, Entity Framework, 728
- square, calculating area of, 629–630
- Square and Circle classes
  - accessing members, 340
  - modifying, 334–335
  - testing, 326–328
  - using with interfaces, 322–326
- square brackets ([])
  - terminology, 68
  - using with arrays, 252
- stack and heap, using, 219–221
- Stack<T> collection class, 446, 452–453
- statement scope, understanding, 142
- statements. *See also* methods
  - associating with different values, 132
  - break, 143
  - continue, 143
  - do, 142–150
  - executing, 151
  - for, 140–142
  - grouping with blocks, 117–118
  - overview, 45–46
  - running, 151
  - while, 135–140
- static classes, understanding, 199
- static fields, declaring and accessing, 204
- static keyword, using to define methods, 208
- static members, writing, 200–202
- static methods and data
  - overview, 197–198
  - shared fields, 198
  - static classes, 199
  - static fields, 199
  - static using statements, 200–202
- static methods, calling, 200–202, 204
- static properties, declaring, 371
- static using statements, 200–202
- Step Into, using with do statement, 148–150
- stepping through methods. *See* Visual Studio Debugger
- steps, showing, 147
- StopWatch type, 568
- string data type, 50
- string interpolation, 60, 72
- string keyword, type and class, 237
- string type, 206
- strings, reading characters of, 126–129
- Struct icon, IntelliSense, 17
- struct keyword, using, 237, 240
- structs, == and != operators, 547
- structure types
  - creating and using, 242–245
  - declaring, 249
- structure variables
  - copying, 245–247
  - declaring, 240, 249
  - initializing to values, 249
  - nullable versions of, 240
- Structured Query Language (SQL), 514
- structures
  - vs. classes, 238–240
  - declaring, 237–238
  - handling large ones, 248
  - initialization, 240–245
  - and operators, 238
  - types, 236–237
  - using, 238
- structures and classes, comparing behaviors of, 246–247
- styles, applying to UI, 674–685
- subtraction (-) operator, 59
- subtractValues method, 66
- summary functions, invoking, 524
- summary methods, using with Select method, 520–521. *See also* methods
- support and errata, xxxv
- Swagger generator, 733–734

## switch expressions, using with pattern matching

- switch expressions, using with pattern matching, 129–131
- switch statements, using, 124–129, 225–226
- symmetric operators, creating, 540–541, 553.
  - See *also* operators
- synchronization, canceling, 627–628
- synchronization primitives, coordinating tasks, 625–627
- synchronizing
  - access to shared pool of resources, 640
  - concurrent access to data, 621–624
  - tasks and implementing thread-safe access, 640
  - threads to wait for events, 640
- Synchronous I/O anti-pattern, 600
- syntax, defined, 46
- System namespace, 22
- system requirements, xxix–xxx
- System.Array class, 256, 469
- System.Collections.Generic namespace, 418, 442, 445
- System.Console class, 23
- System.Diagnostics.Stopwatch object, watch variable, 568
- System.IComparable and System.IComparable<T> interfaces, 424–425
- System.Linq namespace, Enumerable class, 522–523
- System.Numerics namespace, Complex struct, 544–547
- System.Object class, 221, 292
- System.Random class, 252
- System.Runtime.dll assembly, 23. See *also* DLLs (dynamic-link libraries)
- System.Threading namespace, 625

## T

- tabular layout, implementing using Grid control, 659–667
- TabularHeaderStyle style, 680
- Task class
  - features of, 562
  - using to implement parallelism, 566–576, 622

- task exceptions, handing using
  - AggregateException class, 594–596
- Task List window, using, 208
- Task objects
  - increasing CPU use, 576
  - using with GraphDemo application, 573–576
- Task type, using, 601
- Task<TResult> class, 609
- TaskContinuationOptions type, 565
- TaskCreationOptions enumeration, 564–565
- tasks. See *also* canceled tasks; cooperative cancellation; faulted tasks; multitasking using Microsoft .NET; parallel tasks
  - abstracting using Parallel class, 576–580
  - cancellation token, 582–584
  - confirming cancellation, 588–589
  - controlling, 563–565
  - cooperative cancellation, 582–594
  - creating, 563–565
  - creating and running, 597
  - creating continuations, 564
  - displaying status of, 590–592
  - enabling cancellation, 598
  - handling exceptions, 598
  - memory allocation and efficiency, 613–614
  - running, 563–565
  - scheduling, 564
  - specifying methods for, 597
  - synchronization primitives for coordination of, 625–627
  - synchronizing for thread-safe access, 640
  - threads and ThreadPool, 562–563
  - waiting for finishing, 597
- TemperatureMonitor class, 502
- TestHello, Solution Explorer, 14, 22
- text file viewer, creating, 136
- TextBlock controls
  - Grid.Row attribute, 662
  - vs. TextBox control, 30, 32, 34
  - using with Visual State Manager, 671
- TextBox controls
  - data binding, 704–705
  - using, 601
  - using with Visual State Manager, 671
  - UWP app, 651–657

thread safety  
 adding to methods in collection classes, 629  
 and Dispose method, 357–358

threads. *See also* asynchronous methods  
 and garbage collection, 344–345  
 making wait for events, 640  
 synchronizing, 640  
 and ThreadPool, 562–563  
 using to calculate pi, 631–634

thread-safe collection, 636. *See also*  
 collection classes

Thread.Sleep method, 582

throw and switch statements, 126

throw exceptions, using, 174–175

tilde (~), using with finalizers, 341

Time struct, 238–241

Tips, explained, xxviii

title ComboBox controls, implementing data  
 binding for, 700–701

// TODO: comments, 190, 208, 307–308, 617,  
 619–620. *See also* commenting code

Toolbox, showing and hiding, 31

ToString method, 243–244, 268, 296, 312,  
 390, 525

Tree<TItem> class  
 adding enumerator to, 480–481  
 creating, 426–430  
 IEnumerable<TItem> interface, 475–476  
 testing, 430–433  
 using iterator with, 479–481

Tree<TItem> objects, querying data in,  
 525–530

TreeEnumerator class, creating, 470–474

triangle, modeling, 383–385

Triangle public class, 383–384

try block  
 calculating pi, 632–633  
 writing, 154–155, 592–593

try/catch statement block, 159–163, 506.  
*See also* catching exceptions

tuples and methods, 80–82

type inference and new operator, 184

## U

UI (user interface)  
 applying styles to, 674–685  
 creating, 29–36  
 error reporting, 752–755  
 long-running operations, 752–755  
 updating, 766–767  
 updating for ViewModel, 766–767  
 windows in, 28

uint keyword, type and structure, 237

Universal Windows Platform (UWP)  
 device families, 642  
 WinUI 3.0 and Win32 API, 642

ulong keyword, type and structure, 237

unary operators, 69, 538

unassigned local variables, 51

unhandled exceptions, 155–156, 159–163,  
 173–174

user interface (UI)  
 applying styles to, 674–685  
 creating, 29–36  
 error reporting, 752–755  
 long-running operations, 752–755  
 updating, 766–767  
 updating for ViewModel, 766–767  
 windows in, 28

user-defined conversion operators, 552

user-interface events, 503–504

ushort keyword, type and structure, 237

using directive  
 explained, 22–23  
 and IDisposable interface, 347–348  
 static keyword in, 200–202  
 thread safety and Dispose method, 358  
 ViewModel, 708

Util class, 279

Utils.Sum method, testing, 285

UWP (Universal Windows Platform)  
 device families, 642  
 WinUI 3.0 and Win32 API, 642

UWP apps. *See also* Adventure Works  
 Customers app; REST web service;  
 web services  
 adapting layout, 667–674  
 adding buttons to views, 713

UWP apps (*continued*)

- adding commands to ViewModel, 706–715
- adding More button, 759–761
- AdventureWorks REST web service
  - operations, 736–744
- AdventureWorks web API project, 729–730
- applying styles to UI, 674–685
- Azure SQL Database server, 719–724
- building with Blank App template, 645–649
- capabilities supported, 645
- connecting to, 719–723
- vs. Console apps, 96
- creating, 44, 686
- creating entity models, 728–734
- creating ViewModel, 702–706
- custom styles, 686
- Customers entity model, 730–734
- data binding, 672, 688–694, 699–701
- deploying web service to cloud, 746–751
- displaying customers, 759–761
- displaying data, 688–694
- enhanced searching, 745–746
- examining, 37–39
- features of, xxvi, 643–645
- form factors and orientations, 661–665
- gestures, 643
- implementing enhanced searching, 745–746
- lifetime, 643–644
- Model-View-ViewModel pattern, 687–688
- nameof operator, 699
- overview, 641–642
- Package.appxmanifest file, 644
- packaging, 644
- pages in, 28
- partial classes, 188
- removing unwanted columns, 724–728
- REST web service, 735–751
- restricted operations, 644
- retrieving block of customers, 744–745
- retrieving data from databases, 717–728
- running in debug mode, 36
- scalable user interface, 649–659, 686
- searching for data, 761–767
- structures created in, 388
- tabular layout with Grid control, 659–667

- testing, 666–667
- updating to use web services, 752–761
- uploading to Windows Store, 644
- user interface, 686
- Visual State Manager, 667–674
- Windows Runtime, 336

**V**

- value parameters, using, 208–201, 208–211
- value type variables, copying, 205–211, 229
- value types, explained, 205
- values. *See also* numeric values
  - assigning to variables, 132, 216
  - boxing and unboxing, 230
  - changing for variables, 133–134
  - comparing for expressions, 132
  - determining equivalency, 132
  - returning from methods, 80–82, 109
  - in variables, 48, 72
- ValueTask generic type, 614
- var keyword
  - and methods, 74
  - using with LINQ, 518
- variables. *See also* implicitly typed local variables
  - adding amounts to, 151
  - vs. arrays, 251
  - assigning values to, 72, 132, 216
  - camelCase notation, 48
  - changing values of, 72, 133–134
  - declaring, 48–49, 72, 206
  - and fields, 83–84
  - identifying as pointers, 227
  - incrementing and decrementing, 69–70, 72
  - initializing, 211
  - in methods, 185
  - naming, 48, 369
  - and scope, 82
  - specifying numeric values, 49–50
  - subtracting amounts from, 151
- variables defined with type parameter,
  - initializing, 475
- Variant type, 71
- Vehicles solution, opening, 301

- video card requirement, xxx
  - View method, creating, 765
  - ViewModel
    - adding commands to, 706–715
    - browse and search modes, 762–766
    - Command class, 706–709
    - control, 702–705
    - Next and Previous buttons, 713–715
    - NextCustomer command, 710–715
    - PreviousCustomer command, 710–715
    - XAML markup, 706
  - views
    - adding buttons to, 713
    - and models, 702
  - virtual keyword, 337, 375
  - virtual methods. *See also* methods
    - declaring, 296–297, 310
    - and polymorphism, 298–299
  - Visual State Manager, using to adapt layout, 667–674
  - Visual Studio 2022
    - asterisk (\*) after file name, 19
    - building and running console application, 18–21
    - Code and Text Editor window, 13
    - configuring project dialog, 11–12
    - Console Application template, 12–13
    - console application, 10–14
    - console applications, 44
    - Design View window, 27
    - Error List window, 19
    - graphical applications, 25–28
    - Microsoft IntelliSense, 15–17
    - overflow checking, 165
    - overview, 9–14
    - Solution Explorer pane, 13–14, 20–21
    - Start Debugging command, 19
    - user interface, 29–36
    - UWP app, 44
    - writing programs, 14–21
  - Visual Studio Community 2022 requirement, xxix
  - Visual Studio Debugger. *See also* Debug menu; methods
    - amending code, 94–96
    - breakpoints, 94
    - calculating factorials, 98–100
    - Continue button, 93
    - nesting methods, 97–100
    - refactoring code, 96–97
    - stepping through methods, 89–93
  - Visual Studio debugger, exception handling with, 169
  - Vlissides, John, 500
- ## W
- Wait method, using with tasks, 594, 602
  - wait operation, canceling, 640
  - watch variable, System.Diagnostics.Stopwatch object, 568
  - web services. *See also* REST web service; UWP apps
    - building, 718
    - creating in ASP.NET Core web API project, 735
    - DELETE request, 768
    - deploying to cloud, 746–751
    - POST request, 768
    - and relational database, 718
    - updating UWP apps for, 752–761
    - updating UWP apps for, 752–761
  - Where method, using with LINQ, 518–519
  - where operator, using to filter data, 523
  - while loop
    - creating, 138–139
    - rephrasing as for loop, 141
    - variable if in, 135
  - while statements
    - structure, 140
    - writing, 135–140
  - Wide World Importers application, 491–495
  - widening conversion, 551
  - Windows 10 requirement, xxix
  - Windows application, using indexers in, 404–410
  - Windows Forms app, 43. *See also* forms
  - Windows Presentation Foundation (WPF)
    - graphical app, 43
  - Windows Runtime APIs, asynchronous methods, 611–613



## Windows Runtime (WinRT)

- Windows Runtime (WinRT)
  - compatibility with, 336–337
  - hill-climbing algorithm, 563
  - iterative strategy, 563
  - overview, 641
  - tasks and schedules, 562
- WinU13 graphical application, 42
- WPF (Windows Presentation Foundation)
  - graphical app, 43
- WrappedInt object, 211
- Wrapper struct, 402–403
- write access, making exclusive, 640
- WriteLine function, 4
- WriteLine method, 16–17
- write-only properties, 371–372, 378, 383, 393.
  - See also* properties
- writing
  - and calling instance methods, 194–196
  - checked expressions, 166–168
  - checked statements, 166
  - constructors, 188–194
  - conversion operators, 553–555
  - and creating objects, 189–194
  - do statements, 142–150
  - finalizers, 340–342, 361
  - if statements, 119–123
  - indexers, 406–407
  - InsertIntoTree method, 434–436
  - methods, 85–89
  - params array method, 283–284
  - private and static keywords, 201
  - for statements, 140–142
  - static members, 200–202
  - try block, 592–593
  - try/catch statement block, 159–163
  - while statements, 135–140

## X

- XAML (Extensible Application Markup Language), 28–29, 31, 33, 96
- XAML markup
  - Next and Previous buttons, 714
  - UWP app, 647–648, 650, 652, 654–657, 662–665, 668–673, 675–676, 678–681, 685
  - ViewModel, 706
- XML representation, mapping characters to, 126–129
- XOR (^) operator, 398