

Microsoft Visual C# Step by Step

Eighth Edition



 Professional

John Sharp

Microsoft Visual C# Step by Step, 8th Edition

JOHN SHARP

PUBLISHED BY

Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2015 by CM Group, Ltd. All rights reserved.

No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2015940217
ISBN: 978-1-5093-0104-1

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at <http://aka.ms/tellpress>.

This book is provided “as-is” and expresses the author’s views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at www.microsoft.com on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Acquisitions and Developmental Editor: Devon Musgrave

Project Editor: John Pierce

Editorial Production: Rob Nance and John Pierce

Technical Reviewer: Marc Young

Copyeditor: John Pierce

Indexer: Christina Yeager, Emerald Editorial Services

Cover: Twist Creative • Seattle and Joel Panchot

Contents at a glance

Introduction

xix

PART I INTRODUCING MICROSOFT VISUAL C# AND MICROSOFT VISUAL STUDIO 2015

CHAPTER 1	Welcome to C#	3
CHAPTER 2	Working with variables, operators, and expressions	33
CHAPTER 3	Writing methods and applying scope	59
CHAPTER 4	Using decision statements	87
CHAPTER 5	Using compound assignment and iteration statements	107
CHAPTER 6	Managing errors and exceptions	127

PART II UNDERSTANDING THE C# OBJECT MODEL

CHAPTER 7	Creating and managing classes and objects	153
CHAPTER 8	Understanding values and references	177
CHAPTER 9	Creating value types with enumerations and structures	201
CHAPTER 10	Using arrays	221
CHAPTER 11	Understanding parameter arrays	243
CHAPTER 12	Working with inheritance	255
CHAPTER 13	Creating interfaces and defining abstract classes	277
CHAPTER 14	Using garbage collection and resource management	305

PART III DEFINING EXTENSIBLE TYPES WITH C#

CHAPTER 15	Implementing properties to access fields	329
CHAPTER 16	Using indexers	353
CHAPTER 17	Introducing generics	369
CHAPTER 18	Using collections	399
CHAPTER 19	Enumerating collections	423
CHAPTER 20	Decoupling application logic and handling events	439
CHAPTER 21	Querying in-memory data by using query expressions	469
CHAPTER 22	Operator overloading	493

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

CHAPTER 23	Improving throughput by using tasks	517
CHAPTER 24	Improving response time by performing asynchronous operations	559
CHAPTER 25	Implementing the user interface for a Universal Windows Platform app	601
CHAPTER 26	Displaying and searching for data in a Universal Windows Platform app	651
CHAPTER 27	Accessing a remote database from a Universal Windows Platform app	697
	<i>Index</i>	749

Associativity and the assignment operator	53
Incrementing and decrementing variables	54
Prefix and postfix	55
Declaring implicitly typed local variables	56
Summary	57
Quick Reference	58
Chapter 3 Writing methods and applying scope	59
Creating methods	59
Declaring a method	60
Returning data from a method	61
Using expression-bodied methods	62
Calling methods	63
Applying scope	66
Defining local scope	66
Defining class scope	67
Overloading methods	68
Writing methods	68
Using optional parameters and named arguments	77
Defining optional parameters	79
Passing named arguments	79
Resolving ambiguities with optional parameters and named arguments	80
Summary	85
Quick reference	86
Chapter 4 Using decision statements	87
Declaring Boolean variables	87
Using Boolean operators	88
Understanding equality and relational operators	88
Understanding conditional logical operators	89
Short circuiting	90
Summarizing operator precedence and associativity	90

Using <i>if</i> statements to make decisions	91
Understanding <i>if</i> statement syntax	91
Using blocks to group statements	93
Cascading <i>if</i> statements	94
Using <i>switch</i> statements	99
Understanding <i>switch</i> statement syntax	100
Following the <i>switch</i> statement rules	101
Summary	104
Quick reference	105

Chapter 5 Using compound assignment and iteration statements 107

Using compound assignment operators	107
Writing <i>while</i> statements	108
Writing <i>for</i> statements	114
Understanding <i>for</i> statement scope	115
Writing <i>do</i> statements	116
Summary	125
Quick reference	125

Chapter 6 Managing errors and exceptions 127

Coping with errors	127
Trying code and catching exceptions	128
Unhandled exceptions	129
Using multiple catch handlers	130
Catching multiple exceptions	131
Propagating exceptions	136
Using checked and unchecked integer arithmetic	138
Writing checked statements	139
Writing checked expressions	140
Throwing exceptions	143
Using a <i>finally</i> block	148

Summary.....	149
Quick reference.....	150

PART II UNDERSTANDING THE C# OBJECT MODEL

Chapter 7 Creating and managing classes and objects 153

Understanding classification.....	153
The purpose of encapsulation	154
Defining and using a class.....	154
Controlling accessibility.....	156
Working with constructors.....	157
Overloading constructors.....	158
Understanding static methods and data	167
Creating a shared field	168
Creating a static field by using the <i>const</i> keyword.....	169
Understanding static classes	169
Static <i>using</i> statements.....	170
Anonymous classes	172
Summary.....	174
Quick reference.....	174

Chapter 8 Understanding values and references 177

Copying value type variables and classes.....	177
Understanding null values and nullable types.....	183
Using nullable types	185
Understanding the properties of nullable types	186
Using <i>ref</i> and <i>out</i> parameters.....	187
Creating <i>ref</i> parameters	188
Creating <i>out</i> parameters.....	188
How computer memory is organized	190
Using the stack and the heap	192
The <i>System.Object</i> class	193
Boxing.....	194

Unboxing	194
Casting data safely	196
The <i>is</i> operator	196
The <i>as</i> operator	197
Summary.....	199
Quick reference.....	199

Chapter 9 Creating value types with enumerations and structures 201

Working with enumerations	201
Declaring an enumeration	202
Using an enumeration.....	202
Choosing enumeration literal values	203
Choosing an enumeration's underlying type	204
Working with structures	206
Declaring a structure.....	208
Understanding differences between structures and classes	209
Declaring structure variables	210
Understanding structure initialization	211
Copying structure variables	215
Summary.....	219
Quick reference.....	219

Chapter 10 Using arrays 221

Declaring and creating an array.....	221
Declaring array variables.....	221
Creating an array instance	222
Populating and using an array	223
Creating an implicitly typed array.....	224
Accessing an individual array element.....	225
Iterating through an array	225
Passing arrays as parameters and return values for a method ..	227
Copying arrays.....	228

Using multidimensional arrays	230
Creating jagged arrays	231
Summary.	241
Quick reference.	242
Chapter 11 Understanding parameter arrays	243
Overloading—a recap	243
Using array arguments.	244
Declaring a <i>params</i> array	245
Using <i>params object[]</i>	247
Using a <i>params</i> array.	249
Comparing parameter arrays and optional parameters	252
Summary.	254
Quick reference.	254
Chapter 12 Working with inheritance	255
What is inheritance?	255
Using inheritance	256
The <i>System.Object</i> class revisited	258
Calling base-class constructors	258
Assigning classes	259
Declaring new methods	261
Declaring virtual methods	262
Declaring <i>override</i> methods.	263
Understanding <i>protected</i> access	265
Understanding extension methods	271
Summary.	275
Quick reference.	276
Chapter 13 Creating interfaces and defining abstract classes	277
Understanding interfaces	277
Defining an interface.	278
Implementing an interface.	279

Referencing a class through its interface	280
Working with multiple interfaces	281
Explicitly implementing an interface	282
Interface restrictions	283
Defining and using interfaces	284
Abstract classes	293
Abstract methods.	295
Sealed classes.	295
Sealed methods	295
Implementing and using an abstract class	296
Summary.	302
Quick reference.	303

Chapter 14 Using garbage collection and resource management 305

The life and times of an object.	305
Writing destructors	306
Why use the garbage collector?	308
How does the garbage collector work?	310
Recommendations	310
Resource management	311
Disposal methods.	311
Exception-safe disposal.	312
The <i>using</i> statement and the <i>IDisposable</i> interface	312
Calling the <i>Dispose</i> method from a destructor.	314
Implementing exception-safe disposal	316
Summary.	325
Quick reference.	325

PART III DEFINING EXTENSIBLE TYPES WITH C#

Chapter 15 Implementing properties to access fields 329

Implementing encapsulation by using methods.	329
What are properties?	331

Using properties.	333
Read-only properties.	334
Write-only properties	334
Property accessibility.	335
Understanding the property restrictions	336
Declaring interface properties	337
Replacing methods with properties	339
Generating automatic properties.	343
Initializing objects by using properties.	345
Summary.	349
Quick reference.	350

Chapter 16 Using indexers 353

What is an indexer?	353
An example that doesn't use indexers	353
The same example using indexers	355
Understanding indexer accessors	357
Comparing indexers and arrays.	358
Indexers in interfaces	360
Using indexers in a Windows application.	361
Summary.	367
Quick reference.	368

Chapter 17 Introducing generics 369

The problem with the <i>object</i> type	369
The generics solution	373
Generics vs. generalized classes	375
Generics and constraints.	375
Creating a generic class	376
The theory of binary trees	376
Building a binary tree class by using generics	379
Creating a generic method	389

Defining a generic method to build a binary tree	389
Variance and generic interfaces	391
Covariant interfaces	393
Contravariant interfaces	395
Summary	397
Quick reference	397

Chapter 18 Using collections 399

What are collection classes?	399
The <i>List</i> < <i>T</i> > collection class	401
The <i>LinkedList</i> < <i>T</i> > collection class	403
The <i>Queue</i> < <i>T</i> > collection class	404
The <i>Stack</i> < <i>T</i> > collection class	405
The <i>Dictionary</i> < <i>TKey</i> , <i>TValue</i> > collection class	407
The <i>SortedList</i> < <i>TKey</i> , <i>TValue</i> > collection class	408
The <i>HashSet</i> < <i>T</i> > collection class	409
Using collection initializers	411
The <i>Find</i> methods, predicates, and lambda expressions	411
The forms of lambda expressions	413
Comparing arrays and collections	415
Using collection classes to play cards	416
Summary	420
Quick reference	420

Chapter 19 Enumerating collections 423

Enumerating the elements in a collection	423
Manually implementing an enumerator	425
Implementing the <i>IEnumerable</i> interface	429
Implementing an enumerator by using an iterator	431
A simple iterator	432
Defining an enumerator for the <i>Tree</i> < <i>TItem</i> > class by using an iterator	434

Summary.....	436
Quick reference.....	437

Chapter 20 Decoupling application logic and handling events 439

Understanding delegates	440
Examples of delegates in the .NET Framework class library.....	441
The automated factory scenario	443
Implementing the factory control system without using delegates.....	443
Implementing the factory by using a delegate	444
Declaring and using delegates	447
Lambda expressions and delegates.....	455
Creating a method adapter	455
Enabling notifications by using events	456
Declaring an event.....	456
Subscribing to an event.....	457
Unsubscribing from an event.....	457
Raising an event.....	458
Understanding user interface events	458
Using events	460
Summary.....	466
Quick reference.....	466

Chapter 21 Querying in-memory data by using query expressions 469

What is LINQ?	469
Using LINQ in a C# application	470
Selecting data.....	472
Filtering data.....	474
Ordering, grouping, and aggregating data	475
Joining data.....	477
Using query operators.....	479
Querying data in <i>Tree<TItem></i> objects.....	481
LINQ and deferred evaluation.....	487

Summary.....	491
Quick reference.....	491

Chapter 22 Operator overloading 493

Understanding operators	493
Operator constraints	494
Overloaded operators.....	494
Creating symmetric operators.....	496
Understanding compound assignment evaluation.....	498
Declaring increment and decrement operators	499
Comparing operators in structures and classes.....	500
Defining operator pairs	500
Implementing operators	501
Understanding conversion operators.....	508
Providing built-in conversions.....	508
Implementing user-defined conversion operators	509
Creating symmetric operators, revisited	510
Writing conversion operators	511
Summary.....	513
Quick reference.....	514

PART IV BUILDING UNIVERSAL WINDOWS PLATFORM APPLICATIONS WITH C#

Chapter 23 Improving throughput by using tasks 517

Why perform multitasking by using parallel processing?	517
The rise of the multicore processor	518
Implementing multitasking by using the Microsoft .NET Framework... 519	
Tasks, threads, and the <i>ThreadPool</i>	520
Creating, running, and controlling tasks	521
Using the <i>Task</i> class to implement parallelism	524
Abstracting tasks by using the <i>Parallel</i> class.....	536
When not to use the <i>Parallel</i> class	541

Canceling tasks and handling exceptions.	543
The mechanics of cooperative cancellation	543
Using continuations with canceled and faulted tasks	556
Summary.	557
Quick reference.	557

Chapter 24 Improving response time by performing asynchronous operations 559

Implementing asynchronous methods	560
Defining asynchronous methods: The problem	560
Defining asynchronous methods: The solution	564
Defining asynchronous methods that return values	569
Asynchronous method gotchas.	570
Asynchronous methods and the Windows Runtime APIs.	572
Using PLINQ to parallelize declarative data access.	575
Using PLINQ to improve performance while iterating through a collection	576
Canceling a PLINQ query	580
Synchronizing concurrent access to data	581
Locking data	584
Synchronization primitives for coordinating tasks.	584
Canceling synchronization	587
The concurrent collection classes	587
Using a concurrent collection and a lock to implement thread-safe data access.	588
Summary.	598
Quick reference.	599

Chapter 25 Implementing the user interface for a Universal Windows Platform app 601

Features of a Universal Windows Platform app.	602
Using the Blank App template to build a Universal Windows Platform app.	605

Implementing a scalable user interface	607
Applying styles to a UI	638
Summary	649
Quick reference	649
Chapter 26 Displaying and searching for data in a Universal Windows Platform app	651
Implementing the Model-View-ViewModel pattern	651
Displaying data by using data binding.	652
Modifying data by using data binding.	659
Using data binding with a <i>ComboBox</i> control	663
Creating a ViewModel.	665
Adding commands to a ViewModel.	669
Searching for data using Cortana	680
Providing a vocal response to voice commands	692
Summary	695
Quick reference	696
Chapter 27 Accessing a remote database from a Universal Windows Platform app	697
Retrieving data from a database	698
Creating an entity model	703
Creating and using a REST web service	712
Inserting, updating, and deleting data through a REST web service. . .	728
Reporting errors and updating the UI	738
Summary	746
Quick reference	747
<i>Index</i>	749

Introduction

Microsoft Visual C# is a powerful but simple language aimed primarily at developers who create applications built on the Microsoft .NET Framework. Visual C# inherits many of the best features of C++ and Microsoft Visual Basic, but few of the inconsistencies and anachronisms, which results in a cleaner and more logical language. C# 1.0 made its public debut in 2001. With the advent of C# 2.0 with Visual Studio 2005, several important new features were added to the language, 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 facility, or LINQ. C# 4.0, was released in 2010 and 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. An important addition to the .NET Framework, and 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 is an incremental upgrade with features that are intended to make life simpler for developers. These features include 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. They are all described in this book.

Another important event for Microsoft is the launch of Windows 10. This new version of Windows combines the best (and most loved) aspects of previous versions of the operating system and 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 10 is Universal Windows Platform (UWP) apps—applications designed to run on any Windows 10 device, whether a fully fledged desktop system, a laptop, a tablet, a smartphone, or even an IoT (Internet of Things) device with limited resources. Once you have mastered the core features of C#, gaining the skills to build applications that can run on all these platforms is important.

Voice activation is another feature that has come to the fore, and Windows 10 includes Cortana, your personal voice-activated digital assistant. You can integrate your own apps with Cortana to allow them to participate in data searches and other operations. Despite the complexity normally associated with natural-language speech analysis, it is surprisingly easy to enable your apps to respond to Cortana's requests, and I cover this in Chapter 26. Additionally, 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 users smartphones, that I decided to focus on this aspect of development in the final chapter of the book.

The development environment provided by Visual Studio 2015 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 2015 and the .NET Framework version 4.6. 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 10 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 applications for Windows, such as ADO.NET, ASP.NET, Windows Communication Foundation, or Windows Workflow Foundation. If you require more information on any of these items, you might consider reading some of the other titles available from Microsoft Press.

Organization of this book

This book is divided into four sections:

- Part I, “Introducing Microsoft Visual C# and Microsoft Visual Studio 2015,” provides an introduction to the core syntax of the C# language and the Visual Studio programming environment.
- Part II, “Understanding the C# object model,” goes into detail on how to create and manage new types in C# and how to manage the resources referenced by these types.

- Part III, “Defining extensible types with C#,” includes extended coverage of the elements that C# provides for building types that you can reuse across multiple applications.
- Part IV, “Building Universal Windows Platform applications with C#,” describes the universal Windows 10 programming model and how you can use C# to build interactive applications for this new model.

Finding your best starting point in this book

This book is designed to help you build skills in a number of essential areas. You can use this book if you are new to programming or if you are 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"> 1. Install the practice files as described in the upcoming section, “Code samples.” 2. Work through the chapters in Parts I, II, and III sequentially. 3. Complete Part IV as your level of experience and interest dictates.
Familiar with procedural programming languages such as C but new to C#	<ol style="list-style-type: none"> 1. Install the practice files as described in the upcoming section, “Code samples.” 2. Skim the first five chapters to get an overview of C# and Visual Studio 2015, and then concentrate on Chapters 6 through 22. 3. Complete Part IV as your level of experience and interest dictates.
Migrating from an object-oriented language such as C++ or Java	<ol style="list-style-type: none"> 1. Install the practice files as described in the upcoming section, “Code samples.” 2. Skim the first seven chapters to get an overview of C# and Visual Studio 2015, and then concentrate on Chapters 8 through 22. 3. For information about building Universal Windows Platform applications, read Part IV.

If you are	Follow these steps
Switching from Visual Basic to C#	<ol style="list-style-type: none"> 1. Install the practice files as described in the upcoming section, "Code samples." 2. Work through the chapters in Parts I, II, and III sequentially. 3. For information about building Universal Windows Platform applications, read Part IV. 4. Read the Quick Reference sections at the end of the chapters for information about specific C# and Visual Studio 2015 constructs.
Referencing the book after working through the exercises	<ol style="list-style-type: none"> 1. Use the index or the table of contents to find information about particular subjects. 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.

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" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears 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 Professional (or above) edition.
- Visual Studio Community 2015 edition, Visual Studio Professional 2015 edition, or Visual Studio Enterprise 2015 edition.



Important You must install the Windows 10 developer tools with Visual Studio 2015.

- Computer that has a 1.6 GHz or faster processor (2 GHz recommended).
- 1 GB (32-bit) or 2 GB (64-bit) RAM (add 512 MB if running in a virtual machine).
- 10 GB of available hard disk space.
- 5400 RPM hard-disk drive.
- DirectX 9–capable video card running at 1024 × 768 or higher resolution display.
- DVD-ROM drive (if installing Visual Studio from a DVD).
- 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 2015.

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 preexercise and postexercise formats, from the following page:

<http://aka.ms/sharp8e/companioncontent>



Note In addition to the code samples, your system should have Visual Studio 2015 installed. If available, install the latest service packs for Windows and Visual Studio.

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 from which you downloaded the CSharpSBS.zip file.

Using the code samples

Each chapter in this book explains when and how to use any 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 have dependencies 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 may report a large number of errors for unresolved references. Building the project will cause these references to be resolved, and the errors should disappear.

For those of you who like to know all the details, here's a list of the sample Visual Studio 2015 projects and solutions, grouped by the folders where 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
Chapter 1	
TextHello	This project gets you started. It steps through the creation of a simple program that displays a text-based greeting.
Hello	This project opens a window that prompts the user for his or her name and then displays a greeting.
Chapter 2	
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.
MathsOperators	This program introduces the arithmetic operators (+ - * / %).
Chapter 3	
Methods	In this project, you'll reexamine the code in the MathsOperators 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.
Chapter 4	
Selection	This project shows you how to use a cascading <i>if</i> statement to implement complex logic, such as comparing the equivalence of two dates.
SwitchStatement	This simple program uses a <i>switch</i> statement to convert characters into their XML representations.
Chapter 5	
WhileStatement	This project demonstrates a <i>while</i> 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 <i>do</i> statement to convert a decimal number to its octal representation.

Project/Solution	Description
Chapter 6	
MathsOperators	This project revisits the MathsOperators project from Chapter 2 and shows how various unhandled exceptions can make the program fail. The <i>try</i> and <i>catch</i> keywords then make the application more robust so that it no longer fails.
Chapter 7	
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 <i>new</i> keyword and how to define static methods and fields.
Chapter 8	
Parameters	This program investigates the difference between value parameters and reference parameters. It demonstrates how to use the <i>ref</i> and <i>out</i> keywords.
Chapter 9	
StructsAndEnums	This project defines a <i>struct</i> type to represent a calendar date.
Chapter 10	
Cards	This project shows how to use arrays to model hands of cards in a card game.
Chapter 11	
ParamsArray	This project demonstrates how to use the <i>params</i> keyword to create a single method that can accept any number of <i>int</i> arguments.
Chapter 12	
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 <i>int</i> type, providing a method that converts an integer value from base 10 to a different number base.
Chapter 13	
Drawing	This project implements part of a graphical drawing package. The project uses interfaces to define the methods that drawing shapes expose and implement.
Drawing Using Interfaces	This project acts as a starting point for extending the Drawing project to factor common functionality for shape objects into abstract classes.

Project/Solution	Description
Chapter 14	
GarbageCollectionDemo	This project shows how to implement exception-safe disposal of resources by using the Dispose pattern.
Chapter 15	
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.
Chapter 16	
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.
Chapter 17	
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.
Chapter 18	
Cards	This project updates the code from Chapter 10 to show how to use collections to model hands of cards in a card game.
Chapter 19	
BinaryTree	This project shows you how to implement the generic <i>IEnumerator<T></i> interface to create an enumerator for the generic <i>Tree</i> class.
IteratorBinaryTree	This solution uses an iterator to generate an enumerator for the generic <i>Tree</i> class.
Chapter 20	
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
Chapter 21	
QueryBinaryTree	This project shows how to use LINQ queries to retrieve data from a binary tree object.

Project/Solution	Description
Chapter 22	
ComplexNumbers	This project defines a new type that models complex numbers and implements common operators for this type.
Chapter 23	
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 <i>Parallel</i> 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 <i>Parallel</i> class to create and run tasks.
Chapter 24	
GraphDemo	This is a version of the GraphDemo project from Chapter 23 that uses the <i>async</i> keyword and the <i>await</i> 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.
Chapter 25	
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.
Chapter 26	
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 <i>INotifyPropertyChanged</i> 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.
Cortana	This project integrates the Customers app with Cortana. A user can issue voice commands to search for customers by name.

Project/Solution	Description
Chapter 27	
Web Service	This solution includes a web application that provides an ASP.NET Web API web service that the Customers application uses to retrieve customer data from a SQL Server database. The web service uses an entity model created with the Entity Framework to access the database.

Acknowledgments

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 exercise.

First, Devon Musgrave at Microsoft Press, who awoke me from my interedition slumber. (I was actually quite busy writing material for Microsoft Patterns & Practices but managed to take a sabbatical to work on this edition of the book.) He prodded, cajoled, and generally made sure I was aware of the imminent arrival of Windows 10 and Visual Studio 2015, drew up the contract, and made sure that I signed it in blood, with agreed delivery dates!

Next, Jason Lee, my former underling and now immediate boss at Content Master (it's a somewhat complicated story, but he seems to have found some interesting photographic negatives I left lying carelessly around). He took on much of the initial donkey work generating new screen shots and making sure that the code for the first 20 or so chapters was updated (and corrected) for this edition. If there are any errors, I would like to point the finger at him, but of course any issues and mistakes are entirely my responsibility for missing them during review.

I also need to thank Marc Young, who had the rather tedious task of examining my code to make sure it stood a decent chance of compiling and running. His advice was extremely useful.

Of course, like many programmers, I might understand the technology, but my prose is not always as fluent or clear as it could be. I would like to show my gratitude to John Pierce for correcting my grammar, fixing my spelling, and generally making my material much easier to understand.

Finally, I must thank my long-suffering wife, Diana, who thought I was going slowly mad (maybe I was) when I started uttering peculiar phrases at my laptop to try to coax Cortana into understanding my application code. She thought I was continually on the phone to Orlando Gee (one of the sample customers used in the exercises toward the

end of the book) because I kept repeating his name quite loudly. Sadly, because of my accent, Cortana kept thinking I was asking for Orlando T, Orlando Key, or even Orlando Quay, so I subsequently changed the example to refer to Brian Johnson instead. At one point I overheard a conversation Diana was having with our decorator, Jason (who was painting our hallway at the time), about whether he would be able to convert one of the bedrooms into a padded cell, such was her concern with my state of mind! Still, that is now all water under the bridge, or “water under the breach” if you are Cortana trying to recognize my hybrid Gloucestershire/Kentish mode of speech.

And finally, finally, my daughter Francesca would be frightfully upset if I didn’t mention her. Although she still lives at home, she is all grown up and now works for a software development company in Cam, Gloucestershire (they didn’t offer me any freebies, so I am not going to mention their name).

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:

<http://aka.ms/sharp8e/errata>

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *msspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback is our most valuable asset. Please tell us what you think of this book at:

<http://aka.ms/tellpress>

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

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

Using decision statements

After completing this chapter, you will be able to:

- Declare Boolean variables.
- Use Boolean operators to create expressions whose outcome is either true or false.
- Write *if* statements to make decisions based on the result of a Boolean expression.
- Write *switch* statements to make more complex decisions.

Chapter 3, “Writing methods and applying scope,” shows how to group related statements into methods. It also demonstrates how to use parameters to pass information to a method and how to use *return* statements to pass information out of a method. Dividing a program into a set of discrete methods, each designed to perform a specific task or calculation, is a necessary design strategy. Many programs need to solve large and complex problems. Breaking up a program into methods helps you to understand these problems and focus on how to solve them, one piece at a time.

The methods in Chapter 3 are very straightforward, with each statement executing sequentially after the previous statement completes. However, to solve many real-world problems, you also need to be able to write code that selectively performs different actions and that takes different paths through a method depending on the circumstances. In this chapter, you’ll learn how to accomplish this task.

Declaring Boolean variables

In the world of C# programming (unlike in the real world), everything is black or white, right or wrong, true or false. For example, if you create an integer variable called *x*, assign the value 99 to it, and then ask whether *x* contains the value 99, the answer is definitely true. If you ask if *x* is less than 10, the answer is definitely false. These are examples of *Boolean expressions*. A Boolean expression always evaluates to true or false.



Note The answers to these questions are not necessarily definitive for all other programming languages. An unassigned variable has an undefined value, and you cannot, for example, say that it is definitely less than 10. Issues such as this one are a common source of errors in C and C++ programs. The Microsoft Visual C# compiler solves this problem by ensuring that you always assign a value to a variable before examining it. If you try to examine the contents of an unassigned variable, your program will not compile.

Visual C# provides a data type called *bool*. A *bool* variable can hold one of two values: *true* or *false*. For example, the following three statements declare a *bool* variable called *areYouReady*, assign *true* to that variable, and then write its value to the console:

```
bool areYouReady;  
areYouReady = true;  
Console.WriteLine(areYouReady); // writes True to the console
```

Using Boolean operators

A Boolean operator is an operator that performs a calculation whose result is either true or false. C# has several very useful Boolean operators, the simplest of which is the *NOT* operator, represented by the exclamation point (!). The ! operator negates a Boolean value, yielding the opposite of that value. In the preceding example, if the value of the variable *areYouReady* is true, the value of the expression *!areYouReady* is false.

Understanding equality and relational operators

Two Boolean operators that you will frequently use are equality (==) and inequality (!=). These are binary operators with which you can determine whether one value is the same as another value of the same type, yielding a Boolean result. The following table summarizes how these operators work, using an *int* variable called *age* as an example.

Operator	Meaning	Example	Outcome if age is 42
==	Equal to	age == 100	false
!=	Not equal to	age != 0	true

Don't confuse the *equality* operator == with the *assignment* operator =. The expression *x=y* compares *x* with *y* and has the value *true* if the values are the same. The expression *x=y* assigns the value of *y* to *x* and returns the value of *y* as its result.

Closely related to == and != are the *relational* operators. You use these operators to find out whether a value is less than or greater than another value of the same type. The following table shows how to use these operators.

Operator	Meaning	Example	Outcome if age is 42
<	Less than	age < 21	false
<=	Less than or equal to	age <= 18	false
>	Greater than	age > 16	true
>=	Greater than or equal to	age >= 30	true

Understanding conditional logical operators

C# also provides two other binary Boolean operators: the logical AND operator, which is represented by the `&&` symbol, and the logical OR operator, which is represented by the `||` symbol. Collectively, these are known as the *conditional logical operators*. Their purpose is to combine two Boolean expressions or values into a single Boolean result. These operators are similar to the equality and relational operators in that the value of the expressions in which they appear is either true or false, but they differ in that the values on which they operate must also be either true or false.

The outcome of the `&&` operator is *true* if and only if both of the Boolean expressions it's evaluating are *true*. For example, the following statement assigns the value *true* to *validPercentage* if and only if the value of *percent* is greater than or equal to 0 and the value of *percent* is less than or equal to 100:

```
bool validPercentage;
validPercentage = (percent >= 0) && (percent <= 100);
```



Tip A common beginner's error is to try to combine the two tests by naming the *percent* variable only once, like this:

```
percent >= 0 && <= 100 // this statement will not compile
```

Using parentheses helps to avoid this type of mistake and also clarifies the purpose of the expression. For example, compare

```
validPercentage = percent >= 0 && percent <= 100
```

and

```
validPercentage = (percent >= 0) && (percent <= 100)
```

Both expressions return the same value because the precedence of the `&&` operator is less than that of `>=` and `<=`. However, the second expression conveys its purpose in a more readable manner.

The outcome of the `||` operator is *true* if either of the Boolean expressions it evaluates is *true*. You use the `||` operator to determine whether any one of a combination of Boolean expressions is *true*. For example, the following statement assigns the value *true* to *invalidPercentage* if the value of *percent* is less than 0 or the value of *percent* is greater than 100:

```
bool invalidPercentage;  
invalidPercentage = (percent < 0) || (percent > 100);
```

Short circuiting

The `&&` and `||` operators both exhibit a feature called *short circuiting*. Sometimes, it is not necessary to evaluate both operands when ascertaining the result of a conditional logical expression. For example, if the left operand of the `&&` operator evaluates to *false*, the result of the entire expression must be *false*, regardless of the value of the right operand. Similarly, if the value of the left operand of the `||` operator evaluates to *true*, the result of the entire expression must be *true*, irrespective of the value of the right operand. In these cases, the `&&` and `||` operators bypass the evaluation of the right operand. Here are some examples:

```
(percent >= 0) && (percent <= 100)
```

In this expression, if the value of *percent* is less than 0, the Boolean expression on the left side of `&&` evaluates to *false*. This value means that the result of the entire expression must be *false*, and the Boolean expression to the right of the `&&` operator is not evaluated.

```
(percent < 0) || (percent > 100)
```

In this expression, if the value of *percent* is less than 0, the Boolean expression on the left side of `||` evaluates to *true*. This value means that the result of the entire expression must be *true*, and the Boolean expression to the right of the `||` operator is not evaluated.

If you carefully design expressions that use the conditional logical operators, you can boost the performance of your code by avoiding unnecessary work. Place simple Boolean expressions that can be evaluated easily on the left side of a conditional logical operator, and put more complex expressions on the right side. In many cases, you will find that the program does not need to evaluate the more complex expressions.

Summarizing operator precedence and associativity

The following table summarizes the precedence and associativity of all the operators you have learned about so far. Operators in the same category have the same precedence. The operators in categories higher up in the table take precedence over operators in categories lower down.

Category	Operators	Description	Associativity
Primary	<code>()</code> <code>++</code> <code>--</code>	Precedence override Post-increment Post-decrement	Left

Category	Operators	Description	Associativity
Unary	!	Logical NOT	Left
	+	Returns the value of the operand unchanged	
	-	Returns the value of the operand negated	
	++	Pre-increment	
	--	Pre-decrement	
Multiplicative	*	Multiply	Left
	/	Divide	
	%	Division remainder (modulus)	
Additive	+	Addition	Left
	-	Subtraction	
Relational	<	Less than	Left
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
Equality	==	Equal to	Left
	!=	Not equal to	
Conditional AND	&&	Conditional AND	Left
Conditional OR		Conditional OR	Left
Assignment	=	Assigns the right-hand operand to the left and returns the value that was assigned	Right

Notice that the && operator and the || operator have a different precedence: && is higher than ||.

Using *if* statements to make decisions

In a method, when you want to choose between executing two different statements depending on the result of a Boolean expression, you can use an *if* statement.

Understanding *if* statement syntax

The syntax of an *if* statement is as follows (*if* and *else* are C# keywords):

```
if ( booleanExpression )
    statement-1;
else
    statement-2;
```

If *booleanExpression* evaluates to *true*, *statement-1* runs; otherwise, *statement-2* runs. The *else* keyword and the subsequent *statement-2* are optional. If there is no *else* clause and the *booleanExpression* is *false*, execution continues with whatever code follows the *if* statement. Also,

notice that the Boolean expression must be enclosed in parentheses; otherwise, the code will not compile.

For example, here's an *if* statement that increments a variable representing the second hand of a stopwatch. (Minutes are ignored for now.) If the value of the *seconds* variable is 59, it is reset to 0; otherwise, it is incremented by using the ++ operator:

```
int seconds;
...
if (seconds == 59)
    seconds = 0;
else
    seconds++;
```

Boolean expressions only, please!

The expression in an *if* statement must be enclosed in parentheses. Additionally, the expression must be a Boolean expression. In some other languages—notably C and C++—you can write an integer expression, and the compiler will silently convert the integer value to *true* (nonzero) or *false* (0). C# does not support this behavior, and the compiler reports an error if you write such an expression.

If you accidentally specify the assignment operator (=) instead of the equality test operator (==) in an *if* statement, the C# compiler recognizes your mistake and refuses to compile your code, such as in the following example:

```
int seconds;
...
if (seconds = 59) // compile-time error
...
if (seconds == 59) // ok
```

Accidental assignments were another common source of bugs in C and C++ programs, which would silently convert the value assigned (59) to a Boolean expression (with anything nonzero considered to be true), with the result being that the code following the *if* statement would be performed every time.

Incidentally, you can use a Boolean variable as the expression for an *if* statement, although it must still be enclosed in parentheses, as shown in this example:

```
bool inWord;
...
if (inWord == true) // ok, but not commonly used
...
if (inWord) // more common and considered better style
```

Using blocks to group statements

Notice that the syntax of the *if* statement shown earlier specifies a single statement after the *if* (*booleanExpression*) and a single statement after the *else* keyword. Sometimes, you'll want to perform more than one statement when a Boolean expression is true. You could group the statements inside a new method and then call the new method, but a simpler solution is to group the statements inside a *block*. A block is simply a sequence of statements grouped between an opening brace and a closing brace.

In the following example, two statements that reset the *seconds* variable to 0 and increment the *minutes* variable are grouped inside a block, and the entire block executes if the value of *seconds* is equal to 59:

```
int seconds = 0;
int minutes = 0;
...
if (seconds == 59)
{
    seconds = 0;
    minutes++;
}
else
{
    seconds++;
}
```



Important If you omit the braces, the C# compiler associates only the first statement (*seconds = 0;*) with the *if* statement. The subsequent statement (*minutes++;*) will not be recognized by the compiler as part of the *if* statement when the program is compiled. Furthermore, when the compiler reaches the *else* keyword, it will not associate it with the previous *if* statement; instead, it will report a syntax error. Therefore, it is good practice to always define the statements for each branch of an *if* statement within a block, even if a block consists of only a single statement. It might save you some grief later if you want to add additional code.

A block also starts a new scope. You can define variables inside a block, but they will disappear at the end of the block. The following code fragment illustrates this point:

```
if (...)
{
    int myVar = 0;
    ... // myVar can be used here
} // myVar disappears here
else
{
    // myVar cannot be used here
    ...
}
// myVar cannot be used here
```


Cascading *if* statements

You can nest *if* statements inside other *if* statements. In this way, you can chain together a sequence of Boolean expressions, which are tested one after the other until one of them evaluates to *true*. In the following example, if the value of *day* is 0, the first test evaluates to *true* and *dayName* is assigned the string “*Sunday*”. If the value of *day* is not 0, the first test fails and control passes to the *else* clause, which runs the second *if* statement and compares the value of *day* with 1. The second *if* statement executes only if the first test is *false*. Similarly, the third *if* statement executes only if the first and second tests are *false*.

```
if (day == 0)
{
    dayName = "Sunday";
}
else if (day == 1)
{
    dayName = "Monday";
}
else if (day == 2)
{
    dayName = "Tuesday";
}
else if (day == 3)
{
    dayName = "Wednesday";
}
else if (day == 4)
{
    dayName = "Thursday";
}
else if (day == 5)
{
    dayName = "Friday";
}
else if (day == 6)
{
    dayName = "Saturday";
}
else
{
    dayName = "unknown";
}
```

In the following exercise, you’ll write a method that uses a cascading *if* statement to compare two dates.

Write *if* statements

1. Start Microsoft Visual Studio 2015 if it is not already running.
2. Open the Selection project, which is located in the \Microsoft Press\VCSBS\Chapter 4\Selection folder in your Documents folder.

3. On the Debug menu, click Start Debugging.

Visual Studio 2015 builds and runs the application. The form displays two *DatePicker* controls, called *firstDate* and *secondDate*. Both controls display the current date.

4. Click Compare.

The following text appears in the text box in the lower half of the window:

```
firstDate == secondDate : False
firstDate != secondDate : True
firstDate < secondDate : False
firstDate <= secondDate : False
firstDate > secondDate : True
firstDate >= secondDate : True
```

The Boolean expression, *firstDate* == *secondDate*, should be *true* because both *firstDate* and *secondDate* are set to the current date. In fact, only the less-than operator and the greater-than-or-equal-to operator seem to be working correctly. The following image shows the application running.



5. Return to Visual Studio 2015. On the Debug menu, click Stop Debugging.
6. Display the code for the *MainPage.xaml.cs* file in the Code and Text Editor window.
7. Locate the *compareClick* method, which should look like this:

```

private void compareClick(object sender, RoutedEventArgs e)
{
    int diff = dateCompare(firstDate.Date.LocalDateTime, secondDate.Date.LocalDateTime);
    info.Text = "";
    show("firstDate == secondDate", diff == 0);
    show("firstDate != secondDate", diff != 0);
    show("firstDate < secondDate", diff < 0);
    show("firstDate <= secondDate", diff <= 0);
    show("firstDate > secondDate", diff > 0);
    show("firstDate >= secondDate", diff >= 0);
}

```

This method runs whenever the user clicks the Compare button on the form. The expressions `firstDate.Date.LocalDateTime` and `secondDate.Date.LocalDateTime` hold *DateTime* values; they represent the dates displayed in the `firstDate` and `secondDate` controls on the form elsewhere in the application. The *DateTime* data type is just another data type, like *int* or *float*, except that it contains subelements with which you can access the individual pieces of a date, such as the year, month, or day.

The `compareClick` method passes the two *DateTime* values to the `dateCompare` method. The purpose of this method is to compare dates and return the *int* value 0 if they are the same, -1 if the first date is less than the second, and +1 if the first date is greater than the second. A date is considered greater than another date if it comes after it chronologically. You will examine the `dateCompare` method in the next step.

The `show` method displays the results of the comparison in the `info` text box control in the lower half of the form.

8. Locate the `dateCompare` method, which should look like this:

```

private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    // TO DO
    return 42;
}

```

This method currently returns the same value whenever it is called—rather than 0, -1, or +1—regardless of the values of its parameters. This explains why the application is not working as expected. You need to implement the logic in this method to compare two dates correctly.

9. Remove the `// TO DO` comment and the `return` statement from the `dateCompare` method.
10. Add the following statements shown in bold to the body of the `dateCompare` method:

```

private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    int result = 0;

    if (leftHandSide.Year < rightHandSide.Year)
    {
        result = -1;
    }
    else if (leftHandSide.Year > rightHandSide.Year)

```

```

    {
        result = 1;
    }
}

```



Note Don't try to build the application yet. The *dateCompare* method is not complete and the build will fail.

If the expression *leftHandSide.Year < rightHandSide.Year* is *true*, the date in *leftHandSide* must be earlier than the date in *rightHandSide*, so the program sets the *result* variable to *-1*. Otherwise, if the expression *leftHandSide.Year > rightHandSide.Year* is *true*, the date in *leftHandSide* must be later than the date in *rightHandSide*, and the program sets the *result* variable to *1*.

If the expression *leftHandSide.Year < rightHandSide.Year* is *false* and the expression *leftHandSide.Year > rightHandSide.Year* is also *false*, the *Year* property of both dates must be the same, so the program needs to compare the months in each date.

11. Add the following statements shown in bold to the body of the *dateCompare* method. Type them after the code you entered in the preceding step:

```

private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    ...
    else if (leftHandSide.Month < rightHandSide.Month)
    {
        result = -1;
    }
    else if (leftHandSide.Month > rightHandSide.Month)
    {
        result = 1;
    }
}

```

These statements compare months following a logic similar to that used to compare years in the preceding step.

If the expression *leftHandSide.Month < rightHandSide.Month* is *false* and the expression *leftHandSide.Month > rightHandSide.Month* is also *false*, the *Month* property of both dates must be the same, so the program finally needs to compare the days in each date.

12. Add the following statements to the body of the *dateCompare* method after the code you entered in the preceding two steps:

```

private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    ...
    else if (leftHandSide.Day < rightHandSide.Day)
    {
        result = -1;
    }
}

```

```

        else if (leftHandSide.Day > rightHandSide.Day)
        {
            result = 1;
        }
        else
        {
            result = 0;
        }

        return result;
    }
}

```

You should recognize the pattern in this logic by now.

If *leftHandSide.Day < rightHandSide.Day* and *leftHandSide.Day > rightHandSide.Day* both are *false*, the value in the *Day* properties in both variables must be the same. The *Month* values and the *Year* values must also be identical, respectively, for the program logic to have reached this point, so the two dates must be the same, and the program sets the value of *result* to *0*.

The final statement returns the value stored in the *result* variable.

13. On the Debug menu, click Start Debugging.

The application is rebuilt and runs.

14. Click Compare.

The following text appears in the text box:

```

firstDate == secondDate : True
firstDate != secondDate : False
firstDate < secondDate: False
firstDate <= secondDate: True
firstDate > secondDate: False
firstDate >= secondDate: True

```

These are the correct results for identical dates.

15. Use the *DatePicker* controls to select a later date for the second date and then click Compare.

The following text appears in the text box:

```

firstDate == secondDate: False
firstDate != secondDate: True
firstDate < secondDate: True
firstDate <= secondDate: True
firstDate > secondDate: False
firstDate >= secondDate: False

```

Again, these are the correct results when the first date is earlier than the second date.

16. Test some other dates, and verify that the results are as you would expect. Return to Visual Studio 2015 and stop debugging when you have finished.

Comparing dates in real-world applications

Now that you have seen how to use a rather long and complicated series of *if* and *else* statements, I should mention that this is not the technique you would employ to compare dates in a real-world application. If you look at the *dateCompare* method from the preceding exercise, you will see that the two parameters, *leftHandSide* and *rightHandSide*, are *DateTime* values. The logic you have written compares only the date part of these parameters, but they also contain a time element that you have not considered (or displayed). For two *DateTime* values to be considered equal, they should have not only the same date but also the same time. Comparing dates and times is such a common operation that the *DateTime* type actually has a built-in method called *Compare* for doing just that: it takes two *DateTime* arguments and compares them, returning a value indicating whether the first argument is less than the second, in which case the result will be negative; whether the first argument is greater than the second, in which case the result will be positive; or whether both arguments represent the same date and time, in which case the result will be 0.

Using *switch* statements

Sometimes when you write a cascading *if* statement, each of the *if* statements look similar because they all evaluate an identical expression. The only difference is that each *if* compares the result of the expression with a different value. For example, consider the following block of code that uses an *if* statement to examine the value in the *day* variable and work out which day of the week it is:

```
if (day == 0)
{
    dayName = "Sunday";
}
else if (day == 1)
{
    dayName = "Monday";
}
else if (day == 2)
{
    dayName = "Tuesday";
}
else if (day == 3)
{
    ...
}
else
{
    dayName = "Unknown";
}
```

Often in these situations, you can rewrite the cascading *if* statement as a *switch* statement to make your program more efficient and more readable.

Understanding *switch* statement syntax

The syntax of a *switch* statement is as follows (*switch*, *case*, and *default* are keywords):

```
switch ( controllingExpression )
{
    case constantExpression :
        statements
        break;
    case constantExpression :
        statements
        break;
    ...
    default :
        statements
        break;
}
```

The *controllingExpression*, which must be enclosed in parentheses, is evaluated once. Control then jumps to the block of code identified by the *constantExpression* whose value is equal to the result of the *controllingExpression*. (The *constantExpression* identifier is also called a *case label*.) Execution runs as far as the *break* statement, at which point the *switch* statement finishes and the program continues at the first statement that follows the closing brace of the *switch* statement. If none of the *constantExpression* values is equal to the value of the *controllingExpression*, the statements below the optional *default* label run.



Note Each *constantExpression* value must be unique so that the *controllingExpression* will match only one of them. If the value of the *controllingExpression* does not match any *constantExpression* value and there is no *default* label, program execution continues with the first statement that follows the closing brace of the *switch* statement.

So, you can rewrite the previous cascading *if* statement as the following *switch* statement:

```
switch (day)
{
    case 0 :
        dayName = "Sunday";
        break;
    case 1 :
        dayName = "Monday";
        break;
    case 2 :
        dayName = "Tuesday";
        break;
    ...
    default :
        dayName = "Unknown";
        break;
}
```

Following the *switch* statement rules

The *switch* statement is very useful, but unfortunately, you can't always use it when you might like to. Any *switch* statement you write must adhere to the following rules:

- You can use *switch* only on certain data types, such as *int*, *char*, or *string*. With any other types (including *float* and *double*), you must use an *if* statement.
- The *case* labels must be constant expressions, such as 42 if the *switch* data type is an *int*, '4' if the *switch* data type is a *char*, or "42" if the *switch* data type is a *string*. If you need to calculate your *case* label values at run time, you must use an *if* statement.
- The *case* labels must be unique expressions. In other words, two *case* labels cannot have the same value.
- You can specify that you want to run the same statements for more than one value by providing a list of *case* labels and no intervening statements, in which case the code for the final label in the list is executed for all cases in that list. However, if a label has one or more associated statements, execution cannot fall through to subsequent labels; in this case, the compiler generates an error. The following code fragment illustrates these points:

```
switch (trumps)
{
    case Hearts :
    case Diamonds :    // Fall-through allowed - no code between labels
        color = "Red";    // Code executed for Hearts and Diamonds
        break;
    case Clubs :
        color = "Black";
    case Spades :    // Error - code between labels
        color = "Black";
        break;
}
```



Note The *break* statement is the most common way to stop fall-through, but you can also use a *return* statement to exit from the method containing the *switch* statement or a *throw* statement to generate an exception and abort the *switch* statement. The *throw* statement is described in Chapter 6, "Managing errors and exceptions."

switch fall-through rules

Because you cannot accidentally fall through from one *case* label to the next if there is any intervening code, you can freely rearrange the sections of a *switch* statement without affecting its meaning (including the *default* label, which by convention is usually—but does not have to be—placed as the last label).

C and C++ programmers should note that the *break* statement is mandatory for every case

in a *switch* statement (even the default case). This requirement is a good thing—it is common in C or C++ programs to forget the *break* statement, allowing execution to fall through to the next label and leading to bugs that are difficult to spot.

If you really want to, you can mimic C/C++ fall-through in C# by using a *goto* statement to go to the following *case* or *default* label. Using *goto* in general is not recommended, though, and this book does not show you how to do it.

In the following exercise, you will complete a program that reads the characters of a string and maps each character to its XML representation. For example, the left angle bracket character (<) has a special meaning in XML (it's used to form elements). If you have data that contains this character, it must be translated into the text entity `<`; so that an XML processor knows that it is data and not part of an XML instruction. Similar rules apply to the right angle bracket (>), ampersand (&), single quotation mark ('), and double quotation mark (") characters. You will write a *switch* statement that tests the value of the character and traps the special XML characters as *case* labels.

Write *switch* statements

1. Start Visual Studio 2015 if it is not already running.
2. Open the SwitchStatement project, which is located in the \Microsoft Press\VCSBS\Chapter 4\SwitchStatement folder in your Documents folder.
3. On the Debug menu, click Start Debugging.

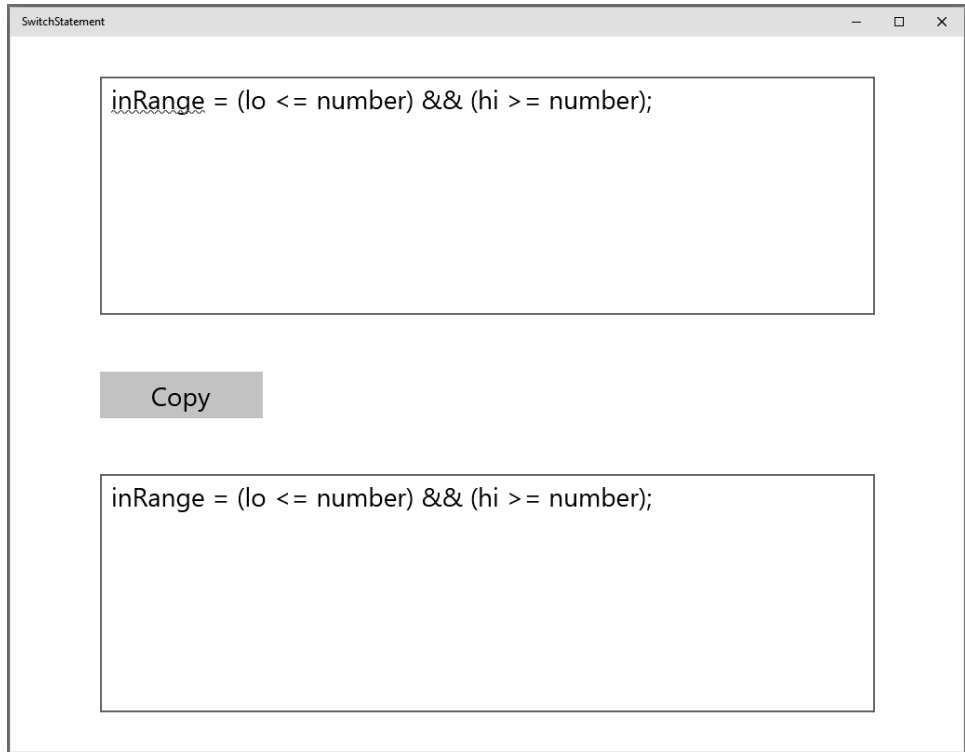
Visual Studio 2015 builds and runs the application. The application displays a form containing two text boxes separated by a Copy button.

4. Type the following sample text into the upper text box:

```
inRange = (lo <= number) && (hi >= number);
```

5. Click Copy.

The statement is copied verbatim into the lower text box, and no translation of the <, &, or > characters occurs, as shown in the following screen shot.



6. Return to Visual Studio 2015 and stop debugging.
7. Display the code for `MainPage.xaml.cs` in the Code and Text Editor window and locate the `copyOne` method.

The `copyOne` method copies the character specified as its input parameter to the end of the text displayed in the lower text box. At the moment, `copyOne` contains a *switch* statement with a single *default* action. In the following few steps, you will modify this *switch* statement to convert characters that are significant in XML to their XML mapping. For example, the `<` character will be converted to the string `<`.

8. Add the following statements shown in bold to the *switch* statement after the opening brace for the statement and directly before the *default* label:

```
switch (current)
{
    case '<' :
        target.Text += "&lt;";
        break;
    default:
        target.Text += current;
        break;
}
```

If the current character being copied is a left angle bracket (<), the preceding code appends the string "<" to the text being output in its place.

9. Add the following statements to the *switch* statement after the *break* statement you have just added and above the *default* label:

```
case '>' :
    target.Text += "&gt;";
    break;
case '&' :
    target.Text += "&amp;";
    break;
case '\"' :
    target.Text += "&#34;";
    break;
case '\\' :
    target.Text += "&#39;";
    break;
```



Note The single quotation mark (') and double quotation mark (") have a special meaning in C#—they are used to delimit character and string constants. The backslash (\) in the final two *case* labels is an escape character that causes the C# compiler to treat these characters as literals rather than as delimiters.

10. On the Debug menu, click Start Debugging.
11. Type the following text into the upper text box:
inRange = (lo <= number) && (hi >= number);

12. Click Copy.

The statement is copied into the lower text box. This time, each character undergoes the XML mapping implemented in the *switch* statement. The target text box displays the following text:

inRange = (lo <= number) && (hi >= number);

13. Experiment with other strings, and verify that all special characters (<, >, &, " , and ') are handled correctly.
14. Return to Visual Studio and stop debugging.

Summary

In this chapter, you learned about Boolean expressions and variables. You saw how to use Boolean expressions with the *if* and *switch* statements to make decisions in your programs, and you combined Boolean expressions by using the Boolean operators.

- If you want to continue to the next chapter, keep Visual Studio 2015 running and turn to Chapter 5, “Using compound assignment and iteration statements.”
- If you want to exit Visual Studio 2015 now, on the File menu, click Exit. If you see a Save dialog box, click Yes and save the project.

Quick reference

To	Do this	Example
Determine whether two values are equivalent	Use the == operator or the != operator.	answer == 42
Compare the value of two expressions	Use the <, <=, >, or >= operator.	age >= 21
Declare a Boolean variable	Use the <i>bool</i> keyword as the type of the variable.	bool inRange;
Create a Boolean expression that is true only if two conditions are both true	Use the && operator.	inRange = (lo <= number) && (number <= hi);
Create a Boolean expression that is true if either of two conditions is true	Use the operator.	outOfRange = (number < lo) (hi < number);
Run a statement if a condition is true	Use an <i>if</i> statement.	if (inRange) process();
Run more than one statement if a condition is true	Use an <i>if</i> statement and a block.	if (seconds == 59) { seconds = 0; minutes++; }
Associate different statements with different values of a controlling expression	Use a <i>switch</i> statement.	switch (current) { case 0: ... break; case 1: ... break; default : ... break; }

Index

Symbols

- & (ampersand)
 - & (AND) operator, 354
 - && (logical AND) operator, 89–90, 105
 - < > (angle brackets)
 - => operator, 62
 - => operator in lambda expressions, 413
 - > (greater than) operator, 89
 - >= (greater than or equal to) operator, 89
 - < (less than) operator, 89
 - <= (less than or equal to) operator, 89
 - * (asterisk), 11, 45, 198, 504
 - @ (at symbol), 178
 - \ (backslash), 104
 - { } (braces or curly brackets), 53, 60
 - :
 - \$ (dollar sign), 46
 - = (equal sign), 37, 53–54, 91
 - == (equal to) operator, 88–89, 505
 - associativity and, 53–54
 - ! (exclamation point)
 - != (inequality) operator, 505
 - ! (NOT operator), 88
 - / (forward slash), 11, 45, 504
 - (minus sign), 45, 503–504
 - (compound subtraction) operator, 108, 125
 - (decrement) operator, 55
 - (subtraction and assignment) operator, 457
 - () (parentheses), 53, 60, 413
 - in Boolean expressions, 92
 - in expressions, 89
 - in method calls, 64, 66
 - precedence override, 90
 - | (pipe symbol)
 - | (OR) operator, 354
 - || (logical OR) operator, 89–90
 - + (plus sign), 45, 54, 495, 503
 - ++ operator, 55
 - += (compound addition) operator, 108, 125
 - += (compound assignment) operator, 108, 125, 444, 457
 - ++ (increment) operator, 54–55, 499
 - with delegates, 444
 - ? (question mark)
 - ? : operator, 365–366
 - null-conditional operator, 184–186
 - % (remainder operator), 46
 - ;
 - in for statements, 115
 - in return statements, 61
 - in statements, 33
- [] (square bracket notation), 221, 357, 401, 407
- _ (underscore character), 36

A

- abstract classes, 258, 277, 293–295, 303
 - implementing, 296–300
- abstract keyword, 294
- abstract methods, 295, 303
- access modifiers
 - destructors and, 307
 - interfaces and, 283
- access to data, 581–598
- accessibility
 - class, 156–167
 - of properties, 335–336
- accessor methods, 330–331. *See also* get accessors; set accessors
 - for indexers, 357–358
- Action delegates, 443, 521–522, 563, 670–671
- Activated events, 685
- Adams, Douglas, 214

adapters

- adapters, 455
- Add method, 401, 409, 446
 - initializing collections, 411
- Add Scaffold wizard, 712–713, 728, 747
- AddAfter method, 403
- AddBefore method, 403
- AddCount method, 586
- AddFirst method, 403
- additive operators, 91
- AddLast method, 403
- AddParticipant method, 586
- addresses in memory, 177
- AddToAccumulator method, 542
- addValues method, 50–51, 62–64
- AggregateException class, 554–556, 558
- aggregating data, 475–477
- Allow Unsafe Code option, 198
- AND (&) operator, 354
- angle brackets (< >)
 - < (less than) operator, 89
 - <= (less than or equal to) operator, 89
 - for type parameters, 373
- anonymous arrays, 245
- anonymous classes, 172–173
- anonymous methods, 415
- anonymous types, 224–225
- “App capability declarations” page, 605
- App class, 686
- App.config (application configuration) file, 8
- application logic, 69
 - decoupling from applications, 439–467
- application window, 687
- applications
 - with asynchronous methods, 566–569
 - building, 12–13, 32
 - CIL instructions, 218
 - console, 3–17
 - deadlocks, 571–572
 - debugging, 13–14, 32
 - designing for multitasking, 519–520
 - fields vs. properties, 343–344
 - graphical, 17–32. *See also* Universal Windows Platform (UWP) apps
 - indexers in, 361–367
 - LINQ in, 470–472
 - multitasking, 517–519, 533–543. *See also* multitasking
 - parallel processing, 517, 594–596. *See also* parallelization; Task objects
 - partitioning into tasks, 521, 534–536.
 - See also* tasks
 - responsiveness, 559. *See also* responsiveness
 - running without debugging, 32, 73
 - single-threaded, 517, 590–594
 - solution files, 7
 - threads, 310
 - thread-safe, 596–597
- App.xaml file, 27
 - updating, 639–640
- App.xaml.cs file, 27–29
 - OnLaunched method, 29
- ArgumentException exceptions, 229, 239, 245, 249–250
- argumentList, 64
- ArgumentOutOfRangeException exceptions, 143–144, 229
- arguments
 - in arrays, 244–245
 - modifying, 187–190
 - omitting, 80
 - passing to methods, 187
 - references, 188
 - sequence, 79
 - variable numbers, 245–247. *See also* parameter arrays
- arithmetic operations, 47–48
 - on complex numbers, 502
 - types of results, 46
- arithmetic operators, 45–54
 - precedence, 53
- array instances, 228
 - copying, 229
 - creating, 222–223, 242
- array notation
 - for collection classes, 401–403
 - for collection lookups, 411
- array properties, 358–359
- array variables, 228
 - declaring, 221–222, 242
 - plural names, 222
- arrays, 227–228
 - accessing elements, 225, 242
 - anonymous, 245
 - arguments, 244–245
 - of arrays, 232
 - associative, 407
 - of bool variables, 356–357
 - vs. collections, 415
 - copying, 228–230, 359

- creating, 232–241
 - defined, 221
 - implicitly typed, 224–225
 - indexers, 353–368
 - indexes, 225, 235
 - initializing elements, 222–224, 242
 - inserting elements, 401
 - iterating, 225–227, 242
 - jagged, 231–232, 242
 - limitations, 399, 401
 - memory allocation, 222–223
 - multidimensional, 230–242
 - null elements, 237–238
 - number of elements, 225–226, 242
 - parameter, 243–254
 - as parameters, 227–228
 - passing as return values, 227
 - populating, 223–228, 239–240, 526, 533–534
 - rectangular, 231
 - removing elements, 237–238, 401
 - resizing, 401
 - retrieving specific data, 472–474
 - size, 222–223
 - storing items, 239
 - Art of Computer Programming, Volume 3: Sorting and Searching, The* (Knuth), 376
 - as operator, 197, 260–261
 - AsParallel method, 575, 577, 580, 599
 - ASP.NET Web API template, 705, 712–713, 747
 - ASP.NET Web client libraries, 748
 - assemblies, 7–8, 16–17, 301
 - adding to project references, 386
 - class libraries, 379
 - namespaces and, 17
 - referencing, 17
 - AssemblyInfo.cs file, 7
 - assigning
 - definite assignment rule, 38
 - objects to variables, 259–261
 - structure variables to structure variables, 215–216
 - assignment operator (=), 37, 91
 - associativity and, 53–54
 - assignment operators, compound, 107–108
 - associative arrays, 407
 - associativity, 53, 90–91, 493–494
 - assignment operator and, 53–54
 - of compound assignment operators, 108
 - asterisk (*), 11, 45, 198, 504
 - async modifier, 560, 599, 685
 - considerations, 570–572
 - AsyncCallback delegates, 574
 - asynchronicity, 560
 - asynchronous methods, 460, 560–575, 723
 - await operator, 547
 - defining, 560–572
 - implementing, 566–569, 599
 - for I/O operations, 573–574
 - parallelization, 565–566
 - return values, 569–570
 - Windows Runtime APIs and, 572–575
 - for writing to streams, 573–574
 - asynchronous operations, 559–600
 - enabling, 685
 - status, 574
 - attached properties, 623–624
 - automatic properties, 343–345, 351
 - object initializers, 347–349
 - Average method, 441–442
 - await operator, 547, 552, 556, 560, 564–566, 599
 - for asynchronous methods with return values, 570
 - considerations, 570–572
 - awaitable objects, 564
 - Azure, 697, 705
 - “Azure Active Directory” page, 705
 - Azure SQL Databases, 699
 - connections, 706–707, 710
 - creating, 699–701
 - removing columns, 702–703
- ## B
- background activation, 680
 - background images, 640–641
 - background threads, 561
 - backslash (\), 104
 - Barrier class, 586
 - base classes, 256, 276. *See also* inheritance
 - constructors, 258–259, 276
 - sealed, 295
 - virtual methods, 262–263
 - base keyword, 258–259, 263
 - BasedOn property, 642
 - BeginWrite method, 574
 - bidirectional data binding, 659–663
 - bin folder, 14
 - binary operators, 494–497
 - infix notation, 495

binary trees

- binary operators, *continued*
 - naming convention, 495
 - binary trees
 - building, 377–378
 - building with generic methods, 389–391
 - building with generics, 379–388
 - creating and populating, 430–431
 - displaying contents, 378–379
 - inserting items, 377, 380–381
 - theory, 376–379
 - traversing, 384–385
 - traversing with enumerator, 425–429
 - traversing with enumerator implemented by iterator, 434–436
 - binary values, 356
 - BinarySearch method, 396
 - binding expressions, 654–655. *See also* data binding
 - BitArray class, 400
 - bitwise operators, 354–355
 - Blank App template, 18–19
 - Blend for Visual Studio 2015, 648
 - blocking wait operations, 600
 - blocks of memory, 177–178
 - blocks of statements, 93
 - for statements, 114
 - while statements, 109
 - bool variables, 88
 - arrays of, 356–357
 - Boolean expressions. *See also* if statements
 - parentheses, 92
 - in while statements, 109
 - Boolean operators, 88–91, 105
 - Boolean values
 - negating, 186
 - returning, 237
 - true and false, 356
 - Boolean variables, 87–88
 - bottlenecks, CPU, 530–533
 - boxing, 194, 200, 247
 - overhead, 372
 - braces or curly brackets ({ }), 53, 60
 - for array element initialization, 223
 - for class definitions, 154
 - grouping statements, 93
 - for scope definition, 66–67
 - break statements, 100–102, 237
 - in iteration statements, 116
 - Build Solution command, 12–13, 32
 - business logic, 69
 - decoupling from applications, 439–467
 - exposing to view, 666. *See also* ViewModel
 - separation from UI and data, 651–652, 696
 - in ViewModel, 652
 - busy indicators, 726–728
 - Button class, 458–459
 - Button controls, 24, 524
 - adding to UWP apps, 677–680
 - coding, 30–32
 - Command property, 678
 - byte arrays, 525
- ## C
- C#
 - case sensitivity, 8, 34
 - compiler. *See* compiler
 - creating projects in Visual Studio 2015, 3. *See also* Visual Studio 2015
 - displaying code files, 26–27
 - keyword combinations, 302
 - layout style, 34
 - positional notation, 280
 - primitive data types, 37–44
 - source files, 8
 - statements, 33–34. *See also* statements
 - syntax and semantics, 33–58
 - type-checking rules, 259
 - variables, 36–37, 54–57. *See also* variables
 - white space characters, 34
 - cached collections, 487–490
 - calculateClick method, 65, 137, 144–145
 - callback methods, 574
 - camelCase notation, 36
 - Cancel method, 543
 - cancelling tasks, 543–554, 558
 - cancellation
 - of blocking wait operations, 600
 - of PLINQ queries, 580
 - of synchronization, 587
 - cancellation tokens, 543–549, 558
 - creating, 543–544
 - registering callback methods, 544
 - task status, 551–552
 - CancellationToken objects, 543–544, 551
 - for PLINQ queries, 580
 - CancellationTokenSource objects, 544, 558, 580
 - CanExecute method, 669, 671–672, 679

- CanExecuteChanged events, 670, 672–673, 679
- canExecuteChangedEventTimer_Tick method, 672–673
- Canvas class, 285
- canvas controls, 284
- Carroll, Lewis, 433
- case keyword, 100
- case labels, 100–101
- casting, 195–198, 200
 - explicit, 372
- catch handlers, 128–129, 150. *See also* try/catch blocks
 - matching exceptions to, 132–133
 - missing, 129
 - multiple, 130–131
 - placement in try blocks, 132
- catchall handlers, 147, 150
- catching exceptions, 128–129, 146–148, 152
- characters, unique codes, 119
- checked expressions, 140–143
- checked keyword, 139–143
- checked statements, 139–140
- CIL (Common Intermediate Language), 218
- Circle class, 154–155
- Circle objects, 168
- Circle variables, 177
- class hierarchies, 266–271
- class keyword, 154
- class libraries, 379, 382
 - .NET Framework, 17, 399–400, 441–443, 587–588
- class methods, 168
- class scope, 66–67
- class type variables, 155
- class types, 177
- classes, 8. *See also* objects
 - abstract, 258, 277, 293–300
 - accessibility, 156–167
 - anonymous, 172–173
 - bringing into scope, 170
 - collection, 277–278, 399–410, 587–598. *See also* collections
 - comparing operators, 500
 - compatibility with WinRT, 218–219
 - constructed types, 375
 - constructors, 157–164, 258
 - copying, 217–218
 - creating, 286–290
 - declaring, 160–161, 174
 - defining, 154–155
 - derived and base, 256
 - event sources, 456
 - field initialization, 209–210
 - fields, 153, 155. *See also* fields
 - generic, 373–388, 397
 - inheritance, 255–276
 - interfaces, 277–278. *See also* interfaces
 - keyword combinations, 302
 - methods, 153. *See also* methods
 - namespaces, 14–15, 301. *See also* namespaces
 - partial, 159–160, 711
 - private data, 180
 - referencing through interfaces, 280–281
 - rules for, 301–302
 - sealed, 256, 295–302
 - static, 169
 - static methods and data, 167–173
 - vs. structures, 209–210
 - synchronization primitives, 584–586
 - testing, 290–293
- classification, 153–154
 - inheritance, 255–256
- Click events, 458–459
- Clone method, 179–180, 229–230
 - copying arrays, 359
- Close button, 26
- Close method, 311
- cloud, deploying web services to, 719–721
- “Cloud Service Fundamentals Data Access Layer—Transient Fault Handling” page, 715–716
- CLR. *See* common language runtime (CLR)
- code
 - accessing, 40–41
 - blocking, 571
 - commenting out, 488
 - deadlocks, 571–572
 - design and properties, 337
 - displaying files, 26–27
 - duplication, 293–294, 298–300. *See also* abstract classes
 - ensuring it will run, 148–149
 - execution, 129, 439–440, 460, 517
 - failures, 127
 - flow of control, 148
 - managed, 218–219
 - native, 218
 - parallelizing, 537–540
 - refactoring, 73

Code and Text Editor window

- Code and Text Editor window. *See also*
 - Visual Studio 2015
 - generating and editing code shortcut menu, 69–70
 - Quick Actions option, 70
 - “Code First to an Existing Database” page, 704
 - code-first entity modeling approach, 704
 - codes for characters, 119
 - Collect method, 309
 - collection classes, 277–278, 399–410
 - thread-safe, 587–598
 - collections
 - adding elements, 401, 421
 - AggregateException exceptions, 555
 - vs. arrays, 415
 - BitArray class, 400
 - collection classes, 399–410. *See also* collection classes
 - counting elements, 403, 421
 - creating, 420
 - data integrity, 478
 - defining type, 474
 - Dictionary<TKey, TValue> class, 407–408
 - enumerating elements, 423–431
 - finding elements, 421
 - HashSet<T> class, 409–410
 - implementing, 416–420
 - initializers, 411–413
 - inserting elements into middle, 401
 - inserting items, 403
 - iterating, 401, 403, 421
 - iterating in parallel, 576–578
 - joining, 578–580
 - lambda expressions, 413–415
 - LinkedList<T> class, 403–404
 - List<T> class, 401–403
 - nongeneric, 400
 - predicates, 411–413
 - Queue<T> class, 404–405
 - removing elements, 401, 403, 421
 - sizing, 416
 - SortedList<TKey, TValue> class, 408–409
 - sorting elements, 401
 - Stack<T> class, 405–406
 - storing and retrieving data, 587–588
 - thread-safe, 400
 - colon (:) in named parameters, 79
 - Colors class, 291
 - ColumnHeader style, 643
 - COM (Component Object Model), 77
 - ComboBox controls
 - adding to page, 613–614
 - adding values, 614–615
 - data binding with, 663–665
 - ComboBoxItem controls, 614–615
 - command buttons, 233
 - command Mode property, 692
 - Command pattern, 669–680
 - Command Prompt windows, 3
 - Command property, 678
 - command sets, Cortana, 682–683
 - CommandBar controls, 678
 - CommandPrefix element, 682
 - commands. *See also names of individual commands*
 - activating with Cortana, 680–695
 - adding to ViewModel, 669–680
 - binding control action to, 669–670
 - spoken or typed, 692
 - comments, 11
 - commenting out code, 488
 - TODO comments, 161
 - Common Intermediate Language (CIL), 218
 - common language runtime (CLR), 77, 218, 300–301
 - catching all exceptions, 142–143, 146
 - object destruction, 306. *See also* garbage collection
 - Compare method, 99, 395–396
 - compareClick method, 95–96
 - CompareTo method, 380–381, 384
 - default implementation, 482
 - compiler
 - destructor conversion, 308
 - memory allocation for class types, 177
 - memory allocation for value types, 177
 - property generation, 344–345
 - resolving method calls, 80–85
 - type checking, 374
 - compile-time errors, 68
 - compiling code, 12
 - complex numbers, 501–502
 - Complex type, 502
 - Component Object Model (COM), 77
 - compound addition (+) operator, 108
 - compound assignment evaluation, 498–499
 - compound assignment operators, 107–108, 119
 - compound subtraction (-) operator, 108
 - computer memory, 184, 190–193, 198, 206, 305
 - concurrency, 519. *See also* multitasking
 - synchronizing access to data, 581–598

- concurrent operations. *See also* parallelization
 - data corruption, 583
 - parallelized, 594–596
 - scheduling, 581
 - tasks for, 517–558, 560–561
- ConcurrentBag<T> class, 587, 596–597
- ConcurrentDictionary<TKey, TValue> class, 587–588
- ConcurrentQueue<T> class, 588
- ConcurrentStack<T> class, 588
- conditional logical operators, 89–91
- ConfigureAwait(false) method, 571
- connection strings, passwords in, 711
- Console Application template, 5
- console applications, 3–17
 - creating in Visual Studio 2015, 3–8, 32
 - defined, 3
- Console class, 8–9
- Console.Write method, 71
- Console.WriteLine method, 166, 214
 - overloading, 243–244, 248
- const keyword, 169
- constantExpression identifier, 100
- constraints
 - generics and, 375
 - placement, 429
- constructed types, 375
- constructors, 26–27, 156–164
 - base class, 258–259
 - calling, 162, 174
 - creating, 317
 - declaring, 161–162, 174–175
 - default, 158–159, 162, 183, 209, 235
 - in derived classes, 258
 - field initialization, 211–212, 258–259
 - initializing objects, 345–346
 - interfaces and, 283
 - invoking, 162
 - overloading, 158–159
 - protected, 301
 - public and private, 158
 - public default, 259
 - ViewModel, 687–688
 - writing, 160–164
- containers
 - Grid controls, 608–609
 - StackPanel controls, 609
- ContainsKey method, 407
- continuations, 522–523, 556, 561–564
- continue statements, 116–117
- ContinueWith method, 522–523, 556, 558
- contravariance, 395–398
- controllingExpression identifier, 100
- controls
 - adding to forms, 20–21
 - aligning on form, 23
 - anchoring to edges, 610–611
 - binding action to commands, 669–670, 696
 - binding expressions, 654–655. *See also* data binding
 - connectors, 610
 - default properties, 22–23
 - Document Outline window, 49
 - event handlers, 30
 - font styles, 641–642
 - header styles, 642–643
 - label styles, 644–646
 - Name property, 613
 - naming, 23
 - properties, 21–23
 - repositioning, 21
 - resizing, 23
 - scaling, 623–630, 649
 - text wrapping, 613
 - width, 613
 - XAML descriptions, 21
- conversion operators, 508–514
 - implicit and explicit, 509–510
 - user-defined, 509–510
 - writing, 511–513
- Convert.ToChar method, 119
- cooperative cancellation, 543–549
- Copy method, 229–230
- copying
 - arrays, 228–230, 359
 - classes, 217–218
 - deep and shallow, 179
 - reference types, 179–180
 - structure variables, 215–218
 - value type constant to nullable type, 186
 - value type variable to nullable type, 186
 - value types, 177–183
- CopyTo method, 229–230
- Cortana, 680–696
 - enabling, 680–681
 - registering voice commands, 681, 685–686
- Count method, 442, 480
- Count property, 403, 419, 676
- CountdownEvent class, 585–586, 599

covariance

- covariance, 394, 396, 398
- CPU utilization
 - identifying bottlenecks, 530–533
 - multitasking applications, 535–536
 - parallelized applications, 540
 - single-threaded applications, 529–530
- “Create a Windows app” page, 697
- .csproj suffix, 39
- Current property, 424, 426, 428, 667–669, 674
- CurrentCount property, 586

D

- dangling references, 309
- data
 - corruption, 583
 - deleting, 729
 - displaying using data binding, 652–659
 - enumerating in specific order, 491
 - filtering, 474–475, 491
 - inserting, updating, and deleting, 728–746
 - joining, 477–478
 - modeling. *See* classes; structures
 - ordering, grouping, aggregating, 475–477, 491–492
 - privacy, 179–180
 - querying. *See* query operators
 - retrieving, 712
 - retrieving from cloud, 720–726
 - retrieving from databases, 698–728
 - searching with Cortana, 680–695
 - selecting, 472–474
 - shared, 577
 - storing. *See* arrays; collections
 - validating, 739–740
 - views of, 652
- data access
 - concurrent, 581–598
 - locking data, 584
 - to resource pools, 585
 - response time, 575
 - thread-safe, 588–598
- data binding, 635, 652–665, 696
 - bidirectional, 652, 659–663
 - with ComboBox controls, 663–665
 - displaying data, 652–659
 - Mode parameter, 659–663
 - modifying data, 659–663
 - syntax, 654
- “Data Consistency Primer” page, 729
- data model
 - connection to view, 666. *See also* ViewModel
 - controlled access, 668
- data sources
 - for data binding, 652–665
 - relational databases, 698
- data types
 - numeric, 47
 - operators and, 45–47
 - switch statements and, 101
 - ToString methods, 43
- database-first entity modeling approach, 704
- databases. *See also* Azure SQL Databases
 - accessing, 697
 - data validation, 739–740
 - entity models, 698. *See also* entity models
 - error reporting, 738–741
 - GUIDs, 737
 - inserting, updating, and deleting data, 728–746
 - retrieving data from, 698–728
- DataContext property, 656
- dateCompare method, 96–99
- dates, comparing, 94–99
- DateTime data type, 96, 99
- DbContext class, 710
- DbSet generic type, 710–711
- deadlocks, 571–572
- Debug folder, 14
- Debug mode, 13–14, 32
 - exceptions, viewing, 134–135
 - frame rate, 25
 - stepping through iteration statements, 120–124
 - for Universal Windows Platform apps, 25
- debug targets, Device and Local Machine options, 24
- Debug toolbar, 74–77, 86, 120–124
- decimal numbers, converting to octal, 117–119
- decision statements, 42, 87–105
 - Boolean operators, 88–91
 - Boolean variables, 87–88
 - if statements, 91–99
 - switch statements, 99–104
- declaring methods, 60–61
- decrement (-) operator, 55, 499
- deep copying, 179
 - of arrays, 230
- default constructors, 158–159
 - initializing values, 183
 - invoking, 162

- for structures, 209
- writing, 235
- default keyword, 100, 426, 429
- deferred evaluation, 487–490
- definite assignment rule, 38
- delegate keyword, 415
- delegates, 439–454
 - accessing, 445–446
 - async modifier, 566
 - in Barrier constructor, 586
 - declaring, 447–451, 466
 - events, 456–465
 - examples, 441–443
 - implementing, 444–446, 451–454
 - initializing, 444–445, 466
 - invoking, 445, 456
 - lambda expressions and, 455–456
 - method adapters, 455
 - multiple methods, 445
 - public, 445
 - removing methods, 445
 - signatures, 455
- DELETE requests, 728
- delimiter characters, 104
- dependencies, eliminating, 651–652, 698
- Dequeue method, 370–372, 399–400, 404
- derived classes, 256, 276. *See also* inheritance
 - abstract method overrides, 295, 298
 - override methods, 263–264
- Deserialize Object method, 723
- design
 - application, 337
 - code, 337
 - for multitasking, 519–520
- design patterns
 - Command pattern, 669–680
 - Model-View-ViewModel pattern, 651–680
- Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al.), 455
- Design View window, 21, 24. *See also* Visual Studio 2015
- destroying objects, 306
- destructors, 306
 - calling Dispose method from, 314–316
 - creating, 317–318
 - finalization, 310
 - interfaces and, 284
 - overhead, 310–311
 - overlap, 311
 - restrictions on, 307
 - suppressing, 321
 - timing of running, 309–310
 - writing, 306–308, 325
- developer mode for Windows 10, 18–19
- devices. *See also* Universal Windows Platform (UWP)
 - apps
 - families, 602, 631
 - layout for narrow views, 632–635
- Dictionary class, 375
- Dictionary<TKey, TValue> class, 400, 407–408, 416
 - thread-safe version, 587–588
- Dispatcher objects, 563
- DispatcherTimer class, 672
- DispatcherTimer objects, 673
- displayData method, 111–112
- disposal methods, 311–324
 - exception-safe, 312, 316–324
 - in using statements, 312–314
- Dispose method, 314, 319, 427–428
 - calling from destructor, 314–316
 - calling once, 320–323
 - thread safety and, 322–323
- disposed field, 320–323
- DistanceTo method, 165–166
- Distinct method, 477, 480, 484
- DivideByZeroException exceptions, 147, 555–556
- dlg.ShowAsync method, 572
- do statements, 116–125
 - stepping through, 120–124
 - writing, 117–119
- doCancellationWork method, 556
- documentation, comments, 11
- Documents folder, 5
- doIncrement method, 187–188
- dollar sign (\$), 46
- dot notation, 158, 272
- dot operator (.), 306
- double quotation mark ("), 104
- double.Parse method, 71
- doubly linked lists, 403–404
- doWork method, 161, 166, 181
- duplicate code, 293–294. *See also* abstract classes
 - removing, 298–300
- duplicate values, ignoring, 477, 480

E

Ellipse class, 289

else keyword

- else keyword, 91
- encapsulation, 154, 439
 - Add/Remove approach, 446
 - with methods, 329–331
- EndWrite method, 574
- Enqueue method, 370–372, 399, 404
- EnterReadLock method, 586, 600
- EnterWriteLock method, 586, 600
- Entity Data Model Wizard, 706–709
- Entity Framework, 698
 - creating entity models, 703–711, 747
 - ignoring columns, 701
 - partial classes, 711
- “Entity Framework” page, 699
- entity models
 - code-first approach, 704
 - creating, 703–711, 747
 - data connections, 706–707, 710
 - database-first approach, 704
 - editing, 709
 - passwords in connection strings, 711
 - remote database access, 747
 - web service access, 705
 - for web services, 698
- enum keyword, 202
- enum types, 201. *See also* enumerations
- Enumerable class, 473
 - extension methods, 472–479
- enumerable collections, 423–431. *See also* collections
 - cached version, 487–490
 - combining, 480–481
 - Count method, 480
 - defining type, 474
 - enumerating data in specific order, 491
 - enumeration of, 487
 - filtering data, 474–475, 479, 485, 491
 - grouping data, 476, 479–480, 485
 - joining data, 477–478
 - ordering data, 479
 - projecting fields, 474, 491
 - retrieving data, 475–476, 479
 - selecting specific data, 472–474
 - sorting data, 484
 - summary functions, 480
- enumeration variables, 202
 - assigning values, 202, 220
 - converting to strings, 202–203
 - declaring, 219
 - displaying, 202
 - nullable, 202
- enumerations, 201–206
 - creating, 204–206
 - declaring, 202, 219
 - LINQ and, 471
 - literal names, 202
 - literal values, 203, 205
 - underlying types, 204
- enumerator objects, 424
- enumerators
 - for collections, 423–431
 - implementing with iterators, 431–437
 - manually implementing, 425–429, 437
- equality operators, 91
 - = operator, 37, 53–54, 91
 - == operator, 88–89, 505
 - implementing, 505–508
 - overriding, 501
- Equals method, 505
 - overriding, 501, 506
 - for structures, 208
- equals operator, 480
- equi-joins, 480
- Error List window, 12–13
- error reporting, 738–741
- errors. *See also* exception handling; exceptions
 - managing, 127–128
- event handlers, 290
 - adding to controls, 30
 - single-threaded nature, 547
- event keyword, 456
- event sources, 456
- events, 440
 - declaring, 456–457, 466
 - defined, 30
 - enabling notifications, 456–458
 - raising, 458, 461–465, 467
 - security feature, 458
 - subscribers, 456
 - subscribing, 457, 465, 467
 - unsubscribing, 457, 467
 - user interface, 458–465
 - waiting for, 585
- exception classes, 143, 302
- Exception exception family, 131–132
- exception filters, 132–133
- exception handling
 - debugging, 142–143
 - task cancellation exceptions, 552–554
 - task exceptions, 554–556, 558

- exception objects, 144
 - exceptions
 - AggregateException, 554–556
 - catching, 128–129, 146–148, 150
 - catching multiple, 131–132
 - finally blocks, 148–149
 - flow of control and, 148
 - inheritance hierarchies, 131–132
 - matching to catch handlers, 132–133
 - Message property, 129
 - multiple catch handlers, 130–131
 - propagating, 129–130
 - propagating to calling method, 136–138
 - task, 554–556
 - throwing, 143–148, 150
 - tracing back to initial exception, 135
 - try/catch statement blocks, 133–136
 - unhandled, 129–130, 133
 - exception-safe disposal, 312, 316–324, 326
 - ExceptWith method, 409–410
 - Execute method, 670–671
 - execution
 - asynchronous methods, 460
 - flow of control, 148
 - interrupting flow to perform tasks, 439–440
 - parallel paths, 517. *See also* Task objects
 - single-threaded, 517
 - in try/catch blocks, 129
 - Exists method, 418, 441
 - ExitReadLock method, 586, 600
 - ExitWriteLock method, 586, 600
 - explicit keyword, 509, 514
 - expression-bodied methods, 62–63, 72, 413
 - expressions
 - associativity of operators, 53
 - converting to another type, 508–513
 - precedence of operators, 52–53
 - Extensible Application Markup Language (XAML), 18
 - extension methods, 271–276
 - Extract Method command, 73
- ## F
- F10 key, 75
 - F11 key, 75
 - fast lookup of items, 409–411
 - Feedback element, 684
 - fields, 67, 153, 155
 - accessibility, 156–167. *See also* properties
 - hiding, 329–331, 343. *See also* properties
 - initializing, 156, 209–211, 258–259
 - interfaces and, 283
 - naming, 157, 333
 - private, 156, 265–266, 329
 - projecting, 474, 491
 - protected, 265–266
 - public, 156, 265–266, 343–344
 - static, 167–169
 - variables as, 164
 - file handles, 148
 - FileOpenPicker class, 111, 572–573
 - files
 - reading characters from, 111
 - resources, releasing, 112
 - FillList method, 432
 - filtering data, 474–475
 - finalization, 310
 - suppressing, 321
 - Finalize method, 308
 - finally blocks, 148–150
 - disposal method calls, 312
 - Find method, 411–412, 417, 441
 - First property, 403
 - float values, 42–43
 - flow of control. *See also* execution
 - after exceptions, 148
 - interrupting to perform tasks, 439–440
 - FontSize property, 22
 - FontStyle style, 641–642
 - for statements, 114–116, 437
 - initialization, 114–115
 - iterating arrays, 225–226
 - scope, 115–116
 - foreach statements, 250
 - iterating arrays, 226
 - iterating binary trees, 430–431
 - iterating collections, 401, 407
 - listing items in arrays, 423
 - foreground activation, 680
 - FormatException exceptions, 128–129, 131, 135–136
 - forms
 - add and edit functionality, 732, 741–743
 - busy indicators, 726–728
 - controls, adding, 20–21
 - defined, 20
 - discarding data, 733–734
 - displaying data, 725–726

forward slash

- forms, *continued*
 - Grid elements, 22
 - properties, specifying, 21–23
 - saving data, 734
 - testing, 743–746
 - XAML description, 21
- forward slash (/), 45, 504
- forward slashes (//), 11
- Frame objects, 29
- frame rate in Debug mode, 25
- freachable queue, 310
- from query operator, 479, 486
- FromAsync method, 575
- Func<T, TResult> delegate, 442
- Func delegates, 441–442, 473, 670–671

G

- Gamma, Erich, 455
- garbage collection, 184, 306, 308–311
 - execution of, 310
 - finalization, 310, 321
 - invoking, 309, 325
 - timing, 309, 318
- GC class, 321
- GC.Collect method, 309
- GC.SuppressFinalize method, 316
- Generate Method command, 71
- Generate Method Stub Wizard, 69–73, 86
- generic IComparable<T> interface, 381
- generic Swap<T> method, 389
- generics, 373–375
 - classes, creating, 376–388, 397
 - collection classes, 399–400
 - constraints, 375, 397
 - contravariance, 395–398
 - covariance, 394, 396, 398
 - vs. generalized classes, 375
 - IEnumerable<T> interface, 424
 - IEnumerator<T> interface, 424
 - interfaces, 391–397
 - methods, creating, 389–391, 397
 - type parameters, 373–375, 389, 397
- gestures, 602–603
- get accessors, 332, 334, 336, 343
 - accessibility, 335–336
 - implementing properties, 338
 - for indexers, 363–364
 - in indexers, 356–358
- get blocks, 332
- get keyword, 332, 360
- GET requests, 736
- GetAsync method, 736, 748
- GetAwaiter method, 564
- GetData method, 392, 727
- GetDataAsync method, 723–725, 727, 730, 738–739
- GetEnumerator method, 424, 429–430
 - implementing with iterator, 432–433
- GetHashCode method, 395
 - overriding, 501, 506–507
- GetPosition method, 291
- GetType method, 132
- GetTypeName method, 265
- global resource dictionary, 639–640, 649
- globally unique identifiers (GUIDs), 737
- goto statements, 102
- graphical applications, 17–32. *See also* Universal Windows Platform (UWP) apps
 - adding code, 29–32
 - creating in Visual Studio 2015, 18–26
 - MainPage.xaml file, 19–20
 - Model-View-ViewModel design pattern, 651–680
 - separation of UI design from data and business logic, 651–652
 - views of, 18
- graphical user interface (GUI), 602–603. *See also* user interface (UI)
- greater than expression, 377
- Grid controls, 49, 524, 608–609
 - defining rows and columns, 623–626
 - positioning TextBlock controls, 623–624
 - referencing from XAML markup, 632
 - for scalable UI, 649
 - tabular layouts, 621–630
- Grid elements, 22
- GridStyle style, 640
- GroupBy method, 476, 485, 492
- groupby query operator, 479, 492
- grouping data, 475–477
- GUI (graphical user interface), 602–603. *See also* user interface (UI)
- “Guide to Universal Windows Platform (UWP) apps” page, 602
- “Guidelines for app suspend and resume” page, 603
- GUIDs (globally unique identifiers), 737

H

handheld devices. *See* Universal Windows Platform (UWP) apps

Handle method, 555, 558

handleException method, 555–556

HashSet<T> class, 400, 409–410

Haskell, 412

HasValue property, 186–187

HeaderStyle style, 642–643

heap, 191–193

- allocating memory from, 306
- array elements, 222
- boxing and unboxing requirements, 196
- deallocating memory, 306. *See also* garbage collection
- freachable queue, 310
- object references, 194

Helm, Richard, 455

hidden code, 27

hill-climbing algorithm, 521

HorizontalAlignment property, 21

HTTP PUT, POST, and DELETE requests, 728

HttpClient class, 720, 748

HttpResponseMessage class, 720

HttpResponseMessage objects, 736, 748

Hungarian notation, 36

I

IActivatedEventArgs type, 686

IAsyncResult design pattern, 574–575

ICommand interface, 669–670

- implementing, 670–673

ICommandBarElement interface, 679

IComparable interface, 279, 380–381, 482

IComparable<T> interface, 381

IComparer interface, 395–396

IComparer<T> objects, 396

icons

- IntelliSense, 11
- in UWP apps, 678, 689–690

idempotency, 728

identifiers, 34–35

- keywords, 34–35
- naming, 157
- overloading, 68
- scope, 66
- syntax, 34

IDisposable interface, 314, 326, 427

- implementing, 314–316, 319–320

IEnumerable interface, 424–431

- implementing, 429–431, 437
- for LINQ, 471

IEnumerable<T> interface, 424

- covariance, 394
- for LINQ, 471

IEnumerable<TItem> interface, 434–436

IEnumerable objects, 582

IEnumerable.GetEnumerator method, 429–430, 434–436

IEnumerable<TItem>.GetEnumerator method, 429–430, 434–436

IEnumerator interface, 424

IEnumerator<T> interface, 424–429

if statements, 91–99, 105. *See also* Boolean expressions

- cascading, 94–99
- grouping statements, 93
- rewriting as switch statements, 99
- writing, 94–99

if-else statements, 383–384

Image controls, 524, 526

ImageBrush resource, 639

immutable properties, 345

Implement Interface Explicitly command, 287, 427, 429, 482

Implement Interface Wizard, 280

implicit keyword, 509, 514

implicitly typed arrays, 224–225

implicitly typed variables, 56–57

in qualifier, 396

increment (++) operator, 54–55, 499

indexers, 353–368

- as [], 494
- accessing array elements, 359–360
- accessor methods, 357–358, 363–364, 368
- vs. arrays, 358–360
- calling, 365–366
- creating, 368
- defined, 353
- explicit implementations, 361, 368
- in interfaces, 360–361, 368
- notation, 411
- overloading, 363–364
- range checks on index values, 357
- read/write context, 358

indexes for arrays

- indexers, *continued*
 - virtual implementations, 360–361
 - in Windows applications, 361–367
 - writing, 363–364
- indexes for arrays, 225
- IndexOf method, 363
- IndexOutOfRangeException exceptions, 555–556
- indirect unmanaged resources, 307
- inequality (!=) operator, 88–89
- infinite values, 47
- infix notation, 495
- information hiding, 154
- inheritance, 193, 255–276
 - from abstract classes, 277
 - abstract classes and, 293–295
 - assigning classes, 259–261
 - base-class constructors, 258–259
 - class hierarchies, 266–271
 - declaring, 256–257
 - declaring new methods, 261–262
 - declaring override methods, 263–265
 - declaring virtual methods, 262–263
 - extension methods, 271–275
 - hierarchies of exceptions, 131–132
 - from interfaces, 279
 - interfaces and, 283
 - from managed types, 301
 - protected access, 265–266
 - public nature of, 257
 - from System.Object class, 258
- initialization
 - collection class, 411–413
 - in constructors, 258–259
 - fields, 156
- initializeComponent method, 26–27
- INotifyPropertyChanged interface, 660–661, 673–674, 696
- Insert method, 383
- InsertIntoTree method, 389–391
- InstallCommandDefinitionsFromStorageFileAsync method, 685
- instance methods, 164, 181
 - writing, 165–167
- instances, 165
- int keyword, 387
- int parameters, 181–182
- int types, 204
 - as array indexes, 235
 - as array of bits, 353–357
 - binary representation, 353–355
 - parameter arrays of, 244–247, 249–251
- int values
 - adding, 63
 - arithmetic operations, 47–48
 - containers for, 353. *See also* int types
 - converting strings to, 45, 58
 - of enumeration literals, 203, 205
- int variables
 - declaring, 165, 178
 - memory allocation for values, 177
- Int32.Parse method, 45
- integer arithmetic, 119
 - checked and unchecked, 138–143
 - overflow checking, 139, 150
- integer division, 147
- integer indexes, 407
- IntelliSense, 9
 - icons, 11
- interface extension, 280
- interface keyword, 278
- interface properties, 337–339, 350
- interfaces, 277–278, 337–338
 - checking objects for implementation, 281
 - for collectable objects, 277–278
 - contravariant, 395–397
 - covariant, 393–394
 - defined, 277
 - defining, 278–279, 284–286
 - explicitly implementing, 282–283, 291
 - fields vs. properties, 344
 - generic, 391–397
 - I prefix, 279
 - implementing, 279–280, 286–290, 303, 338–339
 - indexers in, 360–361, 368
 - invariant, 393
 - keyword combinations, 302
 - methods in, 278
 - multiple, 281
 - references to, 291
 - referencing classes through, 280–281
 - restrictions, 283–284
- IntersectWith method, 409–410
- int.MinValue and int.MaxValue properties, 138
- int.Parse method, 50, 72, 128, 207
- Invalidate method, 526
- InvalidCastException exceptions, 195–196, 393
- InvalidOperationException exceptions, 145–146, 428
 - catch handler, 146
- invariant interfaces, 393

I/O operations
 asynchronous methods for, 573–574
 responsiveness, 517

IRetrieveWrapper<T> interface, 393–394

is operator, 196–197, 281
 in class assignments, 260–261

IsAtEnd property, 675

IsAtStart property, 675, 679

IsBusy property, 727

IsCancellationRequested property, 543–544

IsChecked property, 145

IsNullOrEmpty method, 365

IsProperSubsetOf method, 409–410

IsProperSupersetOf method, 409–410

IsSubsetOf method, 409–410

IsSupersetOf method, 409–410

IStoreWrapper<T> interface, 393–394

iteration statements, 107
 break statements in, 116
 continue statements in, 116–117
 control variable updates, 109, 114
 do statements, 116–124
 for statements, 114–116
 while statements, 108–114

iterators, 431–436
 implementing, 432–433
 implementing enumerators, 434–436

IWrapper interface, 392–393

J

jagged arrays, 231–232, 242

JavaScript Object Notation (JSON), 712, 723

Johnson, Ralph, 455

Join method, 477–478, 492

join query operator, 480, 492

joining data, 477–478

JsonConvert class, 723

JSON.NET package, 721

K

Key field, 476

Key property, 408

key/value pairs, 407–409

keyword combinations, 302

keywords, 34–35
 list of, 35

Knuth, Donald E., 376

L

LabelStyle style, 644–646

lambda calculus, 413–414

lambda expressions
 anonymous methods and, 415
 delegates and, 455–456
 elements of, 412
 forms of, 413–415
 syntax, 413

language interoperability, 498

Language-Integrated Query (LINQ), 469–470
 in applications, 470–472
 converting to PLINQ queries, 575
 deferred evaluation, 487–490
 equi-joins, 480
 filtering data, 474–475
 forcing evaluation, 487–488, 492
 iterating collections, 576–578
 joining collections, 578–580
 joining data, 477–478
 ordering, grouping, aggregating data, 475–477
 parallelizing queries, 576–580
 query operators, 479–487
 query response time, 559–560
 selecting data, 472–474

Last property, 403

“Launch a background app with voice commands”
 page, 684

left-shift (<<) operator, 354

Length property, 225–226, 242, 403

libraries of compiled code, 7–8. *See also* assemblies

LinkedList<T> class, 400, 403–404

LINQ. *See* Language-Integrated Query (LINQ)

List<T> class, 396, 400–403, 441

List<T> collections, 418

ListCustomer object, 667

ListBox controls, 40

ListenFor element, 684

lists, enumerating, 424

local scope, 66–67

local variables, 66
 declaring, 165
 displaying values, 121–123
 life span, 191

lock statements, 322–323, 584, 597–599

locking data

- locking data, 584
 - serializing method calls, 597–598
 - synchronization primitives, 584–586
- logic, 69. *See also* business logic
 - decoupling from applications, 439–467
- logical AND (&&) operator, 89–90, 105
- logical OR (||) operator, 89–90
- loops. *See also* for statements; foreach statements; while statements
 - canceling, 549
 - dependencies of iterations, 541–543
 - independent iterations, 534
 - iterating arrays, 533–534
 - parallel iterations, 537, 539–540

M

- Main method, 8–10, 27
 - array parameters, 228
- MainPage.xaml file, 19–20
 - for device families, 631
 - displaying, 687
- MainPage.xaml.cs file, 19, 26, 606–607
- managed applications, 300
- managed code, 218–219
- managed execution environment, 218
- managed resources, 306
- managed types, 301
- Manifest Designer, 604
- ManualResetEventSlim class, 585, 599
- mapping elements, 407–408. *See also* indexers
- Margin property, 21
- matched character pairs, 10
- Math class, 154, 167
- Max method, 442
- memory, computer, 190–193
 - garbage collection, 184
 - managing with structures, 206
 - for method calls, 191
 - for objects, 191
 - reclaiming for value types, 305
 - unsafe code and, 198
- memory allocation
 - for arrays, 222–223
 - for class types, 177
 - for value types, 177
- memory blocks, referencing, 177–178
- MemoryStream class, 574
- Message property, 129
- MessageDialog objects, 31, 572
- method adapters, 455
- method calls, 65–66
 - compiler resolution of, 80–85
 - independent, 543
 - memory acquisition, 191–193
 - methods with optional parameters, 79
 - parallel running, 537–538
 - Peek Definition command, 82
 - serializing, 597–598
 - syntax, 63–64
 - through delegates, 440–441
- method parameters
 - life span, 191
 - value vs. reference types, 179
- method scope, 66–67
- method signatures
 - separating from method implementation, 277.
 - See also* interfaces
 - of virtual and override methods, 264
- methodName, 60, 64
- methods
 - abstract, 295
 - accepting multiple arguments of any type, 247–248, 254
 - accepting multiple arguments of given type, 244–247, 254
 - accessor, 330–331
 - adding to events, 457
 - anonymous, 415
 - arguments, 64, 79, 187–190
 - arrays as parameters or return values, 227–228
 - asynchronous, 460, 560–575
 - body statements, 60
 - calling, 63–66
 - constructors, 157–164. *See also* constructors
 - creating, 59–66
 - declaring, 60–61, 86
 - definitions, 62–63
 - disposal, 311–324
 - encapsulation with, 329–331
 - expression-bodied, 62, 86, 413
 - extension, 271–275
 - Extract Method command, 73
 - Generate Method Stub Wizard, 69–73, 86
 - generic, 389–391, 397
 - hiding, 260–263
 - if statements, 91
 - initializing parameters, 188–189
 - instance, 164–167

- invoking through delegates, 440–441
 - keyword combinations, 302
 - length of, 63
 - methodName, 60
 - modifier, 330–331
 - named arguments, 77–85
 - named parameters, 86
 - naming conventions, 157
 - optional parameters, 77–86, 252–254
 - overloading, 10, 68, 243–244
 - override, 295–296
 - parameter arrays for. *See* parameter arrays
 - parameterList, 60, 252
 - parentheses for parameters, 413
 - passing arguments, 187
 - passing objects as arguments to, 183
 - Peek Definition command, 82
 - private, 265–266, 675–676
 - private qualifier, 71
 - propagating exceptions back to, 137–138
 - protected, 265–266
 - public, 265–266
 - public and private keywords, 156
 - referencing, 440. *See also* delegates
 - replacing with properties, 339–343
 - return types, 60, 73
 - return values, 86, 237
 - returning data from, 61–62, 86
 - returnType, 60
 - scope, 66–68
 - sealed, 295–296
 - sharing information between, 67
 - signatures, 261–262
 - statements in, 33
 - stepping through, 74–77, 86
 - subscribers, 456
 - summary, 476–477
 - System.Threading.CancellationToken parameter, 543
 - unsafe keyword, 198
 - variable number of object arguments, 247–248
 - virtual, 295
 - writing, 68–77
 - Microsoft Azure, 697, 705
 - Microsoft Azure SQL Database, 699
 - Microsoft Blend for Visual Studio 2015, 648
 - Microsoft .NET Framework. *See* .NET Framework
 - Microsoft Patterns & Practices Git repository, 560
 - Min method, 245–246
 - minus sign (-), 45, 503–504
 - (decrement) operator, 55
 - = (compound subtraction) operator, 108, 125
 - = (subtraction and assignment) operator, 457
 - Mobile Services, 697
 - Mode parameter, 659–663
 - modeling data. *See* classes; structures
 - Model-View-ViewModel (MVVM) design pattern, 651–680
 - data binding, 652–665
 - data binding with ComboBox controls, 663–665
 - displaying data, 652–659
 - modifying data, 659–663
 - ViewModel, 665–680
 - modifier methods, 330–331
 - Moore, Gordon E., 518
 - Moore's Law, 518
 - mouse position, 291
 - MoveNext method, 424, 426–428
 - mscorlib.dll, 17
 - multicore processors, 518–519
 - multidimensional arrays, 230–242. *See also* arrays
 - accessing elements, 230
 - jagged, 231–232
 - memory requirements, 230
 - number of dimensions, 230
 - params keyword and, 246
 - rectangular, 231
 - multiplicative operators, 91
 - precedence, 53
 - multitasking, 517–519. *See also* tasks
 - canceling, 543–554
 - continuations, 556
 - implementing, 519–543
 - Parallel class, 536–543
 - responsiveness and, 517
 - scalability and, 518
 - Task class, 524–536
 - tasks, 520–524
 - threads, 520–521
 - multithreading, 560. *See also* asynchronous methods; threads
 - storing and retrieving data in collections, 587–588
 - MyFileUtil application, 228
- ## N
- \n (newline character), 238
 - Name property, 613

named arguments

- named arguments, 77–85
 - ambiguities, resolving, 80–81
 - passing, 79–80
- named parameters, 79
- nameof operator, 663
- namespace keyword, 15
- namespaces, 14–17
 - assemblies and, 17
 - bringing into scope, 15–16
 - vs. longhand names, 16
 - XAML namespace declarations, 606
- naming identifiers, 157
- NaN (not a number) value, 47
- narrowing conversions, 509
- native code, 218
- Navigate method, 684, 689
- .NET Framework
 - exception classes, 143
 - IAsyncResult design pattern, 574–575
 - multitasking capabilities, 519–520
 - primitive type equivalents, 207
 - synchronization primitives, 584–586
- .NET Framework class library, 17
 - collection classes, 399–400
 - delegates, 441–443
 - thread-safe collection classes, 587–588
- new keyword, 155, 191
 - for anonymous classes, 172–173
 - for array instances, 222–223
 - creating objects, 305–306
 - initializing delegates, 444–445
 - memory allocation for objects and, 177
 - in method signatures, 262
- new methods, 261–262
- New Project dialog box, 4
- newline character (n), 238
- Next method, 223, 236, 675–676
 - serializing calls to, 597–598
- Next property, 403
- NodeData property, 435
- nongeneric collections, 400
- not a number (NaN) value, 47
- NOT (!) operator, 88, 186
- NOT (~) operator, 354
- notifications, 456–458
- NotImplementedException exceptions, 71, 236, 287, 427
- null elements in arrays, 237–238
- null values, 183–185, 199
- nullable types, 183–187
 - heap memory, 191
 - retrieving values of, 186
- nullable values, 145
- null-conditional operator, 184–185
- NullReferenceException exceptions, 185, 445, 458
- numbers, converting to string representation, 117–119
- NumCircles field, 168
- numeric types, 47

O

- obj folder, 14
- Object class, 193, 392
- Object Collection Editor, 614–615
- object initializers, 347–349
- object keyword, 193
- object references, 400
- object types, 72, 194, 369–372, 375
 - holding values and references, 391–392
 - parameter arrays of, 247–248
- object variables, 194, 260
- Object.Finalize method, 308
- objects, 155. *See also* classes
 - accessing members, 306
 - casting, 196–198
 - comparing, 278
 - constructors, 157–164. *See also* constructors
 - creating, 160–164, 305–306
 - dangling references, 309
 - destroying, 306. *See also* garbage collection
 - destructors, 306–308
 - displaying on canvas, 288
 - enumerator, 424
 - finalization, 310
 - initializing with properties, 345–349
 - lifetime, 305–311
 - locking, 584
 - memory allocation for, 177
 - memory requirements, 191
 - passing as arguments to methods, 183
 - private data, 165
 - read and write locks, 586
 - referencing through interfaces, 280–281
 - storing references to, 183–184
 - unmanaged, 198
 - unreachable, 310
 - verifying type, 196–197

- okClick method, 30–31
- on clause of equals operator, 480–481
- OnActivated method, 685–687
- OnLaunched method, 29, 684–685
- OnPropertyChanged method, 661, 674
- openFileClick method, 111
- operands, 45
- OperationCanceledException exceptions, 551, 556, 558, 580
 - catching and handling, 552–554
- operator !=, 500–501
- operator ==, 500–501
- operator+, 497
- operator keyword, 494, 509, 514
- operator overloading, 444, 493–514
- operators, 493–494
 - arguments, 495
 - arithmetic, 45–54
 - associativity, 53, 90–91, 493–494
 - binary, 494
 - Boolean, 88–91
 - comparing, 500
 - compound assignment, 107–108, 498–499
 - conditional logical, 89–90
 - constraints, 494
 - conversion, 508–514
 - data types and, 45–47
 - equality, 501
 - implementing, 501–508, 514
 - increment and decrement, 499
 - for int type bits, 354–355
 - language interoperability and, 498
 - null-conditional, 184–186
 - overloaded, 494–496
 - pairs, 500–501
 - parameter types, 495–496
 - precedence, 52–53, 58, 90–91, 493–494
 - prefix and postfix forms, 55, 499
 - public, 495
 - static, 495
 - structures and, 208
 - symmetric, 496–497
 - unary, 55, 493
- optional parameters, 77–85
 - ambiguities, resolving, 80–81
 - defining, 79
 - vs. parameter arrays, 252–254
- OR (|) operator, 354
- OrderBy Descending method, 476

- OrderBy method, 475–476, 491
- orderby query operator, 479, 491
- ordering data, 475–477
- out keyword, 189, 394
- out parameters, 187–190, 199
 - params keyword and, 247
- OutOfMemoryException exceptions, 192
- Output window, 12
- overflow checking, 139
 - checked expressions, 140–143
- OverflowException exceptions, 129, 131
 - catch handler for, 141–142
 - in checked statements, 139–140
- overloading, 10, 68
 - constructors, 158–159
 - operators, 444, 493–514
 - vs. parameter arrays, 243–244
 - params keyword and, 246–247
- override keyword, 263–265
- override methods, 295–296
 - declaring, 263–265, 276
 - rules for, 264
 - writing, 269–270
- overriding vs. hiding, 262–263

P

- Pack class, 235
- Package.appxmanifest file, 604
- Page tag, 607
- paging, 721
- Parallel class, 536–543
 - abstracting tasks, 536–540
 - considerations for, 541–543
 - degree of parallelization, 538
 - ParallelLoopState objects, 549
 - scheduling tasks, 542
- Parallel LINQ (PLINQ), 560, 575–580, 599
 - canceling queries, 580, 599
 - iterating collections, 576–578
- Parallel.For method, 537, 539–543, 558
 - canceling, 549
 - lock statements, 597
 - unpredictable behavior, 582–583
- Parallel.ForEach method, 542–543, 558
 - canceling, 549
- Parallel.ForEach<T> method, 537
- Parallel.Invoke method, 537–538, 541, 543

parallelization

- parallelization, 536–540. *See also* concurrent operations; multitasking; tasks; threads
 - in asynchronous methods, 565–566
 - degree of, 520
 - implementing, 538–540
 - iterating collections, 576–578
 - joining collections, 578–580
 - of query operations, 575–580
 - scheduling, 581
 - units of, 520
- ParallelLoopState objects, 537, 549
- ParallelQuery objects, 575, 582
- parameter arrays, 243–254. *See also* arrays
 - declaring, 245–247
 - elements of any type, 254
 - elements of given type, 254
 - int types, 244–247, 249–251
 - number of, 247
 - object types, 247–248
 - vs. optional parameters, 252–254
 - params object[], 247–248
 - priority of, 247
 - writing, 249–251
- parameterList, 60
- parameters
 - adding to class, 238–239
 - arrays as, 227–228
 - destructors and, 307
 - determining minimum value, 244–246
 - in lambda expressions, 414
 - naming, 72, 86
 - optional, 77–86, 252–254
 - parentheses for, 413
 - specifying, 79, 86
 - type, 373–375
 - types, 60
 - variable numbers of, 243–244. *See also* parameter arrays
 - variables as, 164
- params keyword, 244–246. *See also* parameter arrays
 - overloading and, 246–247
- params object[], 247–248
- parentheses, 53, 60, 413
 - in Boolean expressions, 92
 - in expressions, 89
 - in method calls, 64, 66
 - precedence override, 90
- Parse method, 65, 207
- partial classes, 159–160, 711
- partial keyword, 160
- ParticipantCount property, 586
- ParticipantsRemaining property, 586
- Pass.Reference method, 182–183
- Pass.Value method, 181
- Patterns & Practices Git repository, 560
- Peek Definition command, 82
- performance, parallelization and, 540–541
- Performance Explorer, 530–533
- PhraseTopic element, 684
- PI field, 169
- PickMultipleFilesAsync method, 573
- PickSingleFileAsync method, 573
- PLINQ. *See* Parallel LINQ (PLINQ)
- plus sign (+), 45, 54, 495, 503
 - ++ (increment) operator, 54–55, 499
 - with delegates, 444
- pointers, 197–198
- polymorphism
 - testing, 270–271
 - virtual methods and, 264–265
- Pop method, 399
- positional notation, 280
- POST requests, 728, 735–736, 748
- PostAsync method, 735, 748
- precedence, 90–91, 493–494
 - of compound assignment operators, 108
 - overriding, 52–53, 58, 90
- predicates, 411–413, 418
- prefixing and postfixing operators, 55, 499
- Previous method, 675–676
- Previous property, 403
- primitive types, 37–44. *See also* value types
 - displaying values, 38–40
 - fixed size, 138–139
 - list of, 38
 - .NET Framework type equivalents, 207
- private data, 165
- private fields, 163–164, 265–266, 301, 329
 - privacy of, 180
 - properties, accessing with, 179
- private keyword, 71, 156, 180
- private methods, 265–266, 675–676
- private static fields, 170–172
- processing power
 - increasing, 518–519
 - maximizing, 517. *See also* Task objects
- processors
 - displaying utilization, 529
 - multicore, 518–519

- number of, 521
- Program class, 8, 161, 576
- Program.cs file, 8
- programs. *See also* applications; code
 - attributes, 7
 - testing, 73–74
- ProgressRing control, 726–728
- project files, 7
- project solution files, 7
- projecting fields, 474
- propagating exceptions, 136–138
- properties, 331–333
 - accessibility, 335–336
 - accessing private fields, 179
 - attached, 623–624
 - automatic, 343–345, 351
 - binding control properties to object properties, 652. *See also* data binding
 - declaring, 331–332
 - defined, 331
 - immutable, 345
 - initializing, 345
 - initializing objects with, 345–349, 351
 - interface, 337–339, 350
 - naming, 333
 - of nullable types, 186–187
 - reading values, 333–334
 - read-only, 334
 - replacing methods with, 339–343
 - restrictions on, 336–337
 - simulating assignment, 494
 - virtual, 338
 - write-only, 334–335
- Properties folder, 7
- property getters and setters, 331
- PropertyChanged events, 660–661, 674
- protected access, 264–266
- protected keyword, 266
- pseudorandom number generator, 223
- public const int fields, 235, 238
- public constructors, 213. *See also* constructors
 - default, 259
- public events, 458
- public fields, 265–266, 301, 343–344
- public keyword, 156, 514
- public methods, 180, 182, 265–266, 301
 - writing, 171–172
- Publish Web wizard, 718–721
- Push method, 399

- PUT requests, 728, 735
- PutAsync method, 735, 748

Q

- query operations, parallelizing, 575–580
- query operators, 479–487
 - retrieving data, 486–487
- querying data, 469
- Queue class, 374
 - object-based version, 369–374
- Queue<T> class, 399–400, 404–405
 - thread-safe version, 588
- queues, storing values in, 426–427
- Quick Find functionality, 41
- “Quickstart: Translating UI resources (XAML)” page, 678

R

- Random class, 223, 235, 597
- random number generator, 236
- read locks, 586
- read operations, 586, 600
- ReadAsStringAsync method, 723–724, 736, 748
- reader.Dispose method, 148–149
- reader.ReadLine method, 71, 112, 311
- ReaderWriterLockSlim class, 586, 600
- readInt method, 71
- ReadLine method, 71, 112, 311
- read-only fields, 234
- read-only indexers, 358
- read-only properties, 334, 344–345, 350
- Rectangle class, 286
- Rectangle controls, 653
- rectangular arrays, 231
- ref parameters, 187–190, 199
 - params keyword and, 247
- refactoring code, 73
- reference types
 - arrays, 222. *See also* arrays
 - Clone methods, 179
 - copying, 179–180, 199, 208
 - covariance, 394
 - dangling references, 309
 - declaring, 178
 - defined, 177
 - destructors, 307. *See also* destructors

references

- reference types, *continued*
 - heap memory, 191
 - initializing, 183
 - null values, 184
 - number, 308
 - ref and out modifiers, 190
 - string type, 178
 - System.Object class, 193
- references
 - to arguments, 188
 - to memory blocks, 177–178
- References folder, 7–8
 - optional assemblies, 17
- Regex class, 731
- Register method, 544
- regular expression matching, 731
- “The Regular Expression Object Model” page, 731
- relational databases, 698. *See also* databases
 - connecting to, 698. *See also* entity models
- relational operators, 88–89, 91
- remainder (modulus) operator (%), 46
- Remove method, 401, 409, 446
- RemoveParticipant method, 586
- RenderTransform property, 644
- Representational State Transfer (REST) model, 697, 712
- Reset method, 585
- resource dictionary, 639–640, 649
- resource management, 311–316
 - exception-safe disposal, 312–314, 316–324
 - object lifetimes, 305–311
- resource pools, 585, 599
- resource release, 112, 312–314, 325. *See also* garbage collection
 - preventing multiple releases, 320–323
- resources, styles, 638
- response time
 - asynchronous operations and, 559–600
 - data access operations, 575
- responsiveness, 559
 - asynchronous API and, 572–574
 - improving, 517
 - user interface, 561, 567–568
- REST (Representational State Transfer) model, 697, 712
- REST (Representational State Transfer) web services, 697
 - adding data items, 748
 - consuming in UWP apps, 748
 - creating, 712–721
 - idempotency, 728
 - remote database access, 747
 - retrieving data in UWP apps, 748
 - updating data from UWP apps, 748
- result = clauses, 64
- Result property, 569
- retrieving data. *See* arrays; collections; generics
- return keyword, 61
- return statements, 61–62, 101
- return types, 60, 73
- return values, 237, 414
 - arrays as, 227
 - of asynchronous methods, 569–570
- returnType, 60
- Reverse property, 433
- right-shift operator (>>), 354
- RightTapped events, 292
- RoutedEventHandler delegate, 458–459
- run method, 69
- Run method, 522, 560
- Run To Cursor command, 120
- RunAsync method, 563
- runtime
 - casting checks, 196
 - memory management, 191–192

S

- SaveAsync method, 740–741
- saving, 13
- scalability
 - asynchronicity and, 560
 - improving, 518
- scalable user interfaces, 621–630
- scope
 - blocks and, 93
 - class, 66–67
 - local, 66–67
 - method, 66–67
 - for statements, 115–116
 - of variables, 66
- sealed classes, 256, 295–303
- sealed keyword, 295
- sealed methods, 295–296
- searching
 - with Cortana, 680–695
 - Quick Find, 41
 - voice responses, 692–695

- security, unsafe code, 198
- Segoe Print font, 642, 646
- Select method, 472–474, 491
 - invoking, 484
 - specifying fields, 475
 - summary methods over results, 476–477
- select operator, 479, 486, 490
- selecting data, 472–474
- SelectionChanged events, 40
- selector parameter, 473–474
- semaphores, 584
- SemaphoreSlim class, 585, 599
 - cancellation token, 587
- semicolon (;)
 - in interfaces, 278
 - in return statements, 61
 - in statements, 33
 - in for statements, 115
- serializing method calls, 597–598
- set accessors, 332, 334–336
 - accessibility, 335–336
 - OnPropertyChanged method calls, 661–662
- set accessors, 343
 - implementing properties, 338
 - in indexers, 356–358
- set blocks, 332
- set keyword, 332, 360
- Set method, 585
- SetColor method, 291–292
- SetData method, 392
- Setter elements, 637
 - in styles, 639
- shallow copying, 179
 - of arrays, 230
- shared data. *See also* synchronization
 - synchronizing access, 577, 599
- Shift+F11, 76
- short circuiting, 90
- short types, 204
- Show All Files button, 13–14
- show method, 96
- ShowAsync method, 572
- showBoolValue method, 44
- showDoubleValue method, 44
- showFloatValue method, 42–43
- showIntValue method, 43
- showResult method, 63
- showStepsClick method, 118, 120
- SignalAndWait method, 586
- signatures, method, 261–262
- Simulator, 618–621
 - running, 627–630
- single quotation mark ('), 104
- single-threaded applications, 517, 590–594
 - creating, 524–530
- .sln suffix, 39
- solution files, 7–8
- Sort method, 396, 401
- SortedList<TKey, TValue> class, 400, 408–409
- SortedSet<T> collection type, 410
- sorting data, 377
- source files, 6–7
- source parameter, 473–474
- SpeechSynthesizer class, 695
- spinning, 577
- Split method, 579, 688
- SQL (Structured Query Language), 470
- Sqrt method, 165–167
- square bracket notation, 357
 - for array elements, 401
 - for arrays, 221
 - for key/value pairs, 407
- stack, 191–193
 - structures, 206
- Stack<T> class, 399–400, 405–406
 - thread-safe version, 588
- StackOverflowException exceptions, 333
- StackPanel control, 609
- Start Debugging command, 13–14, 32
- Start method, 522, 560
- Start Without Debugging command, 13–14, 32, 73
- state information, 574. *See also* visual state transitions
- statements, 33–34
 - blocks, 93, 116
 - making run, 148–149
 - semantics, 33
 - syntax, 33
- static classes, 169
- static fields, 167–168, 175, 180
 - creating, 169
- static keyword, 514
- static methods, 8, 167–173
 - bringing into scope, 170
 - calling, 167–168, 174
 - declaring, 167, 174
 - implementing and testing, 249–251
- static properties, 334

static using statements

- static using statements, 170
- static void methods, 390
- Status property, 548
- stepping into and out of methods, 74–77, 86
- Stopwatch objects, 526
- StorageFile class, 573
- storing data. *See* arrays; collections; generics
- StreamReader class, 311
- String class, 392, 579
- string concatenation, 45–46
- string interpolation, 46, 50, 248
- string type, 178
- string values
 - converting to integers, 45, 58
 - representing values in variables, 43–44, 50
 - for variable names, 663
- StreamReader class, 311
- strings
 - appending to strings, 108
 - converting enumeration variables, 202–203
 - converting to integers, 128–129
 - determining if empty, 365
 - splitting, 579, 688
 - wrapping, 392
- struct keyword, 208
- structs, 8
- structure variables
 - copying, 215–218
 - declaring, 210–211, 220
 - initializing, 211–212, 220
 - nullable, 211
- Structured Query Language (SQL), 470
- structures, 8, 206–219
 - vs. classes, 209–210
 - comparing operators, 500
 - compatibility with WinRT, 218–219
 - creating, 212–215
 - declaring, 208–209, 220
 - default constructors, 209, 212–213
 - field initialization, 209, 211–212
 - fields, 208
 - inheritance and, 257–258
 - interface implementation, 279
 - keyword combinations, 302
 - operators for, 208
 - public constructors, 213
 - types, 206–207
- Style elements, 639, 649
- styles
 - BasedOn property, 642
 - defining, 639, 649
 - Microsoft Blend, 648
 - Setter elements in, 639
 - for user interface, 638–648
- subscribers, 456
- subtractValues method, 51
- summary functions, 480
- summary methods, 476–477
- SuppressFinalize method, 321–322
- Swap<T> method, 389
- switch keyword, 100
- switch statements, 99–105, 687
 - fall-through, 101–102
 - rules, 101–102
 - syntax, 100
 - writing, 102–104
- symmetric operators, 496–497, 510–511
- synchronization
 - canceling, 587
 - locking data, 584
 - for shared data access, 577
 - task, 581–598
 - of tasks, 581–599
 - threads, 585, 599
- synchronization primitives, 584–586
- synchronous I/O, 560
- System.Array class, 225, 228–229, 424
- System.Collections namespace, 400
- System.Collections.Concurrent namespace, 400, 587
- System.Collections.Generic namespace, 374–376, 395, 399, 410, 424
- System.Collections.IEnumerable interface, 423–431
- System.Diagnostics.Stopwatch objects, 526
- System.Exception exceptions, 131, 563
- System.IComparable interface, 380–381
- System.IComparable<T> interface, 381
- System.Int32 structure, 65, 206–207
- System.Int32 type, 387
- System.Int64 structure, 206–207
- System.InvalidCastException exceptions, 372
- System.Linq namespace, 473
- System.Object class, 193
 - inheritance from, 258
 - overriding methods, 301
- System.Random class, 223
- System.Single structure, 206–207

System.String class, 178
 System.Threading namespace, 520, 584
 System.Threading.CancellationTokenSource objects,
 543
 System.Threading.Tasks namespace, 519, 537
 System.Threading.Tasks.TaskStatus enumeration, 548
 System.ValueType class, 258

T

tablets. *See* Universal Windows Platform (UWP) apps
 TabularHeaderStyle style, 643
 Tapped events, 290
 TappedRoutedEventArgs parameter, 291
 Task class, 519, 524–536
 cancellation strategy, 543–549
 parallelization, 524–543
 threading, 520
 wait methods, 554–556, 561–562
 Task<TResult> class, 569–570
 Task constructor, 521
 Task objects, 520
 Action delegate, 521
 creating, 521
 implementing, 533–543
 of Parallel class, 536–540
 responsiveness and, 561–562
 running, 522
 TaskCanceledException exceptions, 556
 TaskContinuationOptions value, 523
 TaskContinuationOptions.OnlyOnFaulted value, 556
 TaskCreationOptions object, 522
 TaskFactory class, 575
 tasks, 517–558. *See also* multitasking
 abstracting, 536–540
 asynchronous, 559
 awaitable objects, 565
 blocking, 585
 canceled vs. allowed to run to completion, 551
 canceling, 543–554, 558
 checking cancellation, 552–554
 compute-bound, 559
 continuations, 522–523, 558, 561–564
 cooperative cancellation, 543–549
 creating, 521–522, 557
 exceptions handling, 554–556, 558
 global queue, 520
 halting execution, 586

 handling cancellation exceptions, 552–554
 handling exceptions, 554–556
 overhead, 541
 parallel, 536–540, 558
 PLINQ queries, 575–580
 results, 569–570
 status, 548–552, 554
 synchronization primitives, 584–586
 synchronizing, 523, 581–599
 synchronizing concurrent access to data, 581–598
 waiting for, 523, 547, 557, 561–562, 572
 waiting for events, 585
 Task.WaitAll method, 547
 templates, choosing, 4. *See also* Visual Studio 2015
 ternary operators, 365–366
 testing programs, 73–74
 text boxes, displaying values in, 43
 text files, iterating through, 110–113
 Text property, 21–23
 TextBlock controls, 20–21, 524, 609–610
 adding to page, 464, 611–612, 615–616
 TextBox controls, 23
 adding to page, 612–613, 615–616
 TextReader class, 111, 311
 ThenBy method, 476
 ThenByDescending method, 476
 this keyword, 164, 272, 356–357
 Thread objects, 520
 thread pools, 520
 thread safety, 596–597
 collection classes, 587–598
 collections, 400
 Dispose method and, 322–323
 overhead, 588
 ThreadPool class, 520
 threads, 520–521
 asynchronous methods, 560. *See also*
 asynchronous methods
 deadlocks, 571–572
 defined, 310
 hill-climbing algorithm, 521
 number, 521
 scheduling, 520
 sleep, 542
 synchronizing, 585, 599
 waiting for, 585–586, 599
 Thread.Sleep method, 542
 throughput, improving, 517–558

throw statements

- throw statements, 101, 143–144
 - objects for, 144
- ThrowIfCancellationRequested method, 551, 553
- throwing exceptions, 143–148
- tilde (~), 307
- ToArray method, 415, 487–488, 490, 492
- TODO comments, 161, 180
- ToList method, 487, 492
- ToString method, 43, 203, 482
 - formatting output, 238, 240
 - overriding, 234, 503
 - overriding default behavior, 213–214, 262
 - of structures, 207
- touch user experience, 602
- TResult parameter, 473–474
- triggers for visual state transitions, 635–637
- try/catch blocks, 128
 - catch handler placement, 132
 - writing, 133–136
- try/finally blocks
 - for resource release, 312
- TSource parameter, 473–474
- type checking, 374
- type mismatches, 195
- type nesting, 283
- type parameters, 373–375, 397, 425
 - for collection classes, 399–400
 - constraints, 375
 - for generic methods, 389
 - initializing variables defined with, 429
 - out keyword, 394
 - in qualifier, 396
- type safety
 - contravariance and, 396
 - of type parameters, 394
- type-checking rules, 259
- types
 - anonymous, 224–225
 - casting, 196–198
 - converting, 508–513
 - definitions vs. instances of, 155
 - enum, 201. *See also* enumerations
 - extending, 272–275
 - integer, 204
 - interoperability, 301
 - new, defining, 271–272
 - of structures, 206–207
- typeSelectionChanged method, 40–42

U

- unary operators, 55, 91, 493
- unassigned local variables, 38
- unboxing, 194–196, 200
 - overhead, 372
- unchecked keyword, 139–143
- underscore character (_), 36
- unhandled exceptions, 129–130, 133. *See also*
 - exceptions
 - catching, 147–148
- UnionWith method, 409–410
- Universal Windows Platform (UWP) apps, 18,
 - 601–602. *See also* graphical applications
 - adapting layout, 630–637
 - adding buttons, 677–680
 - adding voice activation icons, 689–690
 - app layout and UI styling, 607
 - app-wide and local resources, 638
 - Blank App template, 605–607
 - busy indicators, 726–728
 - command bars, 677–678
 - command buttons, 233
 - creating in Visual Studio 2015, 18–26, 32, 649
 - data binding, 635
 - Debug mode, 25
 - features, 602–605
 - handling voice activation, 681, 686–689
 - icons, 678, 689–690
 - inserting, updating, and deleting data, 728–746
 - layout for narrow views, 632–635, 649
 - lifetime, 603
 - MainPage.xaml files for device families, 631
 - MainPage.xaml.cs file, 26
 - managing state information, 603
 - mobility, 603
 - Model-View-ViewModel design pattern, 651–680
 - packaging, 603–604
 - pages, 20
 - retrieving from databases, 698–728
 - scalable user interfaces, 607–637
 - scaling to device form factors, 603, 617–618, 649
 - styles for UI, 638–649
 - switching between views, 631, 635
 - tabular layout, 621–630
 - Task class, 520
 - testing, 618–621, 690–691
 - touch interaction, 602
 - Visual State Manager for, 631–637
 - vocal responses, 692–695

- voice activation, 680–695
- WindowsRT compatibility, 301
- unmanaged applications, 300
- unmanaged resources, 306
- unreachable objects, 310
- unsafe code, 198
- unsafe keyword, 198
- user interface (UI)
 - creating, 20–26
 - data binding and, 635, 652–665
 - designing, 18
 - Dispatcher objects, 563
 - displaying data, 652–659
 - events, 458–465
 - PropertyChanged events, 660
 - responding to gestures, 602–603
 - responsiveness, 559, 561, 567–568, 570
 - scalable to device form factors, 607–618
 - single-threaded nature, 547
 - styles, 638–648
 - tabular layouts, 621–630
 - for UWP apps, 607–648
- using directives, 15–16
- using statements, 323–324
 - resource lifetime control, 312–314, 316–324
 - static, 170

V

- value keyword, 358
- Value property, 186–187
- value types. *See also* primitive types
 - copying, 177–183, 199, 208
 - defined, 177
 - enumerations, 201–206. *See also* enumerations
 - initializing, 183
 - memory reclamation, 305
 - nullable, 185–187
 - ref and out modifiers, 190
 - stack memory, 191
 - structures, 206–219. *See also* structures
- values
 - determining minimum, 244–246
 - returning, 414. *See also* lambda expressions
- ValueType class, 258
- var keyword, 56–57, 173, 474
- variables, 36–37
 - adding values, 108, 125
 - arrays, 221–222. *See also* arrays
 - assigning values, 53–54, 58
 - Boolean, 87–88
 - of class types, 155
 - creating, 66
 - declaring, 37, 58, 199
 - declaring and initializing in same statement, 56, 58
 - defined with type parameter, initializing, 429
 - displaying to screen, 71
 - displaying values, 76
 - fields, 67
 - implicitly typed, 56–57, 173
 - incrementing and decrementing, 54–55, 58
 - initializing, 50, 183, 429
 - initializing to same value, 54
 - life span, 191
 - local, 66
 - naming, 36, 333
 - pointers, 197–198
 - qualifying as parameters or fields, 164
 - reference types, 177. *See also* reference types
 - returning names of, 663
 - scope, 66–67, 115–116
 - for sets of items, 221. *See also* arrays
 - storing references to objects, 183–184
 - string representation of values, 43–44, 50, 58
 - structure, 210–211
 - testing for initialization, 184
 - of type object, 194
 - type of, 56
 - unassigned, 38
 - value types. *See* value types
 - values, 43, 50, 58
 - values as arguments, 64
- VerticalAlignment property, 21
- ViewModel, 652. *See also* Model-View-ViewModel (MVVM) design pattern
 - adding commands, 669–680
 - Adding mode, 729–730, 733
 - Browsing mode, 729–730, 733
 - Command pattern, 669–680
 - constructor, 687–688
 - creating, 665–669
 - discarding changes, 733–734
 - Editing mode, 730, 733
 - error-reporting capabilities, 738–741
 - tracking state, 675
 - validating and saving changes, 734
- views
 - Command pattern, 669–680

virtual directories

- views, *continued*
 - connection to model, 666. *See also* ViewModel of data, 652
 - referencing data, 674
- virtual directories, 706
- virtual indexer implementations, 360–361
- virtual keyword, 263, 338
- virtual machines, 218
- virtual methods, 268, 295
 - declaring, 262–263, 276
 - polymorphism and, 264–265
 - rules for, 264
- virtual properties, 338
- Visual C# syntax and semantics, 33–58. *See also* C#
- Visual State Manager, 618, 630–637
 - triggers, 635–637
- visual state transitions, 636
- Visual Studio 2015, 3
 - Allow Unsafe Code option, 198
 - ASP.NET Web API template, 712–713
 - Class Library template, 382
 - Close button, 26
 - Code and Text Editor window, 6
 - Console Application template, 5
 - console applications, 3–17
 - Debug toolbar, 74–77
 - default development environment settings, 4
 - default method implementation, 70
 - design surface form factor options, 607–608
 - Design View window, 21, 24
 - Document Outline window, 49
 - Entity Data Model Wizard, 706–709
 - environment, 3–8
 - Error List window, 12–13
 - Exceptions Settings, 553
 - Generate Method Stub Wizard, 69–73
 - generated code, 26–29
 - graphical applications, 17–32
 - Implement Interface Explicitly command, 287, 427, 429
 - Implement Interface Wizard, 280
 - Interface template, 285
 - LINQ documentation, 481
 - Manifest Designer, 604
 - Microsoft Blend, 648
 - namespaces, 14–17
 - Object Collection Editor, 614–615
 - Output window, 12
 - project files, 7
 - projects, 385–386
 - Properties folder, 7
 - Properties window, 21–23
 - Publish Web wizard, 718
 - Reference Manager dialog box, 386
 - References folder, 7–8
 - Simulator, 618–621, 627–630
 - Solution Explorer, 6–7
 - solution files, 7–8
 - solutions, 385
 - Start page, 3–4
 - starter code, 6
 - Task List window, 161
 - Toolbox, 20–21
 - Universal Windows Platform apps, 18–26, 32, 649
 - Visual State Manager, 618, 630–637
 - warnings, 31
 - web service templates and tools, 698
 - Windows Phone SDK 8.0, 601
 - writing programs, 8–14
 - XAML pane, 21
- Visual Studio 2015 debugger
 - Breakpoints Window button, 121
 - exception handling, 142–143
 - exceptions, 129–130
 - Exceptions Settings pane, 142–143
 - Locals window, 121–123
- Visual Studio Performance Explorer and Profiler, 530–533
- Vlissides, John, 455
- voice activation, 680–696
 - command sets, 682
 - Feedback element, 684
 - handling in apps, 681, 686–689
 - language and locale settings, 683
 - ListenFor element, 684
 - Navigate element, 684
 - PhraseTopic element, 684
 - testing, 690–691
- voice commands
 - registering with Cortana, 681, 685–686
 - vocal responses, 692–695
- voice-command definition (VCD) files, 681–684, 696
- VoiceCommandDefinitionManager manager, 685
- void keyword, 60–61, 73

W

- Wait method, 523, 555, 557, 561–562, 585

- blocking current thread, 571
 - canceling, 600
- WaitAll method, 523, 547, 555, 557
- WaitAny method, 523, 555
- WalkTree method, 384–385, 387
- web apps
 - creating, 705–706
 - running, 706
- web services, 698
 - controller classes, 713–716
 - creating, 712–721
 - deploying to cloud, 719–721
 - edit functionality, 730, 733, 737–738
 - failed connection attempts, 715–716
 - RESTful, 697
 - retrieving data, 712
 - retrieving data from cloud, 720–726
- when keyword, 132
- Where method, 474–475, 485, 491
- where query operator, 479, 491
- while statements, 108–114, 125, 236
 - sentinel variable, 109
 - terminating, 109
 - writing, 110–113
- white space characters, 34
- widening conversions, 508
- Win32 APIs, 218
- windows, defined, 20
- Windows 10, 300–301
 - apps for. *See* Universal Windows Platform (UWP) apps
 - Cortana, 680–695
 - developer mode, 18–19
 - devices, 602
 - speech synthesis features, 692
- Windows Phone Runtime, 601
- Windows Phone SDK 8.0, 601
- Windows Runtime (WinRT), 218–219, 300–301, 601
 - app adaptation to device form factors, 601
 - asynchronous methods and, 572–575
 - hill-climbing algorithm, 521
 - thread management, 520
- Windows Store apps, 601, 603
- Windows Universal template, 18
- Windows.Media.SpeechSynthesis namespace, 695
- Windows.UI namespace, 291
- Windows.UI.Popups namespace, 31
- Windows.UI.Xaml namespace, 672
- Windows.UI.Xaml.Media.Imaging namespace, 525

- WithCancellation method, 580, 599
- WrappedInt class, 181–183
- WrappedInt objects, 183
- Wrapper<T> class, 392–393
- write locks, 586
- Write method, 573
- write operations, 586, 600
- WriteableBitmap objects, 525, 573
 - populating, 526
- WriteAsync method, 573–574
- WriteLine method, 9–10, 68, 71, 167
 - format string argument with numeric placeholders, 248
 - overloading, 243–244, 248
- write-only indexers, 358
- write-only properties, 334–335, 341, 350
- writing to streams, 573–574

X

- XAML (Extensible Application Markup Language), 18
 - namespace declarations, 606
 - XAML files for device families, 631
- XOR (^) operator, 354

Y

- yield keyword, 432

About the author



John Sharp is a principal technologist for CM Group Ltd, a software development and consultancy company in the United Kingdom. He is well versed as a software consultant, developer, author, and trainer, with nearly 30 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 10 and Microsoft Azure. He is an expert on building applications with the Microsoft .NET Framework and is also the author of *Windows Communication Foundation 4 Step By Step* (Microsoft Press).