



ESSENTIAL C# 8.0

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

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

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



IntelliTect

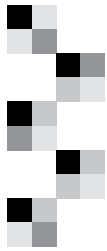
FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Essential C# 8.0

This page intentionally left blank



Essential C# 8.0

Mark Michaelis
with Eric Lippert & Kevin Bost,
Technical Editors

◆ Addison-Wesley

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

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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

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

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

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2020940544

Copyright © 2021 Pearson Education, Inc.

Cover image: Taras Hipp/Shutterstock

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

ISBN-13: 978-0-13-597226-7

ISBN-10: 0-13-597226-4

ScoutAutomatedPrintCode

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

Thanks!

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



This page intentionally left blank



Contents at a Glance

Contents ix
Figures xvii
Tables xix
Foreword xxi
Preface xxiii
Acknowledgments xxxv
About the Author xxxvii

- 1 Introducing C#** 1
- 2 Data Types** 45
- 3 More with Data Types** 81
- 4 Operators and Flow Control** 121
- 5 Methods and Parameters** 195
- 6 Classes** 255
- 7 Inheritance** 335
- 8 Interfaces** 377
- 9 Value Types** 417
- 10 Well-Formed Types** 451
- 11 Exception Handling** 511
- 12 Generics** 533

13	Delegates and Lambda Expressions	587
14	Events	625
15	Collection Interfaces with Standard Query Operators	653
16	LINQ with Query Expressions	705
17	Building Custom Collections	727
18	Reflection, Attributes, and Dynamic Programming	769
19	Introducing Multithreading	813
20	Programming the Task-Based Asynchronous Pattern	853
21	Iterating in Parallel	895
22	Thread Synchronization	913
23	Platform Interoperability and Unsafe Code	947
24	The Common Language Infrastructure	973
	<i>Index</i>	<i>995</i>
	<i>Index of 8.0 Topics</i>	<i>1039</i>
	<i>Index of 7.0 Topics</i>	<i>1041</i>
	<i>Index of 6.0 Topics</i>	<i>1043</i>



Contents

Figures xvii

Tables xix

Foreword xxi

Preface xxiii

Acknowledgments xxxv

About the Author xxxvii

1 Introducing C# 1

Hello, World 2

C# Syntax Fundamentals 13

Working with Variables 22

Console Input and Output 26

Managed Execution and the Common Language Infrastructure 34

Multiple .NET Frameworks 39

2 Data Types 45

Fundamental Numeric Types 46

More Fundamental Types 55

Conversions between Data Types 72

3 More with Data Types 81

Categories of Types 81

Declaring Types That Allow null 84

Implicitly Typed Local Variables 89

Tuples 92

Arrays 98

4 Operators and Flow Control 121

- Operators 122
- Introducing Flow Control 138
- Code Blocks ({}) 144
- Code Blocks, Scopes, and Declaration Spaces 147
- Boolean Expressions 149
- Programming with null 155
- Bitwise Operators (<<, >>, |, &, ^, ~) 162
- Control Flow Statements, Continued 168
- Jump Statements 180
- C# Preprocessor Directives 186
- Summary 193

5 Methods and Parameters 195

- Calling a Method 196
- Declaring a Method 203
- The using Directive 209
- Returns and Parameters on Main() 214
- Advanced Method Parameters 217
- Recursion 229
- Method Overloading 231
- Optional Parameters 234
- Basic Error Handling with Exceptions 239
- Summary 253

6 Classes 255

- Declaring and Instantiating a Class 259
- Instance Fields 262
- Instance Methods 265
- Using the this Keyword 266
- Access Modifiers 274
- Properties 276
- Constructors 293
- Non-Nullable Reference Type Properties with Constructors 303

Nullable Attributes	306
Deconstructors	309
Static Members	311
Extension Methods	321
Encapsulating the Data	323
Nested Classes	326
Partial Classes	329
Summary	333
7 Inheritance	335
Derivation	336
Overriding the Base Class	346
Abstract Classes	357
All Classes Derive from <code>System.Object</code>	363
Pattern Matching with the <code>is</code> Operator	365
Pattern Matching within a <code>switch</code> Expression	371
Avoid Pattern Matching When Polymorphism Is Possible	373
Summary	374
8 Interfaces	377
Introducing Interfaces	378
Polymorphism through Interfaces	380
Interface Implementation	384
Converting between the Implementing Class and Its Interfaces	390
Interface Inheritance	390
Multiple Interface Inheritance	393
Extension Methods on Interfaces	394
Versioning	396
Extension Methods versus Default Interface Members	411
Interfaces Compared with Abstract Classes	413
Interfaces Compared with Attributes	415
9 Value Types	417
Structs	422
Boxing	428

Enums 437
Summary 447

10 Well-Formed Types 451

Overriding object Members 451
Operator Overloading 464
Referencing Other Assemblies 472
Encapsulation of Types 479
Defining Namespaces 481
XML Comments 485
Garbage Collection 489
Resource Cleanup 493
Lazy Initialization 508
Summary 510

11 Exception Handling 511

Multiple Exception Types 511
Catching Exceptions 514
Rethrowing an Existing Exception 517
General Catch Block 518
Guidelines for Exception Handling 519
Defining Custom Exceptions 523
Rethrowing a Wrapped Exception 527
Summary 530

12 Generics 533

C# without Generics 534
Introducing Generic Types 539
Constraints 553
Generic Methods 568
Covariance and Contravariance 573
Generic Internals 580
Summary 585

- 13 Delegates and Lambda Expressions 587**
 - Introducing Delegates 588
 - Declaring Delegate Types 592
 - Lambda Expressions 600
 - Statement Lambdas 601
 - Anonymous Methods 606
 - Delegates Do Not Have Structural Equality 608
 - Outer Variables 611
 - Expression Trees 616
 - Summary 623
- 14 Events 625**
 - Coding the Publish–Subscribe Pattern with Multicast Delegates 626
 - Understanding Events 641
 - Summary 651
- 15 Collection Interfaces with Standard Query Operators 653**
 - Collection Initializers 654
 - What Makes a Class a Collection: IEnumerable 657
 - Standard Query Operators 663
 - Anonymous Types with LINQ 695
 - Summary 704
- 16 LINQ with Query Expressions 705**
 - Introducing Query Expressions 706
 - Query Expressions Are Just Method Invocations 724
 - Summary 726
- 17 Building Custom Collections 727**
 - More Collection Interfaces 728
 - Primary Collection Classes 731
 - Providing an Indexer 750
 - Returning null or an Empty Collection 753
 - Iterators 753
 - Summary 768

18 Reflection, Attributes, and Dynamic Programming 769

- Reflection 770
- nameof Operator 781
- Attributes 783
- Programming with Dynamic Objects 800
- Summary 811

19 Introducing Multithreading 813

- Multithreading Basics 815
- Asynchronous Tasks 822
- Canceling a Task 843
- Working with System.Threading 850
- Summary 851

20 Programming the Task-Based Asynchronous Pattern 853

- Synchronously Invoking a High-Latency Operation 854
- Asynchronously Invoking a High-Latency Operation Using the TPL 856
- The Task-Based Asynchronous Pattern with `async` and `await` 861
- Introducing Asynchronous Return of `ValueTask<T>` 867
- Asynchronous Streams 870
- `IAsyncDisposable` and the `await using` Declaration and Statement 874
- Using LINQ with `IAsyncEnumerable` 875
- Returning `void` from an Asynchronous Method 877
- Asynchronous Lambdas and Local Functions 881
- Task Schedulers and the Synchronization Context 887
- `async/await` with the Windows UI 890
- Summary 893

21 Iterating in Parallel 895

- Executing Loop Iterations in Parallel 895
- Running LINQ Queries in Parallel 905
- Summary 911

22 Thread Synchronization 913

Why Synchronization? 914

Timers 943

Summary 945

23 Platform Interoperability and Unsafe Code 947

Platform Invoke 948

Pointers and Addresses 960

Executing Unsafe Code via a Delegate 971

Summary 972

24 The Common Language Infrastructure 973

Defining the Common Language Infrastructure 974

CLI Implementations 975

.NET Standard 978

Base Class Library 979

C# Compilation to Machine Code 979

Runtime 982

Assemblies, Manifests, and Modules 986

Common Intermediate Language 989

Common Type System 990

Common Language Specification 991

Metadata 991

.NET Native and Ahead of Time Compilation 993

Summary 993

*Index 995**Index of 8.0 Topics 1039**Index of 7.0 Topics 1041**Index of 6.0 Topics 1043*

This page intentionally left blank



Figures

- FIGURE 1.1:** *Installing the C# extension for Visual Studio Code* 4
FIGURE 1.2: *The Create a new project dialog box* 7
FIGURE 1.3: *The new project dialog* 7
FIGURE 1.4: *Dialog that shows the Program.cs file* 8
- FIGURE 3.1:** *Value types contain the data directly* 82
FIGURE 3.2: *Reference types point to the heap* 83
- FIGURE 4.1:** *Corresponding placeholder values* 162
FIGURE 4.2: *Calculating the value of an unsigned byte* 162
FIGURE 4.3: *Calculating the value of a signed byte* 163
FIGURE 4.4: *The numbers 12 and 7 represented in binary* 165
FIGURE 4.5: *Collapsed region in Microsoft Visual Studio .NET* 193
- FIGURE 5.1:** *Exception-handling control flow* 243
- FIGURE 6.1:** *Class hierarchy* 258
- FIGURE 7.1:** *Refactoring into a base class* 337
- FIGURE 8.1:** *Working around single inheritance with aggregation and interfaces* 396
- FIGURE 9.1:** *Value types contain the data directly* 418
FIGURE 9.2: *Reference types point to the heap* 420
- FIGURE 10.1:** *Identity* 456
FIGURE 10.2: *The Project menu* 476

- FIGURE 10.3: *The Browse filter* 477
- FIGURE 10.4: *XML comments as tips in Visual Studio IDE* 485

- FIGURE 13.1: *Delegate types object model* 597
- FIGURE 13.2: *Anonymous function terminology* 601
- FIGURE 13.3: *The lambda expression tree type* 619
- FIGURE 13.4: *Unary and binary expression tree types* 619

- FIGURE 14.1: *Delegate invocation sequence diagram* 635
- FIGURE 14.2: *Multicast delegates chained together* 637
- FIGURE 14.3: *Delegate invocation with exception sequence diagram* 638

- FIGURE 15.1: *A class diagram of the `IEnumerator<T>` and `IEnumerator` interfaces* 658
- FIGURE 15.2: *Sequence of operations invoking lambda expressions* 675
- FIGURE 15.3: *Venn diagram of inventor and patent collections* 679

- FIGURE 17.1: *Collection class hierarchy* 729
- FIGURE 17.2: *`List<>` class diagrams* 732
- FIGURE 17.3: *Dictionary class diagrams* 739
- FIGURE 17.4: *Sorted Collections* 746
- FIGURE 17.5: *`Stack<T>` class diagram* 747
- FIGURE 17.6: *`Queue<T>` class diagram* 748
- FIGURE 17.7: *`LinkedList<T>` and `LinkedListNode<T>` class diagrams* 749
- FIGURE 17.8: *Sequence diagram with `yield return` application* 758

- FIGURE 18.1: *`MemberInfo` derived classes* 778

- FIGURE 19.1: *Clock speeds over time* 813
- FIGURE 19.2: *Deadlock timeline* 822
- FIGURE 19.3: *`CancellationTokenSource` and `CancellationToken` class diagrams* 846

- FIGURE 20.1: *Control flow within each task* 866
- FIGURE 20.2: *`IAsyncEnumerable<T>` and related interfaces* 873

- FIGURE 23.1: *Pointers contain the address of the data* 963

- FIGURE 24.1: *Compiling C# to machine code* 981
- FIGURE 24.2: *Assemblies with the modules and files they reference* 988



Tables

TABLE 1.1:	<i>C# Keywords</i>	14
TABLE 1.2:	<i>C# Comment Types</i>	32
TABLE 1.3:	<i>Predominant .NET Framework Implementations</i>	39
TABLE 1.4:	<i>C# and .NET Versions</i>	41
TABLE 2.1:	<i>Integer Types</i>	46
TABLE 2.2:	<i>Floating-Point Types</i>	48
TABLE 2.3:	<i>Decimal Type</i>	49
TABLE 2.4:	<i>Escape Characters</i>	57
TABLE 2.5:	<i>string Static Methods</i>	63
TABLE 2.6:	<i>string Methods</i>	63
TABLE 3.1:	<i>Sample Code for Tuple Declaration and Assignment</i>	93
TABLE 3.2:	<i>Array Highlights</i>	99
TABLE 3.3:	<i>Common Array Coding Errors</i>	117
TABLE 4.1:	<i>Control Flow Statements</i>	139
TABLE 4.2:	<i>Relational and Equality Operators</i>	150
TABLE 4.3:	<i>Conditional Values for the XOR Operator</i>	153
TABLE 4.4:	<i>Checking for null</i>	156
TABLE 4.5:	<i>Preprocessor Directives</i>	186
TABLE 4.6:	<i>Sample Warnings</i>	187
TABLE 4.7:	<i>Operator Order of Precedence</i>	194
TABLE 5.1:	<i>Common Namespaces</i>	199
TABLE 5.2:	<i>Common Exception Types</i>	247
TABLE 6.1:	<i>Nullable Attributes</i>	307

TABLE 7.1: *Why the New Modifier?* 349

TABLE 7.2: *Members of System.Object* 364

TABLE 7.3: *Type, var, and const Pattern Matching with the is Operator* 367

TABLE 8.1: *Default Interface Refactoring Features* 401

TABLE 8.2: *Comparing Abstract Classes and Interfaces* 413

TABLE 9.1: *Boxing Code in CIL* 429

TABLE 10.1: *Accessibility Modifiers* 481

TABLE 13.1: *Lambda Expression Notes and Examples* 605

TABLE 15.1: *Simpler Standard Query Operators* 693

TABLE 15.2: *Aggregate Functions on System.Linq.Enumerable* 693

TABLE 19.1: *List of Available TaskContinuationOptions Enums* 831

TABLE 22.1: *Sample Pseudocode Execution* 916

TABLE 22.2: *Interlocked's Synchronization-Related Methods* 927

TABLE 22.3: *Execution Path with ManualResetEvent Synchronization* 936

TABLE 22.4: *Concurrent Collection Classes* 938

TABLE 24.1: *Implementations of the CLI* 975

TABLE 24.2: *Common C#-Related Acronyms* 993



Foreword

WELCOME TO ONE OF THE MOST VENERABLE and trusted franchises you could dream of in the world of C# books—and probably far beyond! Mark Michaelis’s *Essential C#* book has been a classic for years, but it was yet to see the light of day when I first got to know Mark.

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

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

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

So instead of the quick sugar rush of affirmation I might have expected, I got to have a frank and wholesome conversation, the first of many over

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

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

Enjoy!

—Mads Torgersen,
C# Lead Designer,
Microsoft



Preface

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

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

This book has three main functions.

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



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

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

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

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

Target Audience for This Book

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

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

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

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

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

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

Features of This Book

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

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

C# Coding Guidelines

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

Guidelines

DO ensure that equal objects have equal hash codes.

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

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

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

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

Code Samples

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

Here is a sample code listing.

Begin 2.0

LISTING 1.19: Commenting Your Code

```

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

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

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

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

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

```

The formatting is as follows.

- Comments are shown in italics.

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

- Keywords are shown in bold.

```
static void Main()
```

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

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

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

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

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

```
// ...
```

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

OUTPUT 1.7

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

Your full name is Inigo Montoya.
```

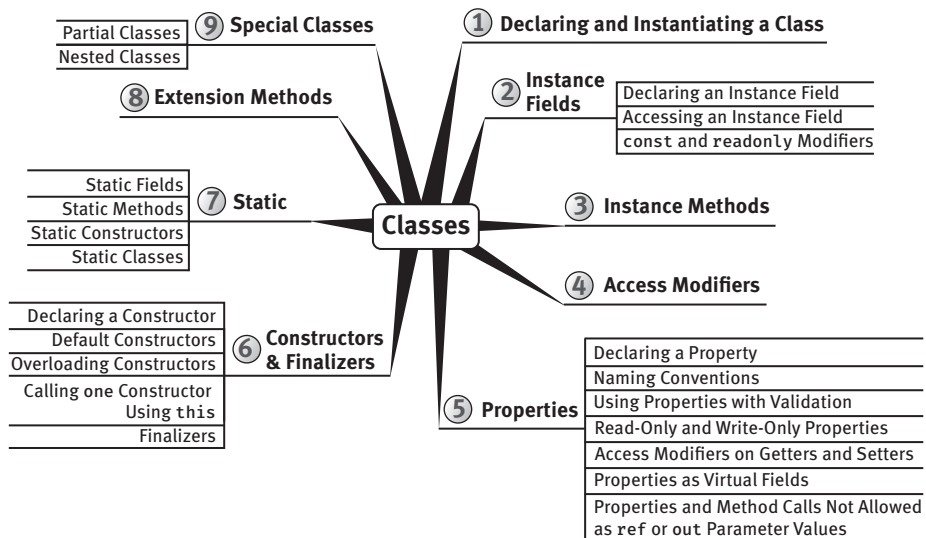
Although it might have been convenient to provide full code samples that you could copy into your own programs, doing so would detract from your learning a particular topic. Therefore, you need to modify the code samples before you can incorporate them into your programs. The core omission is error checking, such as exception handling. Also, code samples

do not explicitly include using System statements. You need to assume the statement throughout all samples.

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

Mind Maps

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



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

Helpful Notes

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

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

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

How This Book Is Organized

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

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

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

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

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

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

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

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

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

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

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

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

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

Register your copy of *Essential C# 8.0* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780135972267) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.



Acknowledgments

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

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

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

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

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

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

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

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



About the Author

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

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

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

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

About the Technical Editors

Kevin Bost is a successful Microsoft MVP and senior software architect at IntelliTect. He has been instrumental in the building of several innovative products, including System.CommandLine, Moq.AutoMocker, and ShowMeTheXAML. When not at work, Kevin can be found online mentoring other developers on YouTube ([youtube.keboo.dev](https://www.youtube.com/channel/UCkeboo-dev)) and maintaining the



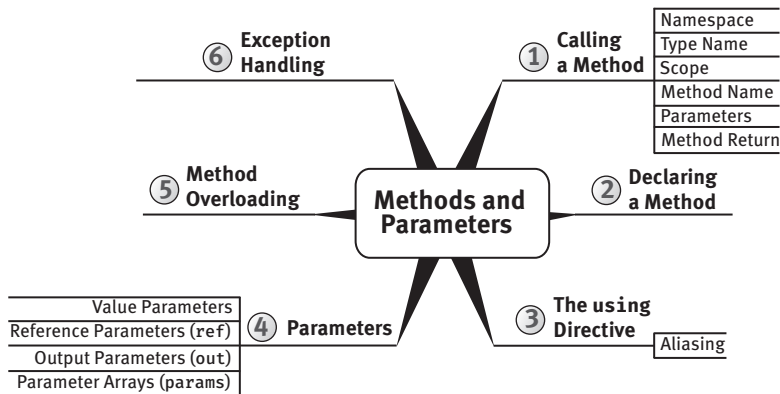
popular Material Design in XAML toolkit (<http://materialdesigninxaml.net/>). He also enjoys board games, Ultimate Frisbee, and riding his motorcycle.

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

5

Methods and Parameters

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



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

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

Calling a Method

■ BEGINNER TOPIC

What Is a Method?

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

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

LISTING 5.1: Grouping Statements into Methods

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

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

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

Guidelines

DO give methods names that are verbs or verb phrases.

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

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

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

LISTING 5.2: A Simple Method Call

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

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

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

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

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

Namespaces

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

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

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

TABLE 5.1: Common Namespaces

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

Begin 4.0

continues

TABLE 5.1: Common Namespaces (continued)

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

End 4.0

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

Guidelines

DO use PascalCasing for namespace names.

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

Type Name

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

1. Or base class.

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

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

Scope

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

Method Name

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

Parameters and Arguments

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

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

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

Method Return Values

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

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

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

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

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

Statement versus Method Call

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

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

■ NOTE

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

Declaring a Method

Begin 6.0

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

LISTING 5.4: Declaring a Method

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

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

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

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

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

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



```

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

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

```

OUTPUT 5.1

```

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

```

End 6.0

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

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

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

BEGINNER TOPIC**Refactoring into Methods**

Moving a set of statements into a method instead of leaving them inline within a larger method is a form of **refactoring**. Refactoring reduces code duplication, because you can call the method from multiple places instead of duplicating the code. Refactoring also increases code readability. As part

of the coding process, it is a best practice to continually review your code and look for opportunities to refactor. This involves looking for blocks of code that are difficult to understand at a glance and moving them into a method with a name that clearly defines the code's behavior. This practice is often preferred over commenting a block of code, because the method name serves to describe what the implementation does.

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

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

Formal Parameter Declaration

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

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

Guidelines

DO use camelCasing for parameter names.

Method Return Type Declaration

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

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

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

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

```

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

```

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

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

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

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

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

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

Begin 7.0

LISTING 5.6: Returning Multiple Values Using a Tuple

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

```

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

```

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

End 7.0

Expression Bodied Methods

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

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

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

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

End 6.0

Language Contrast: C++—Header Files

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

BEGINNER TOPIC

Namespaces

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

The using Directive

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

LISTING 5.7: using Directive Example

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

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

The results of Listing 5.7 appear in Output 5.2.

OUTPUT 5.2

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

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

Language Contrast: Java—Wildcards in the `import` Directive

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

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

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

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

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

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

One interesting effect of the `using System` directive is that the string data type can be identified with varying case: `String` or `string`. The former version relies on the `using System` directive, and the latter uses the `string` keyword. Both are valid C# references to the `System.String` data

type, and the resultant Common Intermediate Language (CIL) code is unaffected by which version is chosen.²

■ ADVANCED TOPIC

Nested using Directives

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

LISTING 5.8: Specifying the using Directive inside a Namespace Declaration

```
namespace EssentialCSharp
{
    using System;

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

The results of Listing 5.8 appear in Output 5.3.

OUTPUT 5.3

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

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

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

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

using static Directive

Begin 6.0

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

LISTING 5.9: using static Directive

```
using static System.Console;

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

        WriteLine("Hey you!");

        Write("Enter your first name: ");

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

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

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

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

End 6.0

Aliasing

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

LISTING 5.10: Declaring a Type Alias

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

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

        // ...
    }
}
```

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

LISTING 5.11: Declaring a Type Alias with the Same Name

```
using System;
using System.Threading;

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

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

        // ...
    }
}
```

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

Returns and Parameters on Main()

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

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

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

LISTING 5.12: Passing Command-Line Arguments to Main

```

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

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

```

The results of Listing 5.12 appear in Output 5.4.

OUTPUT 5.4

```

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

```

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

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

■ ADVANCED TOPIC

Disambiguate Multiple `Main()` Methods

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

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

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

■ BEGINNER TOPIC

Call Stack and Call Site

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

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

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

Advanced Method Parameters

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

Value Parameters

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

LISTING 5.13: Passing Variables by Value

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

        fullName = Combine(driveLetter, folderPath, fileName);

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

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

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

OUTPUT 5.5

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

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

■ BEGINNER TOPIC

Matching Caller Variables with Parameter Names

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

■ ADVANCED TOPIC

Reference Types versus Value Types

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

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

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

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

Reference Parameters (ref)

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

LISTING 5.14: Passing Variables by Reference

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

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

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

OUTPUT 5.6

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


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

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

Output Parameters (out)

Begin 7.0

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

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

LISTING 5.15: Passing Variables Out Only

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

```

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

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

        // ...

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

```

Output 5.7 shows the results of Listing 5.15.

OUTPUT 5.7

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

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

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

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

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

Prior to C# 7.0's tuple syntax, a developer of a method might declare one or more `out` parameters to get around the restriction that a method may have only one return type; a method that needs to return two values can do so by returning one value normally, as the return value of the method,

and a second value by writing it into an aliased variable passed as an out parameter. Although this pattern is both common and legal, there are usually better ways to achieve that aim. For example, if you are considering returning two or more values from a method and C# 7.0 is available, it is likely preferable to use C# 7.0 tuple syntax. Prior to that, consider writing two methods, one for each value, or still using the `System.ValueTuple` type but without C# 7.0 syntax.

NOTE

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

7.0

Read-Only Pass by Reference (in)

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

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

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

Begin 7.2

End 7.2

Return by Reference

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

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

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

```

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

- Properties that return a reference are allowed:

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

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

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

Parameter Arrays (params)

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

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

LISTING 5.17: Passing a Variable Parameter List

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

        // ...

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

        // ...

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

        // ...
    }
}
```

```

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

```

```

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

```

Output 5.8 shows the results of Listing 5.17.

OUTPUT 5.8

```

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

```

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

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

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

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

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

There are a few notable characteristics of the parameter array:

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

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

Guidelines

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

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

Recursion

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

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

```
#nullable enable
using System.IO;

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

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

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

```

        return lineCount;
    }

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

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

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

```

Output 5.9 shows the results of Listing 5.18.

OUTPUT 5.9

```
104
```

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

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

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

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

■ BEGINNER TOPIC

Infinite Recursion Error

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

A common pattern for recursion using pseudocode is as follows:

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

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

Method Overloading

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

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

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

■ NOTE

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

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

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

```
#nullable enable
using System.IO;

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

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

```

        System.Console.WriteLine(totalLineCount);
    }

```

```

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

```

```

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

```

```

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

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

    return lineCount;
}

```

```

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

```

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

```

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

```

OUTPUT 5.10

```

>LineCounter.exe .\ *.cs
28

```

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

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

NOTE

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

Optional Parameters

Begin 4.0

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

6. Introduced in C# 4.0.

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

LISTING 5.20: Methods with Optional Parameters

```
#nullable enable
using System.IO;

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

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

        System.Console.WriteLine(totalLineCount);
    }

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

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

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



```

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

    return lineCount;
}

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

```

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

```

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

```

4.0

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

Guidelines

DO provide good defaults for all parameters where possible.

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

CONSIDER organizing overloads from the simplest to the most complex.

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

LISTING 5.21: Specifying Parameters by Name

```
#nullable enable
using System.IO;

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

    public static void DisplayGreeting(
        string firstName,
        string? middleName = null,
        string? lastName = null
    )
    {
        // ...
    }
}
```

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

7. Introduced in C# 4.0.

Guidelines

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

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

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

End 4.0

ADVANCED TOPIC

Method Resolution

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

For example, a method that takes a `double` parameter will be chosen over a method that takes an `object` parameter if the caller passes

an argument of type `int`. This is because `double` is more specific than `object`. There are objects that are not doubles, but there are no doubles that are not objects, so `double` must be more specific.

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

For example, given the following methods

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

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

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

Basic Error Handling with Exceptions

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

LISTING 5.22: Converting a string to an int

```
using System;

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

```

    int age;

    Console.WriteLine("Hey you!");

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

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

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

```

Output 5.11 shows the results of Listing 5.22.

OUTPUT 5.11

```

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

```

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

BEGINNER TOPIC

42 as a String versus 42 as an Integer

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

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

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

Trapping Errors

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

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

OUTPUT 5.12

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

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

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

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

LISTING 5.23: Catching an Exception

```
using System;

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

```

    int age;
    int result = 0;

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

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

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

    return result;
}
}

```

OUTPUT 5.13

```

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

```

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

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

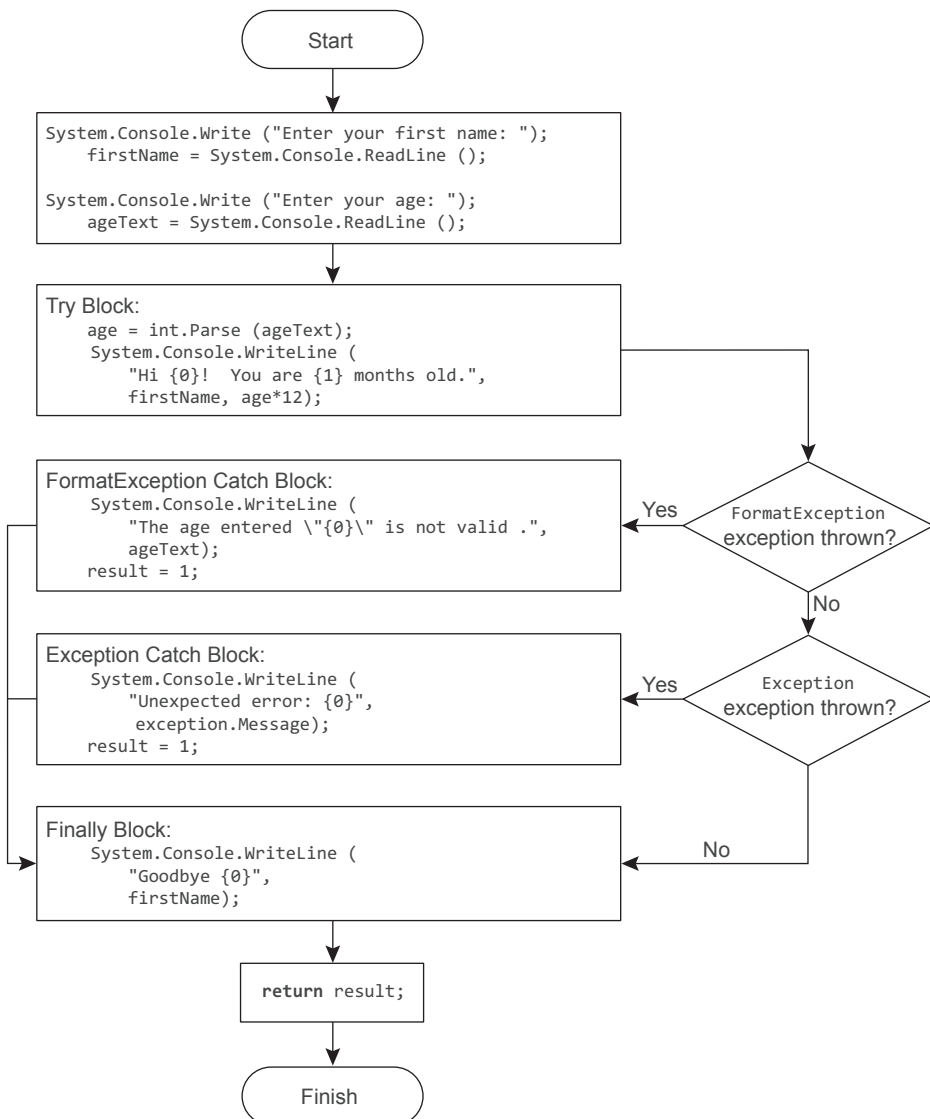


FIGURE 5.1: Exception-handling control flow

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

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

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

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

LISTING 5.24: Finally Block without a Catch Block

```
using System;

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

```

int age;
int result = 0;

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

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

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

return result;
}
}

```

OUTPUT 5.14

```

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

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

```

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

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

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

Guidelines

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

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

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

■ ADVANCED TOPIC

Exception Class Inheritance

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

8. Starting in C# 2.0.

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

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

TABLE 5.2: Common Exception Types

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

continues

TABLE 5.2: Common Exception Types (*continued*)

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

■ ADVANCED TOPIC

General Catch

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

LISTING 5.25: General Catch Blocks

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

```

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

```

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

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

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

Guidelines

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

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

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

Reporting Errors Using a `throw` Statement

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

LISTING 5.26: Throwing an Exception

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

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

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

OUTPUT 5.15

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

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

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

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

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

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

LISTING 5.27: Rethrowing an Exception

```

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

```

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

Guidelines

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

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

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

Avoid Using Exception Handling to Deal with Expected Situations

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

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

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

```

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

```

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

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

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

Guidelines

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

Summary

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

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

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

This page intentionally left blank



Index

Symbols

- & (ampersands). *See* Ampersands (&)
- <> (angle brackets)
 - generics, 540
 - XML, 33
- * (asterisks). *See* Asterisks (*)
- @ (at signs)
 - identifiers, 16–17
 - verbatim strings, 60
- \ (backslashes)
 - escape sequence, 57–58
 - as literals, 60
- ^ (carets). *See* Carets (^)
- : (colons). *See* Colons (:)
- , (commas). *See* Commas (,)
- { } (curly braces). *See* Curly braces ({})
- \$ (dollar signs) for string interpolation, 28, 61–62
- " (double quotes)
 - escape sequence, 57, 59
 - strings, 59–60
- = (equals signs). *See* Equals signs (=)
- ! (exclamation points). *See* Exclamation points (!)
- / (forward slashes). *See* Forward slashes (/)
- > (greater than signs). *See* Greater than signs (>)
- # (hash symbols) for preprocessor directives, 186
- (hyphens). *See* Minus signs (-)
- < (less than signs). *See* Less than signs (<)
- (minus signs). *See* Minus signs (-)
- () (parentheses). *See* Parentheses ()
- % (percent signs)
 - compound assignment, 132
 - overriding, 466, 468
 - precedence, 124
 - remainder operation, 123
- . (periods)
 - fully qualified method names, 198
 - nested namespaces, 210, 483
 - null-conditional operator, 158
- + (plus signs). *See* Plus signs (+)
- ? (question marks)
 - command-line option, 783
 - conditional operators, 154–155
 - null-coalescing operator, 157–158
 - null-conditional operators, 158–160
- ;(semicolons)
 - ending statements, 20
 - for loops, 171–172
 - preprocessor directives, 189
- ' (single quotes)
 - characters, 57
 - escape sequence, 57
- [] (square brackets)
 - arrays, 101–103, 108
 - attributes, 783–784
 - indexers, 750
 - null-conditional operators, 158–160
- ~ (tildes)
 - complement operator, 167
 - finalizers, 494
 - list searches, 736
 - overriding, 468

`_` (underscores). *See* Underscores (`_`)

`|` (vertical bars)

- bitwise operators, 164
- compound assignment, 167
- logical operators, 151–152
- overriding, 466, 468

A

`\a` escape sequence, 58

Abstract classes

- defining, 357–360
- vs. interfaces, 413–414
- overview, 357
- polymorphism, 361–363

abstract modifier

- interface refactoring, 405
- limitations, 357, 386

Access modifiers

- base class overriding, 353–354
- classes, 274–276
- derivation, 341–344
- finalizers, 495
- getters and setters, 289–291
- type declarations, 479–481

Accessing

- arrays, 100, 106
- base members, 354–355
- instance fields, 264–265
- properties, 425–426
- referent type members, 970
- static fields, 312–313

Accuracy of floating-point types, 128–129

Acronyms as identifiers, 15–16

Action delegate, 592–594

Activation frames, 216

`Add()` method

- collection initializers, 298, 654–655
- dictionaries, 739
- interlocks, 927
- lists, 733

Addition

- binary operator, 123
- compound assignment, 132
- increment operator, 133
- strings, 126–127
- unary operator, 122–123

Address operator (`&`), 965

Addresses and pointers. *See* Pointers

`AggregateException` type, 247

- asynchronous requests, 859–860
- parallel loops, 900–901

PLINQ queries, 908, 911

publish-subscribe pattern, 640

tasks, 836–840, 847

wrapped exceptions, 517

Aggregation for inheritance, 345

Ahead of time (AOT) compilation, 993

Alerts escape sequence, 58

Aliases

- command-line option, 789–793
- namespaces, 213–214

Allow Unsafe Code option, 963

`AllowMultiple` parameter, 795

`AllowNull` attribute, 307

`AllowUnsafeBlocks` option, 962

Alternative statements for `if`

- statements, 142

Ampersands (`&`)

- address operator, 965

- bitwise operators, 164

- compound assignment, 167

- logical operators, 152

- overriding, 466, 468

AND operations

- bitwise operators, 164–165

- enums, 444–445

- logical operators, 152

Angle brackets (`<>`)

- generics, 540

- XML, 33

Anonymous functions, 600, 602

Anonymous methods, 600

- internals, 610–611

- lambda expressions as replacement
for, 587–588

- overview, 606–607

- parameterless, 607–608

Anonymous types

- collection initializers, 702–703

- generating, 702

- local variables, 697–701

- overview, 91, 695–696

- selecting into, 698–699

- type safety and immutability,
700–701

Antecedent tasks, 830–831

`Any()` operator, 672

APIs (application programming
interfaces)

- documentation, 770

- as frameworks, 40

- wrapper calls, 959

- AppDomain.CurrentDomain class
 - exceptions, 514
 - process exits, 506
- AppDomains for unhandled exceptions, 841–843
- Append() method, 69
- AppendFormat() method, 69
- Applicable calls, 238
- ApplicationException type, 514
- __arglist keyword, 17
- args parameter for Main(), 18–19
- ArgumentException type, 523
 - description, 247
 - nameof operator, 514
 - properties, 285
 - throwing exceptions, 511–512
- ArgumentNullException type, 513–514
 - description, 247
 - properties, 285
- ArgumentOutOfRangeException type, 513–514
- Arguments
 - generics, 540
 - methods, 197–198, 201
 - named, 237–238
 - passed by value, 217
 - passing methods as, 590–592, 596
- Arithmetic operations, characters in, 127–128
- Arithmetic operators, 123–124
- ArithmeticException type, 248
- Arity
 - generic methods, 569, 581–582
 - tuples, 550–552
 - type parameters, 550
- Array accessors, 106
- Arrays
 - anonymous types, 702–703
 - assigning, 100
 - collection conversion to, 731
 - common errors, 117–119
 - declaring, 98–99, 101–102, 108–109
 - foreach loops, 657–658
 - forward accessing, 100
 - instance members, 114–115
 - instantiating and assigning, 102–106
 - length, 109–110
 - methods, 112–114
 - null-conditional operators, 159
 - overview, 99
 - parameters, 226–229, 552
 - ranges, 101, 111–112
 - redimensioning, 114
 - reverse accessing, 100
 - strings as, 115–117
 - unsafe covariance, 580
 - working with, 106–109
- ArrayTypeMismatchException type, 248
- as operator, 373–374
- ASP.NET, 977
- AsParallel() operator, 670, 906–907, 911
- AspNetSynchronizationContext
 - type, 889
- Assemblies
 - APIs, 40
 - attributes, 784–785
 - boundaries, 479
 - building, 10–11
 - CIL, 34
 - CLI, 35, 986–989
 - constants, 324
 - extensions, 11
 - references, 472–479
- AssemblyCopyrightAttribute, 787
- AssemblyFileVersionAttribute, 787
- AssemblyVersionAttribute, 786–787
- Assert() method, 130
- Assigning
 - arrays, 100, 102–106
 - nulls to references, 71
 - nulls to strings, 70
 - pointers, 964–965
 - tuples, 92–96
 - variables, 22–24
- Assignment operators
 - associativity, 125
 - compound, 132, 167
 - delegates, 634, 641–642
 - null-coalescing, 157–158
 - overriding, 468
- Associations
 - data with static members, 314
 - with instance fields, 263
- Associativity of operators, 124–125
- Asterisks (*)
 - comments, 32
 - compound assignment, 132
 - indirection operators, 968
 - multiplication, 123
 - overriding, 466–468
 - pointers, 963
 - precedence, 124

- Async() method, 875
 - async operator, 861–867, 890–891
 - async return types, 869–870, 877–880
 - async void methods, 877–880
 - AsyncEnumerable class, 875–877
 - Asynchronous invocation, 818
 - Asynchronous lambdas, 881–887
 - Asynchronous methods return types, 877–881
 - Asynchronous pattern. *See* Task-based Asynchronous Pattern (TAP)
 - Asynchronous streams, 870–874
 - Asynchronous tasks
 - complexities from, 822–823
 - introduction, 824–828
 - task continuation, 829–836
 - task exceptions, 836–840
 - thread exceptions, 840–843
 - TPL, 823–824
 - AsyncState property, 828
 - At signs (@)
 - identifiers, 16–17
 - verbatim strings, 60
 - Atomic operations in multithreading, 819–820
 - AttachedToParent task continuation option, 832
 - Attributes
 - custom, 787, 992
 - description, 769
 - enum, 443
 - FlagsAttribute, 795–797
 - initializing, 789–793
 - vs. interfaces, 415
 - named parameters, 794–795
 - nullable, 306–309
 - predefined, 797
 - properties, 783–787
 - restricting, 793–794
 - retrieving, 787–789
 - System.ConditionalAttribute, 797–799
 - System.ObsoleteAttribute, 799–800
 - AttributeTargets flag, 794, 799
 - AttributeUsageAttribute class, 793–795, 797
 - Automatically implemented properties
 - read-only, 287, 305–306, 325–326
 - reference type, 305–306
 - structs, 423–424, 426
 - working with, 279–281
 - AutoResetEvent event, 937
 - Average() function, 693
 - Await() method, 876–877
 - await operator
 - iterating, 892–893
 - task-based asynchronous pattern, 861–867, 890–891
 - await using statement, 874–875
 - AwaitAsync() method, 875, 877
 - AwaitWithCancellationAsync() method, 875–876
- ## B
- \b escape sequence, 58
 - Backslashes (\)
 - escape sequence, 57–58
 - as literals, 60
 - Backspace escape sequence, 58
 - Base Class Library (BCL)
 - description, 977, 993
 - purpose, 35, 979
 - type names, 46–49, 56, 59
 - Base class overriding
 - constructor invocation, 355–356
 - members, 353–355
 - new modifier, 349–353
 - overview, 346
 - sealed modifier, 353–354
 - virtual modifier, 346–349
 - Base classes
 - derived object conversion to, 339
 - inheritance, 257–258
 - protected member access in, 344
 - Base types
 - casting between derived types, 339–340
 - classes, 258–259
 - description, 336
 - BCL (Base Class Library)
 - description, 977, 993
 - purpose, 35, 979
 - type names, 46–49, 56, 59
 - Binary digits (bits), 162–163
 - Binary expression trees, 618–619
 - Binary operators
 - associativity, 125
 - compound assignment, 132, 468
 - non-numeric types, 126–127
 - null-coalescing, 157
 - overriding, 466–467
 - overview, 123–124
 - relational and equality, 150–151

- Binary values
 - conversions to, 165–167
 - floating-point, 47
 - literals, 53
 - shift operators, 163
 - BinaryExpression expression trees, 623
 - BinarySearch() method
 - arrays, 112–114
 - list searches, 736
 - BinaryTree<T> class
 - constraints, 553–557
 - index operators, 152
 - iterators, 755–756, 759–762
 - parameters, 572
 - Binding
 - dynamic, 801, 806–807
 - extension methods, 412
 - late, 518, 985, 992
 - metadata, 992
 - Bits (binary digits), 162–163
 - Bitwise operators
 - bits and bytes, 162–163
 - complement, 167, 736–737
 - compound assignment, 167
 - enums, 444–445
 - for positions, 181–182
 - shift, 163–164
 - working with, 164–167
 - Block statements. *See* Code blocks
 - BlockingCollection<T> class, 938–939
 - Boolean (bool) type
 - array initial values, 104
 - conversions, 75
 - flow control, 149
 - lambdas, 593
 - logical operators, 151–153
 - overview, 55–56
 - relational and equality operators, 150–151
 - Boolean expressions for if statements, 142
 - Boolean operators
 - logical, 151–155
 - overriding, 468–470
 - Bounds
 - arrays, 109–110
 - floating-point types, 131
 - Boxing
 - avoiding, 435–436
 - generics for, 543
 - idiosyncrasies, 432–434
 - in loops, 448
 - overview, 428–432
 - Brand casing for namespaces, 482
 - Break() method, 905
 - break statements
 - lambda expressions, 606
 - overview, 141
 - switch statements, 177–178
 - working with, 180–181
 - yield break, 764
 - Breaking parallel loops, 904–905
 - Brittle base class problem, 349–351
 - Bubble sorts, 588–602
 - Buffer overflows
 - C# vs. C++, 109–110
 - managed execution, 986
 - type safety, 35, 984
 - unsafe code, 962
 - byte type, 46
 - Bytes, 162–163
- ## C
- C language and C# syntax similarities, 2–3
 - C++ language vs. C#
 - arrays, 101–102
 - Boolean conversions, 75
 - buffer overflows, 109–110
 - delete operator, 262
 - deterministic destruction, 502, 983
 - equality checks, 150
 - global methods, 204
 - global variables and functions, 311
 - header files, 208
 - implicit overriding, 347
 - increment and decrement operators, 136
 - local variable scope, 149
 - Main(), 19
 - methods called during construction, 348–349
 - multiple inheritance, 345
 - operand evaluation, 126
 - operator-only statements, 123
 - pointer declarations, 964
 - preprocessor directives, 186
 - pure virtual functions, 361
 - short type, 47
 - string concatenation, 61
 - structs, 427
 - switch statements, 179

- syntax similarities, 2–3
 - Variant and var, 90
 - void type, 71, 90
- Calculated properties, 287–289
- Call sites, 217
- Call stacks for methods, 216–217
- Callback functions, 587
- CallerMemberName parameter attribute, 782
- Calling
 - collection initializers, 297–298
 - constructors, 294–295
 - external functions, 956–958
 - object initializers, 296–297
- CallSite<T> type, 805–806
- camelCase
 - description, 16
 - parameter names, 205, 268, 300
 - property names, 282
 - tuples, 95
 - variable names, 24, 92
- "can do" relationships, 377–378
- Cancel() method
 - parallel iterations, 903
 - PLINQ queries, 911
 - tasks, 845–846
- Canceling
 - iterations, 763–764
 - parallel loops, 901–903
 - PLINQ queries, 908–911
 - tasks, 843–849
- CancellationToken class
 - asynchronous methods, 885
 - asynchronous streams, 872–873
 - overview, 843–847
 - PLINQ queries, 911
- CancellationToken property, 904
- CancellationTokenSource class, 845–846, 903, 911
- Capacity of lists, 731–733
- Capture() method, 517
- Captured outer variables, 611–612
- Carets (^)
 - arrays, 107–108
 - bitwise operators, 164
 - compound assignment, 167
 - index from end operator, 99, 107, 108, 110, 112, 118
 - logical operators, 152
 - overriding, 466, 468
- Carriage returns
 - escape sequence, 58
 - newlines, 66
- Cartesian products
 - inner joins, 684
 - query expressions, 723
- Case labels
 - goto statements, 185
 - switch statements, 177, 179
- Case sensitivity
 - C++ similarity, 3
 - importance, 2
 - literal suffixes, 52
 - suffixes, 51–52
- Casing
 - identifier formats, 15–16
 - local variables, 24
- cast operator and casting
 - base and derived types, 339–340
 - defining, 341
 - enums, 440–441
 - generic methods, 572–573
 - implicit conversions, 76
 - in inheritance chains, 340
 - overriding, 470–471
 - overview, 72–77
- Catch blocks
 - exceptions, 514–516, 518–520
 - general, 248–249
 - rethrowing exceptions, 251
 - working with, 242–245
- Catching exceptions, 241–242, 514–516, 518–519
- cells member in arrays, 115
- Central processing units (CPUs)
 - description, 815
 - multithreading, 814, 818
 - PLINQ queries, 670
- Centralizing constructor initializers, 301–302
- Chains
 - circular, 930
 - constructors, 300–301
 - delegates, 633–634, 637–638, 640
 - expressions, 158–160
 - inheritance, 338–340
 - iterators, 762
 - null-coalescing operator, 158
 - queries, 722
 - tasks, 830–831, 853

- Characters (char) types
 - arithmetic operations, 127–128
 - array initial values, 104
 - description, 23, 56
- Checked conversions, 73–75, 528–530
- checked keyword, 73–75, 529–530
- CheckForOverflowUnderflow property, 529
- Child types, 258, 336
- Church, Alonzo, 608
- CIL. *See* Common Intermediate Language (CIL)
- Circular wait conditions in deadlocks, 930
- Class definitions
 - description, 17
 - guidelines, 17
- class keyword, 259
- Classes
 - abstract, 357–363
 - access modifiers, 274–276
 - associated data, 314
 - concrete, 357
 - constraints, 560–561
 - constructors. *See* Constructors
 - declaring, 259–260
 - deconstructors, 309–311
 - defined, 260
 - delegates, 597–600
 - encapsulation, 261–262, 323–326
 - extension methods, 321–322
 - files for storing and loading, 270–273
 - generics, 540–544, 779–780
 - hierarchy, 257–258
 - inextensible, 320
 - inheritance. *See* Inheritance
 - instance fields, 262–265
 - instance methods, 265–266
 - instantiating, 260–261
 - interfaces. *See* Interfaces
 - libraries, 472–475
 - members, 262
 - methods, 197–198
 - multiple interfaces, 545–546
 - multiple iterators in, 766–767
 - nested, 326–328
 - nullable attributes, 306–309
 - object-oriented programming, 256–259
 - overview, 255–256
 - partial, 329–332
 - properties. *See* Properties
 - refactoring, 336–337
 - sealed, 345
 - static, 319–321
 - static members. *See* Static members
 - vs. structs, 763
 - this keyword, 266–270
- Clear() method, 112, 114
- CLI. *See* Common Language Infrastructure (CLI)
- Clock speeds, 813
- Clone() method, 114–115
- Close() method, 497
- Closed over outer variables, 611–612
- Closures, 614
- CLR (Common Language Runtime)
 - description, 982, 994
 - enums, 440–441
 - namespaces, 481–483
- CLS (Common Language Specification)
 - description, 994
 - libraries, 35
 - metadata, 991
- CLSCompliant attribute, 991
- CLU language, 754
- Clusters, 754
- Code access security, 35
- Code blocks
 - error trapping, 242–244
 - if statements, 144–147
 - loops, 169–170
 - scope, 147–149
- Code editors for preprocessor directives, 191–193
- Code readability. *See* Readability
- CodeAnalysis class, 306–309
- Cold tasks, 825
- Collect() method, 490, 506
- Collections
 - anonymous types, 702–703
 - class categories, 731–733
 - dictionaries, 738–744
 - foreach loops, 174–175, 661–662
 - IEnumerable<T>, 657–663
 - indexers, 750–752
 - initializers, 297–298, 654–657, 702–703
 - interfaces, 653–654, 728–731
 - iterators. *See* Iterators
 - join operations. *See* Join operations
 - linked lists, 749
 - null values, 753

- overview, 727–728
- queues, 748
- searching, 736–738
- shared states, 660
- sorted, 745–747
- sorting, 734–735
- stacks, 747
- standard query operators. *See*
Standard query operators
- thread synchronization, 938–939
- total ordering, 735–736
- Collisions, name, 198
- Colons (:)

 - conditional operators, 154–155
 - constraints, 561
 - constructors, 301
 - derived classes, 336
 - hexadecimal format, 54
 - labels, 185

- COM threading model, 945
- Combine() method
 - delegates, 558–559, 634, 650–651
 - hash codes, 455
 - parameters, 217–218, 220, 226–229
- COMException type, 514
- Command-line arguments, 19, 214–216
- CommandLineInfo class, 773–779
- CommandLineRequiredAttribute, 797
- CommandLineSwitchAliasAttribute, 797
- CommandLineUtils package, 773n
- Commas (,)

 - arrays, 99, 102, 104, 106
 - attributes, 787
 - constraints, 561
 - enums, 447
 - for loops, 173
 - interfaces, 384
 - methods, 201
 - parameters, 205
 - variable declarations, 23, 499

- Comments
 - guidelines, 33
 - overview, 30–33
 - XML, 485–489
- Common Intermediate Language (CIL)

 - anonymous types, 697–698
 - assemblies, 34
 - boxing code, 428–432
 - compilation to, 974–975
 - description, 994
 - events, 649–650
 - general catch blocks, 519
 - generics representation, 581–582
 - ILDASM, 36
 - indexers, 751
 - metadata, 770, 991–992
 - outer variable implementation,
613–614
 - output, 37–39
 - properties, 291–293
 - purpose, 989–990

- Common Language Infrastructure (CLI)

 - assemblies, 986–989
 - Base Class Library, 979
 - CIL, 989–990
 - CLS, 991
 - compilation to, 979–981
 - CTS, 990
 - description, 974–975, 994
 - encapsulation, 985
 - garbage collection, 982–984
 - implementations, 975–976
 - manifests, 986–989
 - metadata, 991–992
 - Microsoft .NET Framework, 975–977
 - modules, 986–989
 - multicast delegates, 636–637
 - namespaces, 482
 - .NET Core, 977–978
 - .NET Native feature, 993
 - .NET Standard version, 978–979
 - overview, 34–35, 973
 - P/Invoke, 948–960
 - performance, 985–986
 - platform portability, 985
 - runtime, 982–984
 - type safety, 984
 - Xamarin compiler, 978

- Common Language Runtime (CLR)

 - description, 982, 994
 - enums, 440–441
 - namespaces, 481–483

- Common Language Specification (CLS)

 - description, 994
 - libraries, 35
 - metadata, 991

- Common Type System (CTS), 34, 990, 994
- Compacting objects in garbage
collection, 490
- Compare() method, 56, 63

- CompareExchange() method, 926–927
- CompareTo() method
 - collection sorts, 734–735
 - interfaces, 556–557
 - tree sorts, 553–554
- Comparing
 - binary tree nodes, 556–557
 - collections items, 734–735
 - compare/exchange pattern, 926–927
 - floating-point types, 129–130
 - strings, 56, 63
- Comparison operators
 - overriding, 464–465
 - pointers, 969
- Compatibility between enum types, 440–441
- Compilation
 - ahead of time, 993
 - assemblies, 10–11
 - disassembling, 36
 - to machine code, 979–981
 - sample output, 37–39
 - source code, 3–8, 34–35
 - static vs. dynamic, 807–808
- CompilerGeneratedAttribute, 293
- Complement operator (~), 167
- Complexity of asynchronous requests, 859–861
- ComponentModel class, 953
- Components in compilation, 980
- Composite formatting
 - description, 29
 - patterns, 65
 - strings, 127
- Compound assignment
 - bitwise, 167
 - mathematical operators, 132
 - overriding, 468
- Compress() method, 379
- Concat() method for strings, 62–63
- Concat() operator for queries, 693
- Concatenating strings, 61–63, 126–127
- Concrete classes, 357
- Concurrent collection classes, 938–939
- Concurrent operations, 817–818
- ConcurrentBag<T> class, 939
- ConcurrentDictionary<TKey, TValue> class, 939
- ConcurrentQueue<T> class, 939
- ConcurrentStack<T> class, 939
- Conditional clauses in catch blocks, 516
- Conditional operators
 - null. *See* Null-conditional operator
 - overriding, 468
 - working with, 154–155
- Conditional types, 55–56
- ConditionalAttribute, 797–799
- ConditionalExpression expression trees, 623
- Conditions
 - for loops, 172
 - if statements, 142
- Consequence statements in if statements, 142
- Console
 - comments, 30–33
 - input, 26–27, 197
 - newlines, 59, 66
 - output, 19, 28–33, 197
 - round-trip formatting, 54–55
 - string methods, 62
- const keyword
 - constants, 137–138
 - encapsulation, 323–324
- ConstantExpression expression trees, 623
- Constants
 - expressions and locals, 137–138
 - pattern matching, 366–368
 - switch statements, 177
- Constraints
 - classes, 560–561
 - constructor, 562–565
 - delegates, 558–559
 - generic methods, 571–572
 - inheritance, 565–567
 - interfaces, 554–557
 - limitations, 567–568
 - multiple, 561–562
 - nonnull, 560
 - operators, 567–568
 - overview, 553–555
 - structs, 560–561
 - type parameters, 557–558
 - unmanaged, 559–560
- Constructed struct types, 560
- Constructors
 - attributes, 789–793
 - base classes, 355–356

- chaining, 300–301
- collection initializers, 297–298
- constraints, 562–565
- declaring, 293–295
- default, 295–296
- finalizers, 298
- generics, 547
- initializers, 300–302
- methods called during, 348–349
- new operator, 294–295
- non-nullable reference type
 - properties, 303–306
- object initializers, 296–297
- overloading, 298–300
- overview, 293
- propagating exceptions from, 507
- static, 316–318
- structs, 424
- Contains() method
 - list searches, 736
 - stack searches, 747
- ContainsGenericParameters property, 779–780
- ContainsKey() method, 741
- ContainsValue() method, 741
- Context switches, 137, 817–820, 850
- Contextual keywords, 15, 766
- Continuation of tasks, 829–836
- continue statements
 - lambda expressions, 606
 - overview, 140
 - switch statements, 177
 - working with, 182–184
- ContinueWith() method
 - asynchronous requests, 859
 - synchronization context, 888–889
 - tasks, 829–831, 834–835, 839–840
- Contravariance
 - delegates, 609
 - generics, 573–580
 - type parameters, 577–579
- Control flow. *See* Flow control
- Conversions
 - as operator, 373–374
 - to binary representation, 165–167
 - Boolean type, 75
 - boxing, 428–432
 - casts. *See* cast operator and casting
 - checked and unchecked, 73–75, 528–530
 - collections to arrays, 731
 - covariant, 574
 - dynamic objects, 803
 - enums, 439–443
 - explicit, 72–73
 - false return values, 77–79
 - implementing classes and interfaces, 390
 - implicit, 75–76
 - overriding operators, 470–472
 - overview, 72
 - type safety, 35
 - without casts, 76–77
- Cooperative cancellation, 843–847
- Copying
 - arrays, 115
 - directories, 316, 321–322
 - null values, 85
 - reference types, 83–84, 421
 - variables, 418–419
- CopyTo() method
 - collections, 730–731
 - directories, 316, 321–322
- Core .NET frameworks, 39
- CoreCLR compiler, 975, 977
- Cores, 670, 815, 817
- Count() method
 - description, 693
 - elements, 671–674
 - index-based collection, 412
- Count property for collections, 730–731
- CountdownEvent event, 937–938
- CountOccurrencesAsync() method, 860–861, 864–866
- Covariance
 - in arrays, 580
 - delegates, 609
 - generics, 573–580
 - out modifier, 575–577
- CPUs (central processing units)
 - description, 815
 - multithreading, 814, 818
 - PLINQ queries, 670
- Create() method, 551
- Cryptographer class, 874
 - .cs file extension, 9
- CTS (Common Type System), 34, 990, 994
- Curly braces ({})
 - arrays, 102–103
 - classes, 259

- code blocks, 144–147
 - code formatting, 22
 - collection initializers, 297, 655–656
 - lambda expressions, 603
 - methods, 18
 - namespaces, 482
 - object initializers, 296
 - pattern matching, 370
 - properties, 278
 - string interpolation, 29, 61–62, 66
 - switch statements, 20
 - type definitions, 17
 - use of, 19
 - Current property for iterators, 764
 - CurrentDomain class
 - exceptions, 514
 - process exits, 506
 - CurrentId property for asynchronous tasks, 828
 - Custom attributes, 787, 992
 - Custom conversions, 340–341
 - Custom exceptions, 523–526
 - Custom objects in dynamic programming, 808–811
 - Custom value types for structs, 422–427
- D**
- Data namespace, 199
 - Data persistence, files for, 270–273
 - Data types. *See* Types
 - DataBinder class, 771
 - Deadlocks
 - avoiding, 929–930
 - causes, 821–823
 - Debug() method, 799
 - DEBUG preprocessor identifier, 799
 - Debugging
 - preprocessor directives for, 188
 - source code, 8–9
 - Decimal type, 49
 - Declaration spaces in code blocks, 147–149
 - Declaring
 - arrays, 98–99, 101–102, 108–109
 - await using statement, 874–875
 - BinaryTree<T> class, 553–554
 - class type constraints, 557–558
 - classes, 259–260
 - constants, 323
 - constructors, 293–295
 - deconstructors, 311
 - delegate types, 592–595
 - enums, 437–438
 - events, 643–644, 648–649
 - external methods, 948–949
 - generic interfaces, 544–545
 - generic methods, 568–569
 - instance fields, 263–264
 - interface constraints, 556–557
 - local variables, 22–23
 - Main() method, 18–19
 - methods, 203–204
 - namespaces, 482–484
 - out variables, 222
 - parameters, 205
 - pointers, 963–964
 - properties, 277–279
 - read-only automatically implemented properties, 325–326
 - read-only fields, 324–325
 - return types, 206–208
 - static constructors, 316–318
 - static fields, 312
 - structs, 422–423
 - tuples, 92–96
 - types from unmanaged structs, 952–953
 - variables, 23–24
 - Deconstructors, 309–311
 - Decrement() method, 137, 927–928
 - Decrement operators
 - description, 133
 - loop example, 133–136
 - thread-safe, 137
 - Default constructors, 295–296
 - Default interface members
 - vs. extension methods, 411–413
 - versioning, 398–400
 - default operator for generics, 547–549
 - default sections in switch statements, 179, 184
 - Default types, 50–52
 - Default values
 - generics, 547–549
 - structs, 424
 - DefaultIfEmpty() method, 688–689
 - Deferred execution
 - LINQ, 672–676
 - query expressions, 711–715
 - #define directive, 187, 189

Defining

- abstract classes, 357–360
- attributes, 786
- calculated properties, 288–289
- cast operators, 341, 470–471
- custom conversions, 340–341
- custom exceptions, 523–526
- enums, 438–439, 445
- finalizers, 494–495
- generic classes, 542
- generic constructors, 547
- generic methods, 568–570
- indexers, 750–752
- iterators, 754
- namespaces, 481–484
- nested classes, 326–328
- partial classes, 329–332
- preprocessor symbols, 189
- publishers, 627–628
- stacks, 537
- static methods, 315–316
- subscriber methods, 626–627

Delay() method, 944

Delegate class, 597–598

delegate keyword, 592

Delegate type, 558–559

Delegates

- constraints, 558–559
- deferred execution, 715
- event notifications, 928–929
- events, 647–648
- vs. expression trees, 620–621
- function pointers to, 960
- instantiating, 595–597
- internals, 597–600, 636–637
- introduction, 588–590
- invoking, 629–632, 634–636
- method returns and pass-by-reference, 640
- nesting, 594–595
- null-conditional operators, 161
- overview, 587–588
- predicates, 666–667
- process exits, 506
- publish-subscribe pattern operators, 633–634
- statement lambdas for, 601–604
- structural equality, 608–609
- tasks, 824

types, 590–594

unsafe code, 971–972

delete operator, 262

Deleting files, 496

Delimited comments, 32

Delimiters, 19–20

DenyChildAttach task continuation

- option, 832

Dependencies in libraries, 10

Dequeue() method, 748

Dereferencing

- description, 83
- null-conditional operator for, 513
- null references, 85–86
- null values, 87
- pointers, 968–970

Derivation

- casting between base and derived types, 339–340
- custom conversions, 340–341
- extension methods, 344
- interfaces, 397–398
- overview, 336–338
- private access modifier, 341–342
- protected access modifier, 342–344
- sealed classes, 345
- single inheritance, 345
 - from System.Object, 363–365

Derived members, 335

Derived objects converted to base class, 339

Derived types

- casting between base types, 339–340
- classes, 258–259
- description, 336

DescriptionAttribute, 786

Deserialize() method, 572–573

Deterministic destruction, 502, 983

Deterministic finalization, 496–500

Deterministic resource cleanup, 262

Diagnostics class

- debugging with, 799
- notifications, 885
- nullable attributes, 306–309
- processes, 815

Diagramming interfaces, 395–396

Dictionaries

- collection initializers, 656
- interfaces, 728–730

- key equality, 743–744
 - sorted, 745–747
 - thread synchronization, 939
 - working with, 738–743
 - Dictionary<TKey, TValue> class
 - collections, 658
 - working with, 738–743
 - Digit separators, 52–53
 - Dimensions for arrays, 102–106, 114
 - Directives
 - dynamic, 803
 - preprocessor. *See* Preprocessor directives
 - strings, 64–65
 - Directories, copying, 316, 321–322
 - DirectoryInfo class
 - extension methods, 321
 - inner joins, 682
 - DirectoryInfoExtension class, 315–316, 321–322
 - DisallowNull attribute, 307
 - Disambiguating multiple Main()
 - methods, 214–216
 - Disassembling programs, 36
 - Discards
 - out parameters, 222
 - tuples, 96
 - DispatcherTimer class, 944
 - Disposability of tasks, 849
 - Dispose() method
 - collections, 661
 - finalization, 500–502
 - resource cleanup, 495, 497–499
 - DisposeAsync() method, 874–875
 - Distinct members in query expressions, 723–724
 - Distinct() operator
 - description, 693
 - query expressions, 723–724
 - Division
 - binary operator, 123
 - compound assignment, 132
 - shift operators, 164
 - by zeros, 131
 - DllImport attribute, 949, 953
 - DLLs (dynamic link libraries), 10–11
 - do/while loops
 - overview, 140
 - working with, 168–170
 - Documentation
 - reflection, 770
 - XML files, 487–489
 - Dollar signs (\$) for string interpolation, 28, 61–62
 - DotGNU Portable.NET compiler, 976
 - Dotnet CLI
 - code building and executing, 12
 - NuGet references, 475–476
 - project and library references, 474
 - working with, 4–6
 - Double quotes (")
 - escape sequence, 57, 59
 - strings, 59–60
 - double type
 - overview, 47–49
 - special characteristics, 128–129
 - DownloadData() method, 856
 - DownloadDataTaskAsync() method, 858–860, 865
 - DownloadString() method, 856
 - Drawing class, 199
 - Duck typing, 662
 - Duplicating interfaces, 545–546
 - dynamic data type principles and behaviors, 803–805
 - Dynamic invocation
 - delegates, 558–559
 - members, 773–779
 - nameof operator, 781–782
 - Dynamic link libraries (DLLs), 10–11
 - Dynamic programming
 - binding, 806–807
 - custom objects, 808–809
 - description, 769
 - overview, 800–801
 - principles and behaviors, 803–805
 - reflection, 801–802
 - vs. static compilation, 807–808
 - DynamicInvoke() method, 558–559
 - DynamicObject class, 808–811
- ## E
- Editors
 - dotnet CLI, 4–6
 - preprocessor directives, 191–193
 - source code, 3–4
 - Visual Studio 2019. *See* Visual Studio 2019

- Elements, counting, 671–672
- `#elif` directive, 186, 188
- `else` clauses, 142
- `#else` directive, 186, 188
- Empty collections, 753
- Empty property for events, 645
- Empty strings, 70
- Encapsulation
 - classes, 261–262
 - CLI, 985
 - `const` modifier, 323–324
 - information hiding, 274–276
 - object-oriented programming, 257
 - with protected interface members, 406–411
 - publications, 642–643
 - `readonly` modifier, 324–326
 - subscriptions, 641–642
 - well-formed types, 479–481
- `EncryptAsync()` method, 874
- `#endif` directive, 188
- `#endregion` directive, 191–193
- `EndsWith()` method, 63
- `Enqueue()` method, 748
- `Enter()` method with monitors, 432, 918–920, 922–924
- Enum class, 439, 442
- enum values
 - conversions, 441–443
 - flags, 443–447
 - `FlagsAttribute`, 795–797
 - metadata, 772–773
 - overview, 437–440
 - parsing, 78
 - type compatibility, 440–441
- Enumerable class
 - delegates, 621
 - element counting, 671–672
 - filters, 667
 - inner joins, 682, 684–688
 - methods, 691–693
 - parallel queries, 907
 - query expressions, 714
 - sorts, 676–678
 - standard query operators, 663
- Enumerator patterns for iterators, 754–755
- Environment class
 - command-line arguments, 19, 216
 - newlines, 66
 - thread synchronization, 933

- Equality
 - C# vs. C++, 150
 - collections, 744–745
 - delegates, 608–609
 - dictionary keys, 743–744
 - floating-point types, 129–132
 - with null, 156–157
 - objects, 456–459
 - operators, 150–151
- `EqualityComparer<T>` class, 463, 743
- `Equals()` method
 - collections, 744–745
 - description, 364
 - with `GetHashCode()`, 453–454, 744
 - implementing, 460–462
 - null values, 87
 - overriding, 455–464
 - reflection, 427
- Equals signs (=)
 - compound assignment, 132, 167, 468
 - delegates, 632–634, 641–642, 650
 - equality operator, 55
 - null checks, 86
 - null-coalescing operator, 157–158
 - overriding, 464–465
 - pointers, 969
 - precedence, 124
 - properties, 278
 - relational and equality operators, 150–151
 - shift operators, 163–164
 - variable assignment, 24–25
- Equi-joins, 688
- `#error` directive, 187, 189–190
- Error handling. *See also* Exceptions
 - catch blocks, 248–249
 - exception class inheritance, 246–248
 - with exceptions, 239–240
 - `P/Invoke`, 953–955
 - publish-subscribe pattern, 637–640
 - throw statements, 249–251
 - trapping, 241–246
- Errors
 - array coding, 117–119
 - preprocessor directives for, 186–191
- Escape sequences, 57–58
- Essential source code, 12–13
- `EssentialCSharp.sln` file, 12
- Etch A Sketch game, 534–537
- Evaluation of operators, 125
- event keyword, 643–644

- EventArgs class, 645–647
- EventHandler<T> class, 644, 647–648, 650–651, 887
- Events
 - coding conventions, 644–647
 - declaring, 643–644, 648–649
 - generics and delegates, 647–648
 - implementation, 650–651
 - internals, 648–650
 - listeners, 835–836
 - overview, 625
 - publication encapsulation, 642–643
 - publish-subscribe pattern. *See* Publish-subscribe pattern
 - purpose, 641
 - subscription encapsulation, 641–642
 - thread synchronization, 928–929
- Exception conditions, 516
- Exceptional continuation, 829
- ExceptionDispatchInfo class
 - asynchronous requests, 860
 - purpose, 517–518
- Exceptions
 - catch blocks, 248–249, 514–516, 518–519
 - class inheritance, 246–248
 - custom, 523–526
 - error trapping, 241–246
 - expected situations, 252–253
 - guidelines, 514, 519–523
 - overview, 239–240, 511
 - parallel loops, 900–901
 - PLINQ queries, 908, 911
 - propagating from constructors, 507
 - publish-subscribe pattern, 637–640
 - rethrowing, 517–518, 522–523, 527–530
 - serializable, 526
 - tasks, 836–840
 - threads, 840–843
 - throw statements, 249–251
 - types, 511–514
- ExceptionServices class, 517
- Exchange() method, 927
- Exclamation points (!)
 - equality operators, 55
 - generics, 582
 - inequality operators, 150–151
 - logical operators, 153
 - null-forgiving operators, 160–161
 - overriding, 464–465, 468
 - pointers, 969
- Excluding code, preprocessor directives
 - for, 188
- Exclusive OR (XOR) operators
 - bitwise, 164–165
 - logical, 152–153
- Executables, stand-alone, 11–12
- ExecuteSynchronously task
 - continuation option, 833
- Execution time, 35
- ExecutionEngineException type, 514
- Exit() method with monitors, 432, 918–920, 923–924
- Expanded form for parameters, 227
- Explicit casts
 - base and derived types, 339
 - enums, 440–441
 - guidelines, 471–472
 - overview, 72–73
- Explicit deterministic resource cleanup, 262
- Explicit member implementation, 386–390
- Exponential notation, 52
- Expression bodied members
 - declaring, 208
 - properties, 279
- Expression trees
 - deferred execution, 715
 - vs. delegates, 620–621
 - description, 616
 - examining, 621–623
 - object graphs, 618–619
 - SQL queries, 617–618
- Expressions
 - constant, 137–138
 - lambda. *See* Lambda expressions
 - query. *See* Query expressions
- Extended Application Markup Language (XAML), 977
- Extensible Markup Language. *See* XML (Extensible Markup Language)
- Extension methods
 - classes, 321–322
 - vs. default interface members, 411–413
 - derivation, 344
 - interfaces, 394–395
- Extensions, file, 9
- External methods and functions
 - calling, 956–958
 - declaring, 948–949

F

- \f escape sequence, 58
- f-reachable (finalization) queues, 298, 500–501
- Factory interfaces for constructor constraints, 564
- FailFast() method, 514
- Failure type, 514
- false operator, overriding, 468–470
- Fat arrow notation (=>) for lambda expressions, 608
- FCL (Framework Class Library), 979, 993–994
- Fibonacci numbers, 168–169
- Fields
 - guidelines, 281–283
 - instance. *See* Instance fields
 - static members, 311–314
 - strings, 66
 - thread synchronization, 926
- FIFO (first in, first out) collections, 748
- FileAttributes class, 443
- FileInfo class
 - inner joins, 682
 - projections, 668–669, 717–718
 - query expressions, 710
- Files
 - deleting, 496
 - extensions, 9
 - Java language vs. C#, 10
 - project, 10
 - storing and loading with, 270–273
- FileStream class
 - finalizers, 495
 - IAsyncDisposable, 874
 - storing and loading files, 270–273
 - ToString(), 452
- Filters
 - Browse, 477
 - collections, 593–594, 666–667, 672–676
 - exceptions, 521–523
 - query expressions, 706, 715–716
- Finalization, deterministic, 496–500
- finalization (f-reachable) queues, 298, 500–501
- finalize() method, 364
- Finalizers
 - constructors, 298
 - generics, 547
 - resource cleanup, 493–496, 500–501
 - structs, 426–427
 - finally blocks in error trapping, 244–246
- FindAll() method, 737–738
- First in, first out (FIFO) collections, 748
- fixed statement for pointers, 966–967
- Flags, enums as, 443–447
- FlagsAttribute
 - enums, 446–447
 - overview, 795–797
- Flatten() method, 860
- Flattening sequences in query expressions, 722–723
- float type
 - overview, 47–48
 - special characteristics, 128–129
- Floating-point types
 - equality issues, 129–132
 - overview, 47–49
 - special characteristics, 128–129
- Flow control
 - Boolean expressions, 149–155
 - code blocks, 144–147
 - description, 121
 - for loops, 170–174
 - foreach loops, 174–175
 - if statements, 142–144
 - introduction, 138–141
 - jump statements, 180–185
 - preprocessor directives, 186–193
 - scope, 147–149
 - switch statements, 176–179
 - task continuation, 829
 - threads, 816
 - while and do/while loops, 168–170
- for loops
 - overview, 140, 170–174
 - parallel iterations, 895–898
- For() method, 897–898
- Forcing resource cleanup, 503–507
- foreach loops
 - arrays, 657–658
 - async streams, 873–874
 - collections, 661–662
 - enumerator patterns, 754
 - IEnumerable<T>, 658–659, 662–663
 - overview, 140, 174–175
 - parallel iterations, 898–899
- Form feeds escape sequence, 58
- Formal parameter lists, 205
- Format items, 29
- Format() method, 62–63, 65

- Format strings, 29–30
 - FormatException type, 244, 247
 - FormatMessage() method, 953
 - Formatting
 - hexadecimal format, 54
 - round-trip, 54–55
 - with string interpolation, 28
 - strings, 65–66
 - Forward accessing arrays, 100
 - Forward slashes (/)
 - command-line option, 783
 - compound assignment, 132
 - overriding, 466, 468
 - precedence, 124
 - XML comments, 32, 486–487
 - XML elements, 33
 - Fragile base class problem, 349–351
 - Framework Class Library (FCL), 979, 993–994
 - Framework Design Guidelines, 15
 - Frameworks
 - APIs, 40
 - .NET. *See* .NET frameworks
 - from keyword in query expressions, 707–709, 714, 722–723
 - FromCurrentSynchronizationContext()
 - method, 887
 - Full outer joins, 679
 - Fully qualified method names, 198
 - Func delegate, 592–594
 - Functions
 - external, 956–958
 - pointers to delegates, 960
- G**
- Garbage collection
 - CLI, 982–984
 - managed execution, 35
 - Microsoft .NET framework, 489–491
 - objects, 261
 - overview, 489
 - resource cleanup. *See* Resource cleanup
 - weak references, 491–493
 - GC class
 - finalization, 500
 - finalizers, 496
 - garbage collection timing, 490
 - resource cleanup, 506–508
 - General catch blocks, 248–249, 518–520
 - Generics
 - benefits, 543–544
 - CIL representation, 581–582
 - classes, 540–544
 - constraints. *See* Constraints
 - constructors, 547
 - covariance and contravariance, 573–580
 - default values, 547–549
 - defining, 542
 - delegates, 609
 - events, 647–648
 - examples without, 534–539
 - finalizers, 547
 - instantiating, 582–584
 - interfaces and structs, 544–546
 - internals, 580–584
 - introduction, 539–540
 - Java, 584–585
 - lazy loading, 509–510
 - linked lists, 749
 - methods, 568–573
 - nested, 552–553
 - overview, 533
 - reflection, 779–781
 - tuples, 550–552
 - type parameters, 308–309, 542–545, 549–552, 780–781
 - working with, 540–542
 - get keyword for properties, 278–279
 - GetAsyncEnumerator() method, 872–873
 - GetAwaiter() method, 870, 881
 - GetCommandLineArgs() method, 216
 - GetCustomAttributes() method, 788–790
 - GetEnumerator() method
 - duck typing, 662
 - iterators, 755, 757–761
 - purpose, 660
 - queryable extensions, 694–695
 - standard query operators, 663
 - GetFiles() method
 - directories, 321
 - inner joins, 682
 - GetGenericArguments() method, 780
 - GetHashCode() method
 - collections, 744–745
 - description, 364
 - overriding, 453–455, 463–464
 - value types, 427

- GetLastError() method, 953
 - GetLength() method, 114
 - GetMethods() method, 771
 - GetProperties() method, 771–772
 - GetResult() method, 881
 - getters
 - access modifiers, 289–291
 - properties, 67, 278–279
 - GetType() method
 - description, 364
 - null values, 87
 - objects, 462
 - reflection, 771–773
 - GetValue() method, 777
 - GhostDoc tool, 489
 - Global methods, 204
 - Global variables and functions, C# vs.
 - C++ and Visual Basic, 311
 - "goes to" operator, 208
 - goto statements
 - lambda expressions, 606
 - overview, 141
 - switch statements, 184–185
 - Graphs for expression trees, 618–619
 - Greater than signs (>)
 - generics, 540
 - overriding, 464, 466
 - pointers, 969
 - referent member access, 970
 - relational operators, 150
 - shift operators, 163–164
 - XML, 33
 - group clauses in query expressions, 707, 720–722
 - group by clause, 719
 - GroupBy() method, 685–686
 - Grouping query expressions, 718–721
 - GroupJoin() method
 - inner joins, 686–688
 - outer joins, 688–691
- H**
- Handle() method
 - asynchronous requests, 860
 - task exceptions, 839
 - Handlers for process exits, 506
 - Hardcoding values, 50
 - HasFlags() method, 444
 - Hash codes and tables
 - collection equality, 744–745
 - dictionaries, 741
 - GetHashCode(), 453–455
 - Hash symbols (#) for preprocessor
 - directives, 186
 - Hat operator (^) for arrays, 107–108
 - Header files, 208
 - Heap
 - boxing, 428
 - new operator, 262
 - reference types, 83, 420–421
 - HelloWorld.dll file, 10
 - HelloWorld program
 - assemblies, 10–11
 - CIL output, 37–39
 - compiling, 10–11
 - debugging, 8–9
 - description, 2–3
 - dotnet CLI for, 4–6
 - preparing, 3–4
 - project files, 9–10
 - stand-alone, 11–12
 - Visual Studio 2019 for, 6–8
 - Hexadecimal numbers
 - formatting numbers as, 54
 - notation, 52–53
 - Unicode characters, 58
 - HideScheduler task continuation option, 833
 - Hierarchy of classes, 257–258
 - High-latency operations
 - async/await syntax, 862–867
 - asynchronous example, 856–858
 - synchronous example, 854–856
 - WPF example, 889–891
 - Hill climbing, 899
 - Hold and wait conditions in deadlocks, 930
 - Hooking up publishers and subscribers, 628–629
 - Horizontal tabs escape sequence, 58
 - Hot tasks, 825
 - Hungarian notation, 16
 - Hyper-Threading, 815
 - Hyphens (-). *See* Minus signs (-)
- I**
- I/O-bound operations
 - latency, 814
 - performance considerations, 818–820
 - IAsyncDisposable interface, 874–875

- IAsyncEnumerable<T> interface
 - async returns, 880–881
 - asynchronous streams, 872–874
 - LINQ with, 875–877
- IAsyncEnumerator<string> interface, 877, 881
- ICollection<T> interface
 - collection initializers, 654–655
 - element counting, 671–672
 - members, 730–731
- IComparable<string> interface, 734
- IComparable<T> interface, 554–557, 572
- IComparer<T> interface, 734–735
- Id property for asynchronous tasks, 828
- IDE (integrated development environment)
 - debugging support, 8
 - Visual Studio 2019, 6
- Identifiers
 - guidelines, 16
 - keywords as, 17
 - overview, 15–16
- Identity of objects, 456–459
- IDictionary<TKey, TValue> interface, 728–730
- IDisposable interface
 - resource cleanup, 497–501, 661–662
 - tasks, 849
- IDynamicMetaObjectProvider interface, 808
- IEnumerable<T> interface
 - collections, 657–663, 730–731
 - delegates, 620–621
 - element counting, 671–672
 - filters, 667
 - foreach loops, 658–663, 754
 - iterators, 755, 760
 - PLINQ queries, 911
 - projections, 669
 - query expressions, 708–709, 714
 - standard query operators, 663
 - task-based asynchronous pattern, 877
 - yield statement, 767
- IEnumerator<T> interface
 - foreach loops, 658–662
 - iterators, 755–756, 764
 - yield statement, 767
- IEqualityComparer<T> interface, 743–744
- IEquatable<T> interface, 463
- #if directive, 186, 188
- if statements
 - Boolean expressions, 149
 - code blocks, 144–147
 - continue statement replacement, 183–184
 - nested, 142–144
 - overview, 139
 - switch statements for, 178
 - working with, 142
- IFileCompression interface, 379
- IGrouping type, 719, 721
- ILDASM (IL Disassembler), 36–37
- IList<T> interface, 728–730
- IList<TKey> interface, 746
- IList<TValue> interface, 746
- Immutability
 - anonymous types, 700–701
 - delegates, 598
 - strings, 26, 67–69, 968–970
 - value types, 424
- Immutable library, 939
- Implementing class conversions with interfaces, 390
- Implicit conversions
 - base and derived types, 339
 - guidelines, 471–472
 - types, 72, 75–76
- Implicit deterministic resource cleanup, 262
- Implicit member implementation, 386–390
- implicit operator for types, 75–76
- Implicitly typed local variables, 89–90
- import directive, 210
- in parameters, 223
- Including code, preprocessor directives for, 188
- Increment() method, 137, 927–928
- Increment operators
 - description, 133
 - prefix and postfix, 134–136
 - thread-safe, 137
- Indenting code
 - if statements, 144, 146
 - readability, 21–22
- Index operator
 - lists, 731–733
 - variable parameters, 752

- Index from end operator, 99, 107, 108, 110, 112, 118
- Index type, 112
- IndexerNameAttribute, 751
- Indexers, 750–752
- Indexes
 - arrays, 99, 107–108
 - dictionaries, 740–741
 - list searches, 736
 - sorted collections, 746
- IndexOf() method, 736
- IndexOutOfRangeException type, 247
- Indirection operators for pointers, 968–969
- Inequality
 - floating-point types, 129–132
 - with null, 156–157
 - operators, 150–151
- Inextensible classes, 320
- Inferences for generic methods, 569–571
- Infinite recursion errors, 231
- Infinity value, 131
- Infix notation for binary operators, 123
- Information hiding in encapsulation, 274–276
- Inheritance
 - abstract classes, 357–363
 - base class overriding. *See* Base class overriding
 - casting in, 340
 - constraints, 565–567
 - definitions, 335–336
 - derivation. *See* Derivation
 - exception classes, 246–248
 - interfaces, 390–393
 - is operator, 365–371
 - overview, 257–259, 335
 - System.Object class in, 363–365
 - value types, 427
- Initial section in for loops, 172–173
- Initialization, lazy, 508–510
- Initializers
 - collections, 297–298, 654–657
 - constructors, 300–302
 - object, 296–297
- Initializing
 - anonymous type arrays, 702–703
 - arrays, 104–106, 108
 - attributes, 789–793
 - dictionaries, 656
 - fields, 548
 - loop counters values, 170
 - static members, 312–313, 317–318
 - structs, 424–426
- Inner classes, 328
- Inner joins
 - description, 679
 - one-to-many relationships, 686–688
 - performing, 682–686
- InnerException property, 522, 525, 527
- InnerExceptions property
 - aggregate exceptions, 838, 908
 - asynchronous requests, 860
 - error handling, 640
 - parallel loops, 901
 - tasks, 838
- INotifyCompletion interface, 881
- INotifyPropertyChanged interface, 782
- Input, console, 26–27
- Insert() method, 69
- Inserting dictionary items, 740–741
- Instance fields
 - accessing, 264–265
 - declaring, 263–264
 - description, 262–263
 - initializing, 312–313
- Instance members
 - arrays, 114–115
 - strings, 62
 - working with, 265–266
- Instantiation
 - arrays, 102–106
 - classes, 260–261
 - defined, 19
 - delegates, 595–597
 - generics, 582–584
- Integers and int type
 - characteristics, 46–47
 - characters, 127
 - conversions, 72
 - default, 50
 - description, 23
 - enums, 439
 - overflow, 528–530
 - vs. strings, 240–241
- Integrated development environment (IDE)
 - debugging support, 8
 - Visual Studio 2019, 6
- IntelliSense feature, 708

Interfaces

- vs. abstract classes, 413–414
 - vs. attributes, 415
 - collections, 728–731
 - constraints, 554–557
 - converting between implementing classes, 390
 - default members, 411–413
 - deriving, 397–398
 - diagramming, 395–396
 - duplicating, 545–546
 - extension methods, 394–395, 411–413
 - generics, 544–546
 - implementation, 377–378, 384–390
 - inheritance, 390–393
 - overview, 377
 - polymorphism, 380–384
 - purpose, 378–380
 - refactoring features, 401–405
 - value types, 427
 - versioning. *See* Versioning
- Interlocked class
- increment and decrement operators, 137
 - working with, 926–928
- internal access modifier
- description, 481
 - interface refactoring, 402
 - on type declarations, 479–480
- Interoperability. *See* Platform interoperability
- InteropServices class
- SafeHandle, 955–956
 - unmanaged exceptions, 519
- Interpolation of strings, 28–30, 61–62
- Intersect() operator, 693
- into clause in query expressions, 721–722
- IntPtr class, 951, 963
- InvalidAddressException type, 523
- InvalidCastException type
- description, 247
 - generics, 543
 - using, 472
- InvalidOperationException type
- catch blocks, 516–517
 - collections, 663
 - description, 247
 - null values, 87
 - tasks, 838
 - wrapped exceptions, 527

Invocation

- base class constructors, 355–356
 - delegates, 629–632, 634–636
 - dynamic, 773–779
 - referenced packages and projects, 477–479
 - using statement, 499
- Invoke() method
- delegates, 630
 - dynamic invocation, 777
 - events, 161
- IOException type, 520, 527
- IOrderedEnumerable<T> interface, 677
- IProducerConsumerCollection<T> interface, 939
- IQueryable<T> interface
- delegates, 620–621
 - query expressions, 708
 - queryable extensions, 694–695
- "is a" relationships
- abstract members, 361
 - classes, 388
 - derivation forms in, 339
- "is a kind of" relationships, 257–258, 336
- is operator
- pattern matching example, 156–157
 - pattern matching overview, 365–371
 - positional pattern matching, 369
 - properties, 369–370
 - recursive pattern matching, 370–371
 - tuples, 368
 - type, var, and const pattern matching, 366–368
 - types with, 365–366
- is { } property pattern, 156–157
- IsCancellationRequested property, 846
- IsCompleted property
- multithreading, 828
 - parallel loops, 905
- IsDefined() method, 445
- ISerializable interface, 526
- IsGenericType property, 780
- IsInvalid member, 956
- Iterations
- loops, 170
 - parallel. *See* Parallel iterations
- Iterators
- compiling, 764–766
 - defining, 754
 - escaping, 763–764

- multiple, 766–767
- origin, 753–754
- recursive, 762–763
- states, 757–759
- syntax, 755
- yielding values from, 755–762

J

Jagged arrays

- defining, 106
- element assignment, 108–109
- length, 110

Java language vs. C#

- arrays, 101–102
- exception specifiers, 518
- filenames, 10
- generics, 584–585
- implicit overriding, 347
- import directive, 210
- inner classes, 328
- Main(), 19
- syntax similarities, 2–3
- virtual methods, 346

JIT (just-in-time) compiler and jitting

- CIL, 980
- description, 34
- thread synchronization, 920

Join() method

- inner joins, 682–686
- threads, 914

Join operations

- inner joins, 682–686
- outer joins, 688–691
- overview, 678–682

Jump statements

- break, 180–181
- continue, 182–184
- goto, 184–185
- lambda expressions, 606

Just-in-time (JIT) compiler and jitting

- CIL, 980
- description, 34
- thread synchronization, 920

K

KeyNotFoundException type, 741

Keys

- dictionaries, 728–730, 740–741
- equality, 743–744
- sorted collections, 746

Keys property, 746

KeyValuePair<TKey, TValue> class, 741

Keywords

- contextual, 766
- as identifiers, 17
- list of, 14
- overview, 13–15
- types, 47

L

Labels for switch statements, 177, 185

Lambda calculus, 608

Lambda expressions

- asynchronous, 881–887
- as data, 616–618
- deferred execution, 674
- expression trees, 616–623
- filters, 715
- internals, 610–611
- lazy loading, 509–510
- notes and examples, 605–606
- outer variables, 611–616
- overview, 587–588, 600–601, 604
- predicates, 593
- purpose, 608
- statement, 601–604

LambdaExpression expression trees, 623

Language Integrated Query (LINQ)

- anonymous types, 695–703
- collections, 653–654
- deferred execution, 672–676
- delegates, 620
- extension methods, 395
- with IEnumerable, 875–877
- parallel queries, 670–671, 905–911
- query expressions. *See* Query expressions

Language interoperability, 35

Last in, first out (LIFO) collections, 747

LastIndexOf() method, 736

Late binding with metadata, 992

Latency

- async/await syntax, 862–867
- asynchronous example, 856–858
- description, 814
- multithreading for, 816–818
- synchronous example, 854–856
- WPF example, 889–891

Lazy initialization, 508–510

- LazyCancellation task continuation
 - option, 834
- Leaf asynchronous task-returning
 - method, 883–887
- Left-associative operators, 125
- Left outer joins, 679
- Length
 - arrays, 109–110
 - strings, 66–67
- Length property, 66
- Less than signs (<)
 - generics, 540
 - overriding, 464, 466
 - pointers, 969
 - relational operators, 150
 - shift operators, 163–164
 - XML, 33
- let clause, 717–718
- Libraries
 - BCL. *See* Base Class Library (BCL)
 - classes, 472–475
 - creating, 473
 - dependencies, 10
 - referencing, 473–475
 - TPL. *See* Task Parallel Library (TPL)
- Lifetime of captured outer variables, 612
- LIFO (last in, first out) collections, 747
- Line-based statements in Visual Basic vs C#, 20
- #line directive, 187, 191
- Line feeds
 - escape sequence, 58–59
 - newlines, 66
- Line numbers, preprocessor directives
 - for, 191
- Linked lists, 749
- LinkedList<T> class, 749
- LinkedListNode<T> class, 749
- LINQ. *See* Language Integrated Query (LINQ)
- Liskov, Barbara, 753–754
- List<T> class
 - overview, 731–733
 - searches, 736–738
- Listeners for events, 627, 835–836
- Lists
 - collections, 728–733
 - linked, 749
 - searching, 736–738
 - sorted, 745–747
- Literal values
 - characters, 57
 - default, 50–51
 - overview, 49–50
 - strings, 59–61
- Load() method, 273
- Loading
 - with files, 270–273
 - lazy, 508–510
- Local variables
 - anonymous types, 695–701
 - assigning values to, 24–25
 - declaring, 22–23
 - implicitly typed, 89–90
 - scope, 147–149
 - thread synchronization, 917–918
 - types, 23
 - working with, 25
- Localized applications, Unicode
 - standard for, 56
- Locals, constant, 137–138
- lock keyword
 - thread synchronization, 920–925
 - value types, 432–434
- Locks
 - deadlocks, 821–823, 929–930
 - thread synchronization, 920–925, 931
 - value types, 432–434
- Logical operators
 - Boolean, 151–153
 - overriding, 468
 - working with, 164–167
- Lollipops, 395
- Long-running tasks, 848–849
- long type
 - characteristics, 46
 - conversions, 72
- LongRunning task continuation option
 - delays, 849
 - description, 832
 - TPL performance, 900
- Loop variables in lambda expressions, 614–616
- Loops
 - for, 170–174
 - foreach, 174–175
 - parallel iterations, 895–905
 - while and do/while, 168–170
- LowestBreakIteration property, 905

M

- Machine code, compilation to, 34–35, 979–981
- Main() method
 - async, 867
 - declaring, 18–19
 - description, 196–197
 - Java vs. C#, 19
 - multiple, 214–216
 - return values and parameters, 214–216
- __makeref keyword, 17
- Managed code, 34
- Managed execution, 34–35
- Manifests in CLI, 986–989
- ManualResetEvent class, 934–937
- ManualResetEventSlim class, 934–937
- Many-to-many relationships in join operations, 679–680
- Masks, logical operators for, 165
- Matching caller variables with parameters, 218
- Max() function, 693
- MaxDegreeOfParallelism property, 904
- MaybeNull attribute, 307, 309
- MaybeNullWhen attribute, 307
- Me keyword, 268
- Mechanism relationships, 388
- Members
 - abstract classes, 357–360
 - base class overriding, 353–355
 - class variables, 262
 - classes, 262
 - default interface, 398–400
 - dynamic invocation, 773–779, 781–782
 - explicit and implicit implementation, 386–390
 - static. *See* Static members
- MemberwiseClone() method, 364
- Memory
 - boxing, 428
 - finalizers, 495
 - generics, 543
 - multithreading complexity, 821
 - objects, 261
 - reference types, 82–84
- Metadata
 - attributes, 306, 783–784
 - CIL, 991–992
 - CLI, 35
 - reflection, 770–773
 - XML, 33
- Method groups for delegates, 596
- Method invocations, query expressions as, 724–725
- MethodCallExpression expression trees, 623
- MethodImplAttribute, 925
- MethodImplOptions class, 925
- MethodInfo class
 - delegates, 598
 - dynamic invocation, 777
 - reflection, 788
- Methods
 - anonymous, 600, 606–608, 610–611
 - arguments, 201
 - attributes, 799
 - call stacks, 216–217
 - called during construction, 348–349
 - casting in, 572–573
 - constraints, 571–572
 - constructors. *See* Constructors
 - declaring, 203–204
 - description, 18, 196–197
 - expression bodied, 208
 - extension, 321–322, 344, 394–395, 411–413
 - external, 948–949
 - generics, 568–573, 779–780
 - instance, 265–266
 - interface refactoring, 401
 - Main(), 214–216
 - names, 201
 - namespaces, 198–200, 209–211
 - overloading, 231–234
 - overview, 197–198
 - parameters. *See* Parameters
 - partial, 330–332
 - passing as arguments, 590–592, 596
 - publish-subscribe pattern, 640
 - recursion, 229–231
 - refactoring into, 204–205
 - resolution, 238–239
 - return values, 202, 206–208
 - scope, 201
 - vs. statements, 202–203
 - static members, 315–316
 - strings, 62–64
 - structs, 424
 - syntax, 18
 - this keyword, 269–270
 - type inferences, 569–571
 - types, 200–201

- using directive, 209–214
 - virtual, 346–349
 - Microsoft .NET Framework
 - assemblies, 40
 - CLI implementation, 975–977
 - description, 39
 - garbage collection, 489–491
 - versioning, 40–42
 - Microsoft Silverlight compiler, 976
 - Min() function, 693
 - Minus signs (-)
 - compound assignment, 132
 - decrement operator, 133
 - delegates, 632–634, 650
 - identifier names, 16
 - overriding, 466–469
 - precedence, 124
 - referent member access, 970
 - subtraction, 123
 - unary operator, 122–123
 - Modules
 - attributes, 784–786
 - CLI, 986–989
 - dependencies, 992
 - Monitor class
 - locks, 432
 - thread synchronization, 918–920, 923–924
 - Mono compiler, 976
 - Mono .NET frameworks, 39
 - Move() method, 321
 - MoveNext() method
 - duck typing, 662
 - iterators, 758–759, 764
 - MoveNextAsync() method, 874
 - MTAs (multithreaded apartments), 945
 - Multicast delegates
 - description, 625
 - internals, 636–637
 - publish-subscribe pattern. *See* Publish-subscribe pattern
 - MulticastDelegate class
 - constraints, 558–559
 - internals, 597–598, 636–637
 - Multidimensional arrays, 102–106, 108
 - Multiple constraints, 561–562
 - Multiple inheritance
 - C++, 345
 - interfaces, 393
 - Multiple interfaces for single classes, 545–546
 - Multiple Iterators, 766–767
 - Multiple Main() methods, 214–216
 - Multiple .NET frameworks, 39–43
 - Multiple searches in lists, 737–738
 - Multiple selection for query expressions, 722–723
 - Multiple threads synchronization. *See* Thread synchronization
 - Multiple type parameters in generics, 549–551
 - Multiplication
 - binary operator, 123
 - compound assignment, 132
 - shift operator, 164
 - Multithreaded apartments (MTAs), 945
 - Multithreading
 - asynchronous tasks. *See* Asynchronous tasks
 - atomic operations, 819–820
 - canceling tasks, 843–849
 - deadlocks, 821–822
 - description, 816
 - disposability of tasks, 849
 - issues, 819–822
 - jargon, 815–816
 - long-running tasks, 848–849
 - memory model complexity, 821
 - overview, 813–815
 - performance considerations, 818–819
 - purpose, 816–818
 - race conditions, 820–821
 - Threading class, 850–851
 - Mutable value types, 434
 - Mutex class, 932–933
 - MutexSecurity class, 932–933
 - Mutual exclusion in deadlocks, 930
- ## N
- \n
 - escape sequence, 58–59
 - new lines, 66
 - Name/value pairs for dictionaries, 656, 738, 741
 - Named arguments, 237–238
 - Named parameters
 - attributes, 794–795
 - changing names of, 237
 - nameof operator
 - arguments, 513
 - properties, 285
 - working with, 781–782

Names

- ambiguity, 267–268
- class definitions, 17
- collisions, 198
- constant values, 137
- constructor parameters, 300
- constructors, 294
- enums, 439
- fields, 281–282
- identifiers, 15–16
- indexers, 751
- local variables, 24
- methods, 197–198, 201
- scope, 147–149
- tuples, 95
- type parameters, 544
- variables, 418

Namespaces

- aliasing, 213–214
- defining, 481–484
- methods, 198–200
- nesting, 483
- using directive, 64–65, 209–211

NaN (Not a Number) value, 131

Native code, 34

NDoc tool, 489

Negation operator (!), 153

Negative infinity value, 131

Negative numbers notation, 163

Negative zero value, 132

Nested items

- classes, 326–328
- delegates, 594–595
- generic types, 552–553
- if statements, 142–144
- iterators, 762–763
- namespaces, 210, 483
- using directive, 211–212

.NET Compact Framework, 976

.NET Core, 975, 977–978

.NET frameworks

- description, 993
- garbage collection, 489–491, 983–984
- multiple, 39–43
- overview, 3
- standard, 42–43
- versioning, 40–42

.NET Micro Framework, 976

Net namespace, 483

.NET Native feature, 993

.NET Standard version, 978–979

new modifier in base class overriding,
349–353

new operator

- arrays, 100, 103
- constructors, 294–295
- objects, 260–261
- projections, 668–669
- value types, 426–427

Newlines

- escape sequence, 58–59
- preprocessor directives, 186
- variations, 66

NGEN tool, 980

No preemption condition in deadlocks, 930

NodeType property expression trees, 623

Non-nullable reference type properties,
303–306

None task continuation option, 831

Normal continuation, 829

Normal form for parameters, 227

Normalized data, 682

Not a Number (NaN) value, 131

NOT operators

- inequality, 150–151
- logical, 153

Notifications for thread synchronization,
928–929

NotImplementedException type, 248

NotNull attribute, 307

nonnull constraints, 560

NotNullIfNotNull attribute, 307–308

NotNullWhen attribute, 307–308

NotOnCanceled task continuation option,
833NotOnFaulted task continuation option,
832NotOnRanToCompletion task
continuation option, 832nowarn preprocessor directive option,
190–191

NuGet packages

- invoking, 477–479
- references, 475–477
- referencing, 473

Null character escape sequence, 57

Null-coalescing operator, 157–160

Null-conditional operator

- dereferencing operations, 513
- null checks, 86, 630
- thread synchronization, 928
- working with, 158–161

- Null-forgiving operator (!), 160–161, 634
 - null modifier, 308–309
 - null references, dereferencing, 85
 - null values
 - checking for, 85–86
 - collections, 753
 - dereferencing, 87
 - pointers, 964–965
 - programming with, 155–157
 - publish-subscribe pattern, 630–632
 - strings, 69–70
 - types allowing, 84–85
 - Nullable attributes, 306–308
 - #nullable directive, 88, 187, 193
 - nullable modifier
 - reference types, 71
 - strings, 70
 - type declarations, 84–85
 - Nullable reference types
 - default values, 548
 - nullable modifier, 84
 - overview, 70–71, 87–89
 - preprocessor directives, 193
 - Nullable values
 - array initial values, 104
 - default values, 548
 - description, 86
 - example, 538–539
 - struct constraints, 561
 - Nullable<T> type, 87, 308, 561
 - NullReferenceException type
 - delegates, 644
 - dereferencing null values, 85, 87, 513
 - description, 248
 - reference type nullability, 88
 - thread synchronization, 928
 - Numeric types
 - decimal, 49
 - floating-point, 47–49
 - hexadecimal notation, 52–54
 - integers, 46–47
 - literal values, 49–50
 - round-trip formatting, 54–55
- O**
- Object class
 - derivation from, 363–365
 - dynamic objects, 805–806
 - Object-oriented programming
 - encapsulation, 257
 - inheritance, 257–259
 - overview, 256–257
 - polymorphism, 259
 - Objects. *See also* Classes
 - associated data, 314
 - CTS, 990
 - defined, 260
 - dynamic programming, 801–802
 - expression trees, 618–619
 - filtering, 666–667
 - identity vs. equality, 456–460
 - initializers, 296–297
 - member overriding, 451–464
 - resurrecting, 507–508
 - ObsoleteAttribute, 797, 799–800
 - OfType<T>() operator, 693
 - One-to-many relationships
 - example, 680–681
 - inner joins, 686–688
 - OnlyOnCanceled task continuation
 - option, 832
 - OnlyOnFaulted task continuation option, 833
 - OnlyOnRanToCompletion task
 - continuation option, 833
 - Operands
 - description, 122
 - evaluation, 126
 - Operational polymorphism, 232
 - OperationCanceledException type, 908, 911
 - operator keyword, 466
 - Operators
 - assignment, 24–25
 - associativity, 124–125
 - binary, 123–124
 - bitwise, 162–167, 181–182
 - characters in arithmetic operations, 127–128
 - compound assignment, 132, 468
 - concatenation, 126–127
 - conditional, 154–155
 - constant expressions and locals, 137–138
 - constraints, 567–568
 - decrement, 133–137
 - delegates, 633–634
 - description, 121
 - evaluation, 125
 - floating-point equality, 129–132
 - increment, 133
 - logical, 151–153, 164–167
 - with null, 155–157

- null-coalescing, 157–158
 - null-conditional, 158–161
 - null-forgiving, 160–161
 - overriding, 464–472
 - overview, 122
 - precedence, 124–125, 194
 - relational and equality, 150–151
 - shift, 163–164
 - special floating-point characteristics, 128–129
 - thread-safe, 137
 - unary, 122–123
 - Optional parameters, 234–238
 - OR operations
 - bitwise operators, 164–165
 - constraints, 568
 - enums, 444–445
 - logical operators, 151–152
 - orderby clause, 716–717
 - OrderBy() method, 676–678
 - OrderByDescending() method, 678
 - Ordering collections, 735–736
 - out parameters
 - conversions, 78
 - properties, 291
 - out type parameter modifier, 575–577
 - Outer joins, 679, 688–691
 - Outer variables in lambda expressions, 611–616
 - OutOfMemoryException type, 514, 520
 - Output, console, 28–33
 - Output parameters, 220–223
 - Overflow, buffer
 - C# vs. C++, 109–110
 - managed execution, 986
 - type safety, 35, 984
 - unsafe code, 962
 - Overflow of integers, 528–530
 - OverflowException type, 73, 529
 - Overloading
 - constructors, 298–300
 - methods, 231–234
 - override keyword, 349–351
 - Overriding
 - assignment operators, 468
 - Base class. *See* Base class overriding
 - binary operators, 466–467
 - comparison operators, 464–465
 - conversion operators, 470–472
 - Equals(), 455–464
 - GetHashCode(), 453–455, 463–464
 - logical operators, 468
 - methods, 349–351
 - object members, 451–464
 - operators, 464–472
 - ToString(), 452–453
 - unary operators, 468–470
- P**
- P/Invoke. *See* Platform Invoke (P/Invoke)
 - Packages, invoking, 477–479
 - Pair<T> type
 - iterators, 760, 764
 - struct vs. class, 763
 - Parallel class, 897–898
 - Parallel iterations
 - loops, 895–905
 - overview, 895
 - performance, 899–900
 - PLINQ queries, 905–911
 - Parallel LINQ (PLINQ), 814
 - canceling queries, 908–911
 - parallel queries, 905–911
 - running queries, 670–671, 905–908
 - Parallel loops
 - breaking, 904–905
 - canceling, 901–903
 - exception handling, 900–901
 - for, 895–898
 - foreach, 898–899
 - options, 903–904
 - performance, 899–900
 - Parallel programming, 818
 - ParallelEnumerable class, 670, 906–907
 - ParallelLoopResult class, 905
 - ParallelLoopState class, 905
 - ParallelOptions class, 904
 - ParallelQuery<T> class, 911
 - ParameterExpression expression trees, 623
 - Parameterized generic types, 539–540
 - Parameters
 - anonymous methods, 607–608
 - arrays, 226–229, 552
 - catch blocks, 516
 - constraints. *See* Constraints
 - constructors, 294, 299–300
 - declaring, 205
 - events, 645
 - extension methods, 322

- finalizers, 495
- generics, 542, 544–545, 549–551
- in, 223
- indexers, 752
- lambda expressions, 606
- Main(), 214–216
- matching caller variables with, 218
- method resolution, 238–239
- methods, 197, 201
- optional, 234–238
- out, 220–223
- reference types, 218–220
- return by reference, 223–225
- statement lambdas, 602–604
- types, 949–951
- value, 217–220
- Parent types, 258, 336
- Parentheses ()
 - casts, 72, 339
 - collection initializers, 655
 - declaration lists, 205
 - lambda expressions, 603
 - logical operators, 152
 - operator precedence, 125
 - query expressions, 723
 - tuples, 92
- Parse() method
 - conversions, 76, 79
 - enum, 442
 - FlagsAttribute, 796–797
 - metadata, 772–773
- Parsing types, 77–79
- Partial classes
 - defining, 329–330
 - methods, 330–332
- Partial interface refactoring, 405
- Partial methods
 - description, 208
 - interface refactoring, 405
- PascalCase
 - description, 15–16
 - field names, 281
 - interfaces, 379
 - namespaces, 482
 - properties, 282
 - tuples, 95
 - type names, 17
- Pass-by-reference, 640
- Passed by reference variables, 220
- Passed by value arguments, 217
- Passing
 - anonymous methods, 607
 - arrays, 226
 - command-line arguments, 215
 - delegates, 509, 597–598, 604
 - instances, 419
 - method return values, 202
 - methods as arguments, 590–592, 596
 - null values, 85
 - out parameters, 220
 - read-only references, 223
 - this keyword, 270, 426
 - values to methods, 82, 217
 - variables by reference, 219
 - variables to methods, 197
- Pattern matching
 - is operator, 365–371
 - with null, 156
 - polymorphism, 373
 - positional, 369
 - properties, 369–370
 - recursive, 370–371
 - switch statements, 179, 371–372
 - tuples, 368
 - types, var, and const, 366–368
- Peek() method, 747
- Percent signs (%)
 - compound assignment, 132
 - overriding, 466, 468
 - precedence, 124
 - remainder operation, 123
- Performance
 - CLI, 985–986
 - hash codes, 454
 - multithreading, 818–819
 - parallel iterations, 899–900
- Periods (.)
 - fully qualified method names, 198
 - nested namespaces, 210, 483
 - null-conditional operator, 158
- Pi calculations, 895–898
- Platform interoperability
 - delegates, 971–972
 - overview, 947–948
 - P/Invoke, 948–960
 - pointers and addresses, 960–970
- Platform Invoke (P/Invoke)
 - description, 948
 - error handling, 953–955
 - external functions, 948–949, 956–958

- function pointers, 960
- parameter data types, 949–951
- references, 951–952
- SafeHandle, 955–956
- sequential layout, 952–953
- wrappers, 959
- Platform portability
 - CLI, 985
 - managed execution, 35
 - preprocessor directives for, 188
- PLINQ. *See* Parallel LINQ (PLINQ)
- Plus signs (+)
 - addition, 123
 - characters, 127
 - compound assignment, 132
 - delegates, 633–634, 650
 - increment operator, 133
 - overriding, 464, 466–469
 - precedence, 124
 - strings, 126–127
 - unary operator, 122–123
- Pointers
 - assigning, 964–965
 - declaring, 963–964
 - dereferencing, 968–970
 - fixing data, 965–967
 - function, 960
 - P/Invoke, 951–952
 - vs. ref, 951
 - referent member access, 970
 - stacks, 967–968
 - unsafe code, 961–963
- Polling tasks, 826–828
- Polyfill code, 526
- Polymorphism
 - abstract classes, 361–363
 - interfaces, 380–384
 - operational, 232
 - overview, 259
 - pattern matching, 373
 - with protected interface members, 406–411
- Pools
 - temporary, 419
 - threads, 816, 824, 850–851
- Pop() method, 534–537, 747–748
- Positional pattern matching, 369
- Positive infinity value, 131
- Positive zero value, 132
- Postfix operators, 134–136
- #pragma directive, 187–188, 190–191
- Precedence of operators, 124–125, 194
- Precision of floating-point types, 47–48, 128–129
- Predefined attributes, 797
- Predefined types, 45
- Predicates
 - delegates, 621, 666–667
 - filters, 715
 - lambdas, 593
- PreferFairness task continuation
 - option, 832
- Prefix operators, 134–136
- Prefixes for hexadecimal notation, 53
- Preprocessor directives
 - excluding and including code, 188
 - line numbers, 191
 - list of, 186–188
 - nullable references, 193
 - purpose, 186
 - symbols, 189
 - visual code editors, 191–193
- private access modifier
 - description, 481
 - inheritance, 341–342
 - interface refactoring, 402
 - nested classes, 328
- private internal access modifier, 481
- Private members for information hiding, 274–276
- private protected access modifier, 402–403
- Procedural programming, 195
- Process class
 - multithreading, 815
 - notifications, 885
- Processes, description, 815
- ProcessExit events, 503–507
- Processor-bound latency, 814
- Projections
 - anonymous types, 698–699
 - collections, 668–669
 - query expressions, 706, 708–711
- Projects
 - creating, 9–10
 - invoking, 477–479
 - referencing, 473–475
- Properties
 - attributes, 783–787
 - automatically implemented. *See* Automatically implemented properties

- calculated, 287–289
 - declaring, 277–279
 - getters and setters, 289–291
 - guidelines, 281–283
 - interface refactoring, 401
 - internals, 291–293
 - lazy loading, 508
 - nameof operator, 285
 - overriding, 346–347
 - overview, 276–277
 - pattern matching, 369–370
 - read-only and write-only, 286–287
 - retrieving, 771–772
 - static, 318–319
 - strings, 66–70
 - structs, 425–426
 - unallowed parameter values, 291
 - with validation, 283–285
 - PropertyChanged event, 782
 - PropertyInfo class
 - attributes, 788, 790
 - dynamic invocation, 777
 - protected access modifier
 - description, 481
 - inheritance, 342–344
 - interface refactoring, 402
 - protected interface members,
 - encapsulation and polymorphism with, 406–411
 - protected internal access modifier, 480–481
 - public access modifier
 - description, 481
 - interface refactoring, 402
 - on type declarations, 479–480
 - Public constants, 324
 - publish command, 11–12
 - Publish-subscribe pattern
 - delegate invocation, 629–632, 634–636
 - delegate operators, 633–634
 - encapsulation, 641
 - error handling, 637–640
 - hooking up publishers and subscribers, 628–629
 - method returns and pass-by-reference, 640
 - multicast delegate internals, 636–637
 - null checks, 630–632
 - publishers, 627–628
 - subscriber methods, 626–627
 - Pulse() method, 920
 - Pure virtual functions, 361
 - Push() method, 534–537, 747
- Q**
- Quantums, 817
 - Queries, parallel, 670–671, 905–911
 - Query continuation clauses, 721–722
 - Query expressions
 - deferred execution, 711–715
 - distinct members, 723–724
 - filters, 715–716
 - grouping, 718–721
 - introduction, 706–708
 - as method invocations, 724–725
 - overview, 705–706
 - projections, 708–711
 - query continuation clauses, 721–722
 - sequences, 722–723
 - sorting, 716–718
 - tuples, 710–711
 - Queryable class
 - delegates, 620
 - queryable extensions, 694–695
 - Question marks (?)
 - command-line option, 783
 - conditional operators, 154–155
 - null-coalescing operator, 157–158
 - null-conditional operators, 158–160
 - nullable modifier, 70, 85
 - Queue<T> class, 658, 748
 - Queues
 - collections, 748
 - finalization, 298, 500–501
 - thread synchronization, 920, 939
- R**
- \r
 - escape sequence, 58
 - newlines, 66
 - Race conditions
 - increment and decrement operators, 137
 - multithreading, 820–821
 - thread synchronization, 916
 - Range type, 112
 - Range variables in query expressions, 707–708, 721
 - Ranges for arrays, 101, 111–112

- Rank members, 114–115
- Ranks of arrays, 101–102
- Read() method
 - console input, 27, 845
 - interlocks, 927
- Read-only automatically implemented
 - property
 - declaring, 325–326
 - reference-type, 305–306
- Read-only pass by reference parameters, 223
- Read-only properties
 - defining, 286–287
 - strings, 67
- Read-only structs, 423–424
- Read/write non-nullable reference type
 - properties, 304–305
- Readability
 - vs. comments, 32–33
 - delegates, 594
 - digit separators, 52
 - enums, 437
 - generics, 543
 - if statements, 183
 - importance, 203
 - indentation, 146
 - methods, 18
 - parentheses, 125
 - refactoring for, 204–205
 - switch statements, 179
 - whitespace, 21–22
- ReadAllTextAsync() method, 874
- ReadAsync() method, 864
- ReadKey() method, 27
- ReadLine() method
 - console input, 26–27
 - invoking, 197–198
- readonly modifier, 324–326
- Recursion
 - infinite, 231
 - iterators, 762–763
 - methods, 229–231
 - pattern matching, 370–371
- Redimensioning arrays, 114
- Reentrant locks, 930
- ref parameters for properties, 291
- Refactoring
 - classes, 336–337
 - interface features, 401–405
 - into methods, 204–205
- ReferenceEquals() method
 - description, 364
 - with null, 156
 - objects, 456–458
- References and reference types
 - array initial values, 104
 - assemblies, 472–479
 - covariance, 576
 - default values, 548
 - garbage collection, 491–493
 - generic instantiation, 583–584
 - identity vs. equality, 456–459
 - invoking, 477–479
 - libraries, 473–475
 - new operator, 426
 - non-nullable properties, 303–306
 - NuGet packages, 475–477
 - overview, 81–84
 - parameters, 218–220
 - vs. pointers, 951–952
 - vs. value, 419–421
- Referent types
 - member access, 970
 - pointers, 963–964
- Reflection
 - description, 769
 - dynamic programming, 801–802
 - generics, 779–781
 - member invocation, 773–779
 - metadata, 770–773, 992
 - overview, 770–771
- Reflection class
 - delegates, 598
 - metadata, 772
- __ref type keyword, 17
- __ref value keyword, 17
- #region directive, 187, 191–193
- Register() method, 846
- Registration
 - assemblies, 989
 - finalization activities, 503–504
 - token cancellation, 846
- RegularExpressions class, 199
- Relational operators, 150–151
- ReleaseHandle() method, 956
- Remainder operations
 - binary operator, 123
 - compound assignment, 132
- Remove() method
 - delegates, 558, 634, 650–651

- dictionary elements, 741
 - lists, 733
 - strings, 69
 - `RemoveAt()` method, 733
 - `Replace()` method, 64, 69
 - Representation errors for floating-point types, 128–129
 - `ReRegisterFinalize()` method, 507–508
 - Reserved keywords, 15, 17
 - Reset events for thread synchronization, 934–937
 - `Reset()` method, 659
 - `Resize()` method, 114
 - Resolution of methods, 238–239
 - Resource cleanup
 - description, 493
 - deterministic finalization, 262, 496–500
 - exception propagating from
 - constructors, 507
 - finalizers, 493–496
 - forcing, 503–507
 - garbage collection, finalization, and `IDisposable`, 500–501
 - guidelines, 501–502
 - resurrecting objects, 507–508
 - Results, operator, 122
 - Resurrecting objects, 507–508
 - Rethrowing exceptions, 517–518, 522–523, 527–530
 - Return attributes, 785–786
 - Return by reference parameters, 223–225
 - return statement, 206–207
 - Return values
 - async, 864, 868–870
 - asynchronous void methods, 877–881
 - declaring, 206–208
 - `Main()`, 214–216
 - methods, 197–198, 202
 - `ReadLine()`, 26–27
 - Reverse accessing arrays, 100
 - `Reverse()` method
 - arrays, 112
 - collections, 693
 - strings, 116–117
 - Right-associative operators, 125
 - Right outer joins, 679
 - Root references in garbage collection, 490
 - Round-trip formatting, 54–55
 - Rounding floating-point types, 48, 128–129
 - `Run()` method
 - asynchronous tasks, 825–826
 - preference for, 847–848, 914
 - `RunContinuationsAsynchronously` task
 - continuation option, 834
 - Running source code, 3–4
 - `RunProcessAsync()` method, 883
 - Runtime
 - CLI, 982–984
 - defined, 35
 - description, 35
 - purpose, 974
 - Runtime callable wrappers (RCWs), 945
 - `Runtime.InteropServices`
 - `COMException` type, 514
 - `RuntimeBinderException` type, 802, 804, 810
- S**
- `SafeHandle` class, 955–956
 - sbyte type, 46
 - Schedulers for tasks, 824, 887–889
 - Scope
 - captured outer variables, 612
 - code blocks, 147–149
 - methods, 201
 - SDK (software development kit), 3
 - sealed access modifier
 - base class overriding, 353–354
 - interface refactoring, 404
 - sealed classes, 345
 - Sealed classes, 345
 - Searching
 - dictionaries, 741
 - lists, 736–738
 - Security
 - buffer overflows, 110
 - code access, 35
 - exposing holes in, 962–963
 - hash codes, 454
 - obfuscation, 36
 - `SEHException` type, 514, 519
 - select clauses in query expressions, 707–708
 - `Select()` method
 - anonymous types, 698–699
 - LINQ queries, 906
 - outer joins, 691
 - projections, 668–669
 - Selecting into anonymous types, 698–699

- SelectMany() method
 - outer joins, 688–691
 - query expressions, 722–723
- Semantic relationships, 388
- Semaphore class
 - reset events, 937–938
 - thread synchronization, 934
- SemaphoreSlim class, 937–938
- Semicolons (;)
 - ending statements, 20
 - for loops, 171–172
 - preprocessor directives, 189
- SequenceEquals() operator, 693
- Sequences in query expressions, 722–723
- Sequential invocation, 634–636
- Sequential layout, 952–953
- Serializable exceptions, 526
- SerializableAttribute, 526
- SerializationInfo type, 526
- set keyword for properties, 278–279
- Set() method for thread
 - synchronization, 934
- SetResult() method, 885
- setters
 - access modifiers, 289–291
 - properties, 67, 278–279
- SetValue() method, 777
- Shared collection states, 660
- Shared Source compiler, 976
- Shift operators, 163–164
- Short-circuiting operators
 - asynchronous methods, 868
 - conditional, 154
 - logical, 152
 - null-coalescing, 157
 - null-conditional expressions, 159
- short type, 46
- SignalAndWait() method, 933
- Signatures
 - constructors, 299–300
 - deconstructors, 311
 - interfaces, 396
 - methods, 147, 231–234
- Signed numbers, 162–163
- Significant digits in floating-point types, 48
- Silverlight compiler, 976
- Simultaneous multithreading, 815
- Single inheritance, 345
- Single-line comments, 32
- Single quotes (')
 - characters, 57
 - escape sequence, 57
- Single-threaded programs, 816
- Size of arrays, 103–105, 114
- Sleep() method, 850–851
- Software development kit (SDK), 3
- Sort() method
 - arrays, 112
 - collections, 734–735
 - lists, 733
- Sorted collections, 745–747
- SortedDictionary<TKey, TValue>
 - class, 745–747
- SortedList<T> class, 745–747
- Sorts
 - bubble, 588–602
 - collections, 734–735
 - query expressions, 716–718
 - standard query operators, 676–678
- Source code
 - compiling, 3–4, 10–11, 34–35
 - creating, 3–4
 - debugging, 8–9
 - editing overview, 3–4
 - editing with dotnet CLI, 4–6
 - editing with Visual Studio 2019, 6–8
 - essential, 12–13
 - running, 3–4
- Special floating-point characteristics, 128–129
- Specialized types, 258–259
- Splitting statements across lines, 20
- Square brackets ([])
 - arrays, 101–103, 108
 - attributes, 783–784
 - indexers, 750
 - null-conditional operators, 158–160
- Stack class, 534–537
- Stack unwinding, 217
- Stack<T> class
 - collections, 658–659, 747
 - declaring, 581, 779
 - generics, 542–543
- StackOverflowException type, 248, 514, 520
- Stacks
 - collections, 747
 - exception handling, 521
 - methods, 216–217

- pointers, 967–968
- rethrowing exceptions, 517–518
- thread synchronization, 939
- for undo operations, 540–542
- value types, 419
- Stand-alone executables, 11–12
- Standard query operators
 - anonymous types, 695–703
 - collections, 653–654
 - deferred execution, 672–676
 - element counting, 671–672
 - example code, 691–692
 - filters, 666–667
 - join, 678–691
 - list of, 693
 - overview, 663–666
 - parallel queries, 670–671
 - projections, 668–669
 - queryable extensions, 694–695
 - sorting, 676–678
- Start() method, 406–407, 826, 829
- StartNew() method, 847–849, 914
- StartsWith() method, 63
- Statement lambdas, 601–604
- Statements
 - delimiters, 19–20
 - vs. methods, 202–203
 - multiple on one line, 20
 - splitting across lines, 20
 - without semicolons, 20
- States
 - collections, 660
 - iterators, 757–759
- STAThreadAttribute, 945
- Static classes, 319–321
- Static compilation vs. dynamic programming, 807–808
- static keyword, 311–313
- Static members
 - associated data, 314
 - constructors, 316–318
 - fields, 311–314
 - interface refactoring, 401
 - methods, 315–316
 - overview, 311
 - properties, 318–319
 - strings, 62–63
- Stop() method, 905
- Store() method, 270
- Storing
 - with files, 270–273
 - reference types, 419–421
 - value types, 418–419
- StreamingContext type, 526
- StreamReader class
 - async methods, 864
 - data retrieval, 273
- Streams, asynchronous, 870–874
- StringBuilder type, 69, 557
- Strings
 - as arrays, 115–117
 - comparing, 56
 - concatenating, 61, 126–127
 - conversions, 76–77
 - description, 59
 - directives, 64–65
 - enum conversions, 441–443
 - format, 29–30
 - formatting, 65–66
 - immutability, 26, 67–69, 968–970
 - vs. integers, 240–241
 - interpolation, 28–30, 61–62
 - length, 66–67
 - literal values, 59–61
 - locks, 924–925
 - methods, 62–64
 - modifying, 69
 - newlines, 66
 - null values, 69–70
 - nullable modifier, 70
 - nullable reference types, 71
 - properties, 66–70
 - reversing, 116–117
 - void values, 69
- Strong references in garbage collection, 491–493
- StructLayoutAttribute, 952–953
- structs
 - vs. classes, 763
 - constraints, 560–561
 - declaring, 422–423
 - generics, 544–545
 - initializing, 424–426
 - readonly, 423
 - unmanaged, 952–953
- Structural equality of delegates, 608–609
- Structured programming, 195
- Subscribers
 - delegate invocation, 629–632
 - events, 645
 - exception handling, 637–640

- hooking up, 628–629
- methods, 626–627
- Subscriptions. *See* Publish-subscribe pattern
- Subtraction
 - binary operator, 123
 - compound assignment, 132
 - decrement operator, 133
 - unary operator, 122–123
- Subtypes, 258, 336
- Suffixes
 - attributes, 786
 - literal values, 51–52
 - numeric types, 46
- Sum() function, 693
- Super types, 258, 336
- SuppressFinalize() method, 500–501
- Surrogate pairs, 57
- switch statements
 - goto in, 184–185
 - overview, 141
 - pattern matching, 371–372
 - working with, 176–179
- Symbols, preprocessor, 189
- Synchronization
 - lock statement, 432–434
 - thread. *See* Thread synchronization
- Synchronization context with task
 - schedulers, 887–889
- Synchronized method, 925
- Synchronous delegates, 824
- Syntax
 - identifiers, 15–16
 - iterators, 755
 - keywords, 13–15, 17
 - Main(), 18–19
 - methods, 18
 - tuples, 92
 - type definitions, 17
 - variables, 22–23
- System.Action delegates, 592–594
- System.ApplicationException type, 247, 514
- System.ArgumentNullException type, 247, 285, 513–514
- System.ArithmeticException type, 248
- System.Array class, 113–114
- System.Array.Reverse() method, 112, 116–117
- System.ArrayTypeMismatchException type, 248
- System.Attribute class, 787
- System.AttributeUsageAttribute class, 793–795, 797
- System.Boolean class, 56
- System.Char class, 56
- System.Collections.Generic class
 - linked lists, 749
 - stacks, 542–543
- System.Collections.Generic.ICollection<T> interface
 - collection initializers, 654–655
 - element counting, 671–672
 - members, 730–731
- System.Collections.Generic.IEnumerable<T> interface. *See* IEnumerable<T> interface
- System.Collections.Generic.IEnumerator<T> interface, 658–662
 - foreach loops, 658–662
 - iterators, 755–756, 764
 - yield statement, 767
- System.Collections.Generic.List<T> class, 736–738
- System.Collections.Generic.Stack<T> type, 658–659
- System.Collections.Generics namespace, 199
- System.Collections.IEnumerable interface, 760
- System.Collections.IEnumerator interface, 658–662
- System.Collections.Immutable library, 939
- System.Collections.Stack class, 534–537
- System.ComponentModel class, 953
- System.Console class
 - input methods, 26–27, 197
 - newlines, 59, 66
 - output methods, 19, 28–33, 197
 - round-trip formatting, 54–55
- System.Convert type, 76
- System.Data namespace, 199
- System.Delegate class
 - assignment operators, 634, 650
 - constraints, 558–559
 - internals, 597–598, 636–637
- System.Diagnostics class
 - debugging with, 799
 - notifications, 885

- nullable attributes, 306–309
 - processes, 815
- System.Drawing class, 199
- System.Dynamic.DynamicObject class, 808–811
- System.Dynamic.
 - IDynamicMetaObjectProvider interface, 808
- System.Enum class, 439, 442
- System.Environment class
 - command-line arguments, 19, 216
 - new lines, 66
 - thread synchronization, 933
- System.EventArgs class, 645–647
- System.EventHandler<T> class, 647–648, 650–651
- System.Exception type, 511–512, 514
 - catch blocks, 518–520
 - description, 247
- System.FormatException type, 244, 247
- System.Func delegates, 592–594
- System.GC class
 - finalization, 500
 - finalizers, 496
 - garbage collection timing, 490
 - resource cleanup, 506–508
- System.Index type, 112
- System.IndexOutOfRangeException type, 247
- System.IntPtr class, 951, 963
- System.InvalidCastException type
 - description, 247
 - generics, 543
 - using, 472
- System.InvalidOperationException type. *See*
 - InvalidOperationException type
- System.IO class, 199, 520
- System.IO.DirectoryInfo class
 - extension methods, 321
 - inner joins, 682
- System.IO.FileAttributes class, 443
- System.IO.FileInfo class
 - inner joins, 682
 - projections, 668–669, 717–718
 - query expressions, 710
- System.IO.FileStream class, *See*
 - FileStream class
- System.Lazy<T> class, 509–510
- System.Linq.Async NuGet package, 875
- System.Linq class, 199, 684–688
- System.Linq.Enumerable class, *See*
 - Enumerable class
- System.Linq.Extensions.Enumerable class, 677
- System.Linq.ParallelEnumerable class, 670, 906–907
- System.Linq.Queryable class
 - delegates, 620
 - queryable extensions, 694
- System.MulticastDelegate class
 - constraints, 558–559
 - internals, 597–598, 636–637
- System namespace, 199
- System.Net namespace, 483
- System.NotImplementedException type, 248
- System.NullReferenceException type. *See*
 - NullReferenceException type
- System.Object class
 - derivation from, 363–365
 - dynamic objects, 805–806
- System.ObsoleteAttribute, 797, 799–800
- System.OutOfMemoryException type, 514, 520
- System.OverflowException type, 73, 529
- System.Range type, 112
- System.Reflection class
 - delegates, 598
 - metadata, 772
- System.Runtime.CompilerServices.
 - CallSite<T> type, 805–806
- System.Runtime.CompilerServices.
 - CompilerGeneratedAttribute, 293
- System.Runtime.ExceptionServices class, 517
- System.Runtime.InteropServices class
 - SafeHandle, 955–956
 - unmanaged exceptions, 519
- System.Runtime.Serialization type, 526
- System.Security.AccessControl.
 - MutexSecurity class, 932–933
- System.StackOverflowException type, 248, 514, 520
- System.STAThreadAttribute, 945

System.String type, 59
 System.Text class
 description, 199
 type parameters, 557
 System.Text.RegularExpressions class, 199
 System.Text.StringBuilder type class, 69, 557
 System.Threading class, 199, 850–851
 System.Threading.CancellationToken class. *See* CancellationToken class
 System.Threading.Interlocked class, 137, 926–928
 System.Threading.ManualResetEvent class, 934, 937
 System.Threading.Monitor class
 locks, 432
 thread synchronization, 918–920, 923–924
 System.Threading.Mutex class, 932–933
 System.Threading.Tasks class, 199, 823, 914
 System.Threading.Tasks.
 TaskCanceledException type, 846–847
 System.Threading.Tasks.
 TaskScheduler class, 887
 System.Threading.Thread class, 823
 System.Threading.WaitHandle class, 933–934
 System.Timers class, 944
 System.Type class
 generic parameters, 780
 metadata, 770–771
 type parameters, 779
 System.ValueTuple class, 96–98, 550–551
 System.ValueType class, 427, 771
 System.Web class, 199, 483
 System.Windows class, 200
 System.Windows.Forms class, 944
 System.Xml namespace, 200

T

\t escape sequence, 58
 Tabs escape sequence, 58
 Task-based Asynchronous Pattern (TAP), 823
 async/await syntax, 861–867, 890–891
 async return types, 869–870

asynchronous lambdas and local functions, 881–887
 asynchronous streams, 870–874
 await operator, 892–893
 await using statement, 874–875
 complexity, 859–861
 description, 814
 high-latency example, 856–858
 IAsyncDisposable, 874–875
 LINQ with IAsyncEnumerable, 875–877
 overview, 853
 return types, 877–881
 synchronization context, 887–889
 synchronous issue example, 854–856
 task schedulers, 887–889
 ValueTask<T> returns, 861–867
 Windows UI, 889–891
 Task class, 823
 asynchronous tasks, 825–826
 thread synchronization, 914
 timers, 944
 Task.Factory class, 847–848, 914
 Task Parallel Library (TPL)
 asynchronous complexity, 859–861
 asynchronous example, 856–858
 cooperative cancellation, 843–847
 description, 814
 for loop parallel iterations, 897
 foreach loop parallel iterations, 898–899
 performance, 899–900
 purpose, 823–824
 thread synchronization, 914
 threads pools, 851
 TaskCanceledException type, 846–847
 TaskCompletionSource<T> class, 884–885
 TaskContinuationOptions enums, 831–834
 Tasks
 asynchronous. *See* Asynchronous tasks
 canceling, 843–849
 continuation, 829–836
 description, 816
 disposability of, 849
 exceptions, 836–840
 long-running, 848–849
 polling, 826–828
 schedulers, 824, 887–889

- TaskScheduler class, 887
- TaskScheduler property, 904
- Temporary storage pools
 - reference types, 421
 - value types, 419
- TemporaryFileStream class, 495–497
- Ternary operator, 154
- Text class
 - description, 199
 - type parameters, 557
- ThenBy() method, 676–678
- ThenByDescending() method, 678
- this keyword
 - class members, 266–270
 - constructors, 300–301
 - extension methods, 322
 - locks, 924–925
 - nested classes, 328
 - static methods, 316
- Thread class, 823, 850–851
- Thread pools
 - description, 816
 - multithreading, 823
- Thread-safe code and operations
 - delegate invocation, 632
 - description, 816, 917
 - event notifications, 929
 - increment and decrement, 137
- Thread synchronization
 - best practices, 929–931
 - COM threading model, 945
 - concurrent collection classes, 938–939
 - deadlocks, 929–930
 - event notification, 928–929
 - Interlocked class, 926–928
 - local variables, 917–918
 - locks, 920–925, 931
 - monitors, 918–920
 - Mutex class, 932–933
 - need for, 931
 - overview, 913–914
 - purpose, 914–917
 - reset events, 934–937
 - semaphores, 937–938
 - task return, 922–923
 - thread local storage, 939–943
 - timers, 943–944
 - volatile fields, 926
 - WaitHandle class, 933–934
- Threading class, 850–851
- ThreadLocal<T> type, 940–941
- ThreadPool type, 850–851
- Threads
 - description, 816
 - exceptions, 840–843
 - local storage, 939–943
 - multithreading. *See* Multithreading
 - pools, 850–851
 - synchronization. *See* Thread synchronization
- ThreadStaticAttribute, 941–942
- Throw() method, 517–518, 521
- throw statements, 249–251, 512
- ThrowIfCancellationRequested()
 - method, 847
- Throwing exceptions, 512
 - in catch blocks, 521
 - error reporting, 249–251
 - error trapping, 241
 - stack information in, 517–518
- Tildes (~)
 - complement operator, 167
 - finalizers, 494
 - list searches, 736
 - overriding, 468
- Time slices, 817–818
- Timers class, 944
- Timers for thread synchronization, 943–944
- ToArray() method, 674–675
- ToAsyncEnumerable() method, 876–877
- ToCharArray() method, 116
- ToDictionary() method, 674–675
- ToEnumerable() method, 876
- ToList() method, 674–675
- ToLookup() method, 674–675
- ToLower() method, 64
- Torn reads, 917
- ToString() method
 - conversions, 77
 - description, 364
 - enum, 441, 447
 - FlagsAttribute, 796–797
 - overriding, 452–453
- Total ordering collections, 735–736
- ToUpper() method, 64
- TPL. *See* Task Parallel Library (TPL)
- Trace() method, 799
- TRACE preprocessor identifier, 799
- Trapping errors, 241–246

- Trim() method, 64
- TrimEnd() method, 64
- TrimStart() method, 64
- TrimToSize() method
 - lists, 731–733
 - queues, 748
- True/false evaluations, 55
- true operator, overriding, 468–470
- Try blocks, 242–244
- TryParse() method
 - attributes, 791–793
 - conversions, 77–79
 - dynamic invocation, 777
 - enum, 442
- Tuples
 - declaring and assigning, 92–96
 - generics, 550–552
 - GetHashCode() and Equals()
 - overriding, 463–464
 - groups, 720–721
 - pattern matching, 368
 - projections, 669
 - query expressions, 710–711
 - return values, 207–208, 222–223
 - System.ValueTuple type, 96–98
- Two-dimensional arrays, 102, 104–105
- Two's complement notation, 162–163
- Type checking, 984
- Type class
 - generic parameters, 779–780
 - metadata, 770–771
 - type parameters, 779
- Type parameters
 - cascading, 552
 - constraints. *See* Constraints
 - contravariance, 577–579
 - covariance, 575–577
 - generics, 542, 545
 - lists, 205
 - multiple, 549–551
 - names, 544
 - null modifier, 308–309
 - obtaining, 780–781
 - type of, 779
- Type safety, 537
 - anonymous types, 700–701
 - CLI, 984
 - generics, 543
 - managed execution, 35
- typeof() method
 - attributes, 789
 - locks, 924–925
 - reflection, 771–773
- Types
 - anonymous. *See* Anonymous types
 - arrays. *See* Arrays
 - Boolean, 55–56
 - character, 56
 - conversion overview, 72–77
 - CTS, 990
 - decimal, 49
 - declarations, access modifiers on,
 - 479–480
 - default, 50–52
 - definitions, 17
 - delegates, 590–594, 647
 - description, 23
 - dynamic, 803
 - exceptions, 511–514
 - floating-point types, 47–49
 - generics, 540
 - hardcoding values, 50
 - hexadecimal notation, 52–54
 - implicitly typed local variables, 89–90
 - integers, 46–47
 - lambda expressions, 605
 - local variables, 23
 - methods, 197–198, 200–201
 - nested, 552–553
 - null allowed, 84–85
 - null checks, 85–86
 - nullable reference, 87–89
 - nullable values, 86
 - overview, 45
 - parameters, 949–951
 - parsing, 77–79
 - pattern matching, 366–368
 - reference, 81–84
 - retrieving, 771–772
 - return values, 206–208
 - strings. *See* Strings
 - tuples, 92–98
 - Unicode standard, 56
 - unmanaged, 963
 - from unmanaged structs, 952–953
 - value. *See* Values and value types
 - verifying, 365–366
 - well-formed. *See* Well-formed types

U

- \u escape sequence, 58
- uint type, 46
- ulong type, 46
- Unary expression trees, 618–619
- Unary operators
 - overriding, 468–470
 - plus and minus, 122–123
- UnauthorizedAccessException type, 527, 901
- Unboxing operation, 428–431, 434–436
- Unchecked conversions, 73–75, 528–530
- unchecked keyword, 75, 530
- Uncompress() method, 379
- #undef directive, 187, 189
- Underscores (_)
 - digit separators, 52
 - field names, 281
 - identifier names, 16–17
 - variable names, 24
 - VB line continuation character, 20
- Undo operations
 - generics for, 540–542
 - stacks for, 534–537
- Unhandled exceptions
 - description, 241
 - parallel loops, 900–901
 - reporting, 520
 - tasks, 836–840
 - threads, 840–843
- UnhandledException type, 841
- Unicode standard
 - character representation, 56–58
 - escape sequence, 58
 - localized applications, 56
- Union() operator, 693
- Unity .NET frameworks, 39
- Unmanaged code and types
 - buffer overflows, 109
 - calls into, 948–949
 - constraints, 559–560
 - description, 34
 - exceptions, 519
 - guidelines, 960
 - parameters, 949–951
 - performance, 985–986
 - pointers, 951, 960, 963
 - resource cleanup, 503–506
 - stack allocation, 967
 - structs, 952–953
- UnobservedTaskException type, 840
- Unreachable end points in methods, 206–207
- Unsafe code, 947
 - delegates, 971–972
 - platform interoperability. *See* Platform interoperability
 - pointers, 961–963
- unsafe modifier, 961–962
- Unwrap() method, 859–860
- UserName class, 933
- ushort type, 46
- using directive
 - aliasing, 213–214
 - deterministic finalization, 496–500
 - namespaces, 209–211
 - nested, 211–212
 - strings, 64–65
- using static directive
 - strings, 64–65
 - working with, 212–213

V

- Validation, properties with, 283–285
- value keyword for properties, 278–279
- Values and value types
 - boxing, 428–436
 - CTS, 990
 - description, 418
 - enums. *See* enum values
 - generic instantiation, 582–583
 - hardcoding, 50
 - immutable, 424
 - inheritance and interfaces, 427
 - from iterators, 755–762
 - lock statement, 432–434
 - new operator, 426–427
 - overview, 81–82, 417
 - parameters, 217–218
 - vs. reference, 419–421
 - structs, 422–427
 - variables, 418–419
- Values property for sorted collections, 746
- ValueTask<T> class, 864, 867–869, 876–877
- ValueTuple class, 550–551
- ValueType class, 427, 771
- var keyword
 - pattern matching, 366–368
 - type declarations, 90

Variable parameters, defining index operators with, 752

Variables

- anonymous types, 695–701
- copying, 418–419
- declaring, 23–24
- description, 22–23
- for loops, 170
- foreach loops, 174
- immutable, 26
- implicitly typed local, 89–90
- instance fields, 262–265
- lambda expressions, 611–616
- parameters, 197
- query expressions, 707–708, 721
- reference types, 84, 419–421
- scope, 147–149
- thread synchronization, 917–918
- tuples, 92–96
- value types, 82, 418–419
- working with, 25

Variadic generic types, 552

Variance. *See* Contravariance; Covariance

Variants, 90

Verbatim strings

- interpolation, 61–62
- working with, 59–60

Versioning

- C# 8.0 and later, 398–400
- encapsulation and polymorphism with protected interface members, 406–411
- .NET frameworks, 40–42
- overview, 396–397
- pre C# 8.0, 397–398
- refactoring features, 401–405

Vertical bars (|)

- bitwise operators, 164
- compound assignment, 167
- logical operators, 151–152
- overriding, 466, 468

Vertical tabs escape sequence, 58

Virtual abstract members, 357–360

Virtual Execution System (VES)

- CLI, 982
- description, 994
- managed execution, 34, 974
- runtime, 35

Virtual functions in C++, 361

Virtual methods

- overridden, 355–356
- overriding, 346–349

virtual modifier

- base class overriding, 346–349
- interface refactoring, 404

VirtualAllocEx() method, 950

VirtualMemoryManager class, 949

Visual Basic language vs. C#

- global methods, 204
- global variables and functions, 311
- Imports directive, 210
- line-based statements, 20
- Me keyword, 268
- named arguments, 238
- redimensioning arrays, 114
- variable declarations, 90
- void type, 72

Visual code editors, 191–193

Visual Studio 2019

- CLI, 3–4
- code building and executing, 13
- debugging with, 9
- NuGet references, 476–477
- preprocessor directives, 192–193
- project and library references, 474–475
- working with, 3–4, 6–8

void type

- asynchronous method returns, 877–880
- partial methods, 332
- pointers, 969
- return values, 207
- strings, 69, 71–72

Volatile fields, 926

volatile modifier, 926

W

Wait() method

- asynchronous requests, 860
- asynchronous tasks, 826
- reset events, 934, 936–937

WaitAll() method


- asynchronous tasks, 826
- thread synchronization, 933

WaitAny() method

- asynchronous tasks, 826
- thread synchronization, 933

- WaitForPendingFinalizers() method, 496, 506
 - WaitHandle class, 933–934
 - WaitOne() method, 934, 936
 - #warning directive, 187, 189–190
 - Warnings
 - method overriding, 349
 - preprocessor directives, 186–191
 - Weak references in garbage collection, 491–493
 - Web class, 199, 483
 - Well-formed types
 - assembly references, 472–479
 - encapsulation, 479–481
 - garbage collection, 489–493
 - lazy initialization, 508–510
 - namespaces, 481–484
 - overriding object members, 451–464
 - overriding operators, 464–472
 - overview, 451
 - resource cleanup. *See* Resource cleanup
 - XML comments, 485–489
 - Where() operator, 666–667, 672–676
 - while loops, 139, 168–170
 - Whitespace
 - overview, 21–22
 - in strings, 60
 - trimming, 284
 - Win32Exception() method, 953–955
 - Windows class, 200
 - Windows Forms, 889, 944
 - Windows Presentation Foundation (WPF)
 - description, 977
 - high-latency example, 889–891
 - Windows UI applications, 889–891
 - WithCancellation() method, 911
 - Work stealing, 899
 - WPF (Windows Presentation Foundation)
 - description, 977
 - high-latency example, 889–891
 - Wrapped exceptions, 246, 527–530
 - Wrappers with API calls, 959
 - Write() method
 - console output, 29–30
 - invoking, 197–198
 - newlines, 59, 66
 - strings, 62
 - Write-only properties, 286–287
 - WriteLine() method
 - console output, 29–30
 - invoking, 197–198
 - newlines, 59, 66
 - round-trip formatting, 54–55
 - strings, 62
- X**
- \x escape sequence, 58
 - Xamarin compiler, 976, 978
 - Xamarin .NET frameworks, 39
 - XAML (Extended Application Markup Language), 977
 - xcopy deployment, 989
 - XML (Extensible Markup Language)
 - binding elements, 806–807
 - comments, 485–489, 770
 - delimited comments, 72
 - documentation files, 487–489
 - namespace, 200
 - overview, 33
 - reflection, 770
 - single-line comments, 32
 - XmlSerializer class, 771
 - XOR (exclusive OR) operators
 - bitwise, 164–165
 - logical, 152–153
- Y**
- yield statements, 15n
 - contextual keyword, 766
 - requirements, 767–768
 - yield break, 763–764
 - yield return, 755–756, 760–762
 - Yielding values from iterators, 755–762
- Z**
- Zero-based arrays, 99–100
 - Zeros
 - division by, 131
 - floating-point types, 132
 - Zip() method, 875

This page intentionally left blank



Index of 8.0 Topics

A

Abstract classes, 357
Aggregation, 345
async returns, 880–881
Asynchronous streams, 854, 870, 872, 874
await using statement, 874

B

Buffer overflows, 110

C

Constraints, 561
Constructor overloading, 300

D

Delegate invocation, 630

I

Index from end operator, 99, 107, 108, 110, 112, 118
Interfaces
 vs. abstract classes, 414
 implementation, 377, 379–380, 386, 413–414
 versioning, 396–411
is operator, 365
is { } property pattern, 157

N

.NET frameworks, 41
Non-nullable reference types, 293

Null assignment for reference types, 70–71

Null-coalescing assignment, 155, 158

Null-forgiving operator, 634

Nullable reference types

 default values, 548

 introduced, 70–71, 88

 nullable modifier, 84

P

Pattern matching types, 366, 368

Positional pattern matching, 369

Property matching, 371

R

Ranges, 101, 111

Read-only struct members, 423

Resource cleanup, 499–500

Reverse accessing arrays, 100

S

String interpolation, 61

T


Task-based Asynchronous Pattern, 863

U

Unmanaged constraints, 560

using statement, 499

This page intentionally left blank



Index of 7.0 Topics

A

Anonymous types, 695, 701
as operator, 374
async
 Main() method, 867, 922
 return types, 864, 868
async/await support, 19

B

Binary values, 53

C

Constructor implementation, 300

D

Deconstructors, 309–310
Default values, 548
Delegate constraints, 558, 567
Delegates, 590
Digit separators, 52–53
dotnet CLI, 474

I

in modifier, 223
is operator, 155, 157, 365–366

L

Local functions, 882

N

.NET frameworks, 41
Null assignments to reference types, 88
Null checks, 155–157

Null-coalescing assignment, 155
Nullable reference types, 548

O

Operator overriding, 465
out argument, 78, 222–223

P

Pattern matching
 is operator, 365–366
 is null operator, 156
 positional, 369
 switch statements, 179, 372
private protected modifier, 481
Properties
 getters and setters, 279
 validation, 283

R

Read-only pass by reference, 223
Return by reference, 223–225
Return types, tuples, 207, 222–223

S

struct declarations, 423
switch statements, 179, 372

T

throw expressions, 512, 517
Tuples
 anonymous type replacement, 91
 example code, 94
 generics, 550–551

introduced, 81
item names, 96, 98
method overriding, 463–464
return types, 207, 222–223
variable assignment, 92

U

Unmanaged constraints, 559

V

Versioning, 397–398



Index of 6.0 Topics

A

Automatically implemented properties,
867
initializing, 281, 319
read-only, 287, 325–326
structs, 423–424, 426
await operator, 875n

C

Catch blocks, 516
Collections, 655–656

D

Delegates, 161
Dictionaries, 740

E

Event notifications, 928
Expression bodied methods, 208

G

Generics, 551

N

nameof() operator, 285, 513, 781
.NET frameworks, 41–42

Null checks, 157

Null-conditional operator
dereferencing operations, 513
event handlers, 161
null checks, 86, 158–159, 630
Nullable reference types, 548

R

Read-only properties, 284, 286–287

S

Strings
formatting, 65
interpolation, 28–30, 61
structs, 423–424, 426

U

using static directive, 64–65

V

Versioning, 397–398

