



ESSENTIAL C# 7.0



“This book has been a classic for years, and remains as one of the most venerable and trusted titles in the world of C# content, and probably far beyond!”

—Mads Torgersen

MARK MICHAELIS
ERIC LIPPERT, *Technical Editor*

Foreword by **MADS TORGERSEN**,
C# Program Manager, Microsoft



IntelliTect

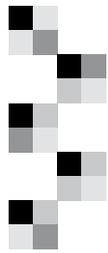
FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Essential C# 7.0

This page intentionally left blank



Essential C# 7.0

■ **Mark Michaelis**
with Eric Lippert, Technical Editor

◆ Addison-Wesley

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

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: 2018933128

Copyright © 2018 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. 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.pearsoned.com/permissions/.

ISBN-13: 978-1-5093-0358-8

ISBN-10: 1-5093-0358-8

To my family: Elisabeth, Benjamin, Hanna, and Abigail.

*You have sacrificed a husband and daddy for countless hours of writing,
frequently at times when he was needed most.*

Thanks!



This page intentionally left blank



Contents at a Glance

Contents ix
Figures xv
Tables xvii
Foreword xix
Preface xxi
Acknowledgments xxxiii
About the Author xxxv

1 Introducing C# 1
2 Data Types 43
3 More with Data Types 77
4 Operators and Control Flow 109
5 Methods and Parameters 181
6 Classes 241
7 Inheritance 313
8 Interfaces 353
9 Value Types 379
10 Well-Formed Types 411
11 Exception Handling 465

12	Generics	487
13	Delegates and Lambda Expressions	537
14	Events	575
15	Collection Interfaces with Standard Query Operators	603
16	LINQ with Query Expressions	657
17	Building Custom Collections	679
18	Reflection, Attributes, and Dynamic Programming	721
19	Multithreading	771
20	Thread Synchronization	863
21	Platform Interoperability and Unsafe Code	897
22	The Common Language Infrastructure	923
	<i>Index</i>	<i>945</i>
	<i>Index of 7.0 Topics</i>	<i>995</i>
	<i>Index of 6.0 Topics</i>	<i>998</i>
	<i>Index of 5.0 Topics</i>	<i>1001</i>



Contents

Figures xv

Tables xvii

Foreword xix

Preface xxi

Acknowledgments xxxiii

About the Author xxxv

1 Introducing C# 1

Hello, World 2

C# Syntax Fundamentals 11

Working with Variables 20

Console Input and Output 24

Managed Execution and the Common Language Infrastructure 32

Multiple .NET Frameworks 37

2 Data Types 43

Fundamental Numeric Types 44

More Fundamental Types 53

null and void 67

Conversions between Data Types 69

3 More with Data Types 77

Categories of Types 77

Nullable Modifier 80

Tuples 83

Arrays 90

4 Operators and Control Flow 109

- Operators 110
- Introducing Flow Control 126
- Code Blocks ({}) 132
- Code Blocks, Scopes, and Declaration Spaces 135
- Boolean Expressions 137
- Bitwise Operators (<<, >>, |, &, ^, ~) 147
- Control Flow Statements, Continued 153
- Jump Statements 165
- C# Preprocessor Directives 171

5 Methods and Parameters 181

- Calling a Method 182
- Declaring a Method 189
- The using Directive 195
- Returns and Parameters on Main() 200
- Advanced Method Parameters 203
- Recursion 215
- Method Overloading 217
- Optional Parameters 220
- Basic Error Handling with Exceptions 225

6 Classes 241

- Declaring and Instantiating a Class 245
- Instance Fields 249
- Instance Methods 251
- Using the this Keyword 252
- Access Modifiers 259
- Properties 261
- Constructors 278
- Static Members 289
- Extension Methods 299
- Encapsulating the Data 301

Nested Classes 304

Partial Classes 307

7 Inheritance 313

Derivation 314

Overriding the Base Class 326

Abstract Classes 338

All Classes Derive from `System.Object` 344

Verifying the Underlying Type with the `is` Operator 345

Pattern Matching with the `is` Operator 346

Pattern Matching within a `switch` Statement 347

Conversion Using the `as` Operator 349

8 Interfaces 353

Introducing Interfaces 354

Polymorphism through Interfaces 355

Interface Implementation 360

Converting between the Implementing Class and Its Interfaces 366

Interface Inheritance 366

Multiple Interface Inheritance 369

Extension Methods on Interfaces 369

Implementing Multiple Inheritance via Interfaces 371

Versioning 374

Interfaces Compared with Classes 375

Interfaces Compared with Attributes 377

9 Value Types 379

Structs 383

Boxing 390

Enums 398

10 Well-Formed Types 411

Overriding object Members 411

Operator Overloading 424

Referencing Other Assemblies 432

- Defining Namespaces 442
- XML Comments 445
- Garbage Collection 449
- Resource Cleanup 452
- Lazy Initialization 461

11 Exception Handling 465

- Multiple Exception Types 465
- Catching Exceptions 469
- General Catch Block 473
- Guidelines for Exception Handling 475
- Defining Custom Exceptions 479
- Rethrowing a Wrapped Exception 483

12 Generics 487

- C# without Generics 488
- Introducing Generic Types 493
- Constraints 506
- Generic Methods 519
- Covariance and Contravariance 524
- Generic Internals 531

13 Delegates and Lambda Expressions 537

- Introducing Delegates 538
- Declaring Delegate Types 542
- Lambda Expressions 550
- Anonymous Methods 556

14 Events 575

- Coding the Publish-Subscribe Pattern with Multicast Delegates 576
- Understanding Events 591

15 Collection Interfaces with Standard Query Operators 603

- Collection Initializers 604
- What Makes a Class a Collection: IEnumerable<T> 607

Standard Query Operators	613
Anonymous Types with LINQ	646
16 LINQ with Query Expressions	657
Introducing Query Expressions	658
Query Expressions Are Just Method Invocations	676
17 Building Custom Collections	679
More Collection Interfaces	680
Primary Collection Classes	683
Providing an Indexer	702
Returning Null or an Empty Collection	705
Iterators	705
18 Reflection, Attributes, and Dynamic Programming	721
Reflection	722
nameof Operator	733
Attributes	735
Programming with Dynamic Objects	759
19 Multithreading	771
Multithreading Basics	774
Working with System.Threading	781
Asynchronous Tasks	789
Canceling a Task	810
The Task-based Asynchronous Pattern	816
Executing Loop Iterations in Parallel	846
Running LINQ Queries in Parallel	856
20 Thread Synchronization	863
Why Synchronization?	864
Timers	893
21 Platform Interoperability and Unsafe Code	897
Platform Invoke	898
Pointers and Addresses	910
Executing Unsafe Code via a Delegate	920

22 The Common Language Infrastructure 923

Defining the Common Language Infrastructure 924

CLI Implementations 925

.NET Standard 928

Base Class Library 929

C# Compilation to Machine Code 929

Runtime 932

Assemblies, Manifests, and Modules 936

Common Intermediate Language 939

Common Type System 939

Common Language Specification 940

Metadata 941

.NET Native and Ahead of Time Compilation 942

Index 945

Index of 7.0 Topics 995

Index of 6.0 Topics 998

Index of 5.0 Topics 1001



Figures

- FIGURE 1.1: *The New Project Dialog* 6
FIGURE 1.2: *Dialog That Shows the Program.cs File* 7
- FIGURE 3.1: *Value Types Contain the Data Directly* 78
FIGURE 3.2: *Reference Types Point to the Heap* 79
- FIGURE 4.1: *Corresponding Placeholder Values* 147
FIGURE 4.2: *Calculating the Value of an Unsigned Byte* 148
FIGURE 4.3: *Calculating the Value of a Signed Byte* 148
FIGURE 4.4: *The Numbers 12 and 7 Represented in Binary* 150
FIGURE 4.5: *Collapsed Region in Microsoft Visual Studio .NET* 178
- FIGURE 5.1: *Exception-Handling Control Flow* 229
- FIGURE 6.1: *Class Hierarchy* 244
- FIGURE 7.1: *Refactoring into a Base Class* 315
FIGURE 7.2: *Simulating Multiple Inheritance Using Aggregation* 324
- FIGURE 8.1: *Working around Single Inheritances with Aggregation and Interface* 373
- FIGURE 9.1: *Value Types Contain the Data Directly* 380
FIGURE 9.2: *Reference Types Point to the Heap* 382
- FIGURE 10.1: *Identity* 416
FIGURE 10.2: *The Project Menu* 437
FIGURE 10.3: *The Browse Filter* 437
FIGURE 10.4: *XML Comments as Tips in Visual Studio IDE* 446

- FIGURE 13.1: *Delegate Types Object Model* 548
- FIGURE 13.2: *Anonymous Function Terminology* 551
- FIGURE 13.3: *The Lambda Expression Tree Type* 569
- FIGURE 13.4: *Unary and Binary Expression Tree Types* 569
-
- FIGURE 14.1: *Delegate Invocation Sequence Diagram* 585
- FIGURE 14.2: *Multicast Delegates Chained Together* 587
- FIGURE 14.3: *Delegate Invocation with Exception Sequence Diagram* 588
-
- FIGURE 15.1: *A Class Diagram of IEnumerator<T> and IEnumerator Interfaces* 608
- FIGURE 15.2: *Sequence of Operations Invoking Lambda Expressions* 625
- FIGURE 15.3: *Venn Diagram of Inventor and Patent Collections* 629
-
- FIGURE 17.1: *Collection Class Hierarchy* 681
- FIGURE 17.2: *List<> Class Diagrams* 684
- FIGURE 17.3: *Dictionary Class Diagrams* 691
- FIGURE 17.4: *Sorted Collections* 698
- FIGURE 17.5: *Stack<T> Class Diagram* 699
- FIGURE 17.6: *Queue<T> Class Diagram* 700
- FIGURE 17.7: *LinkedList<T> and LinkedListNode<T> Class Diagram* 701
- FIGURE 17.8: *Sequence Diagram with yield return* 710
-
- FIGURE 18.1: *MemberInfo Derived Classes* 730
- FIGURE 18.2: *BinaryFormatter Does Not Encrypt Data* 754
-
- FIGURE 19.1: *Clock Speeds over Time* 772
- FIGURE 19.2: *Deadlock Timeline* 780
- FIGURE 19.3: *CancellationTokenSource and CancellationToken Class Diagrams* 813
-
- FIGURE 21.1: *Pointers Contain the Address of the Data* 913
-
- FIGURE 22.1: *Compiling C# to Machine Code* 931
- FIGURE 22.2: *Assemblies with the Modules and Files They Reference* 938



Tables

TABLE 1.1:	<i>C# Keywords</i>	12
TABLE 1.2:	<i>C# Comment Types</i>	30
TABLE 1.3:	<i>Predominant .NET Framework Implementations</i>	37
TABLE 1.4:	<i>C# and .NET Versions</i>	39
TABLE 2.1:	<i>Integer Types</i>	44
TABLE 2.2:	<i>Floating-Point Types</i>	46
TABLE 2.3:	<i>Decimal Type</i>	46
TABLE 2.4:	<i>Escape Characters</i>	55
TABLE 2.5:	<i>string Static Methods</i>	61
TABLE 2.6:	<i>string Methods</i>	61
TABLE 3.1:	<i>Sample Code for Tuple Declaration and Assignment</i>	84
TABLE 3.2:	<i>Array Highlights</i>	91
TABLE 3.3:	<i>Common Array Coding Errors</i>	105
TABLE 4.1:	<i>Control Flow Statements</i>	127
TABLE 4.2:	<i>Relational and Equality Operators</i>	138
TABLE 4.3:	<i>Conditional Values for the XOR Operator</i>	141
TABLE 4.4:	<i>Preprocessor Directives</i>	172
TABLE 4.5:	<i>Operator Order of Precedence</i>	178
TABLE 5.1:	<i>Common Namespaces</i>	185
TABLE 5.2:	<i>Common Exception Types</i>	232
TABLE 7.1:	<i>Why the New Modifier?</i>	331
TABLE 7.2:	<i>Members of System.Object</i>	344
TABLE 8.1:	<i>Comparing Abstract Classes and Interfaces</i>	376

TABLE 9.1:	<i>Boxing Code in CIL</i>	391
TABLE 10.1:	<i>Microsoft.Extension.CommandUtils Examples</i>	439
TABLE 10.2:	<i>Accessibility Modifiers</i>	441
TABLE 13.1:	<i>Lambda Expression Notes and Examples</i>	555
TABLE 15.1:	<i>Simpler Standard Query Operators</i>	643
TABLE 15.2:	<i>Aggregate Functions on System.Linq.Enumerable</i>	644
TABLE 18.1:	<i>Deserialization of a New Version Throws an Exception</i>	756
TABLE 19.1:	<i>List of Available TaskContinuationOptions Enums</i>	798
TABLE 19.2:	<i>Control Flow within Each Task</i>	827
TABLE 20.1:	<i>Sample Pseudocode Execution</i>	866
TABLE 20.2:	<i>Interlocked's Synchronization-Related Methods</i>	877
TABLE 20.3:	<i>Execution Path with ManualResetEvent Synchronization</i>	886
TABLE 20.4:	<i>Concurrent Collection Classes</i>	888
TABLE 22.1:	<i>Implementations of the CLI</i>	925
TABLE 22.2:	<i>Common C#-Related Acronyms</i>	943



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# series 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 Microsoft Professional Developers 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 scooped up like free hot-cakes: It was a big day for C# and .NET, and I was having a great time.

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

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

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

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

These are the same qualities that make Mark such a great writer. He goes right to the essence and communicates with great integrity—no sugarcoating—and has a keen eye for practical value and real-world problems. Mark has a great gift for providing clarity and elucidation, and no one will help you get C# 7.0 like he does.

Enjoy!

*—Mads Torgersen,
C# Program Manager,
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# 7.0 and .NET Framework 4.7/.NET Core 2.0.

- 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, ADO.NET, 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# 7.0* 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, 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 answers to 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 7.0, some of the most prominent enhancements are
 - String interpolation (see Chapter 2)
 - Implicitly typed variables (see Chapter 3)
 - Tuples (see Chapter 3)
 - Pattern matching (see Chapter 4)
 - Extension methods (see Chapter 6)
 - Partial methods (see Chapter 6)
 - 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 19)
- Parallel query processing with PLINQ (Chapter 19)
- Concurrent collections (Chapter 20)

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

Features of This Book

Essential C# 7.0 is a language book that adheres to the core C# Language 7.0 Specification. To help you understand the various C# constructs, it provides numerous examples demonstrating each feature. Accompanying each concept are guidelines and best practices, ensuring that code compiles, avoids likely pitfalls, and achieves maximum maintainability.

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

C# Coding Guidelines

One of the more significant enhancements included in *Essential C# 7.0* is 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.

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, or 7.0 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 Comment Samples
{
    static void Main()
    {

        string firstName; //Variable for storing the first name
        string lastName; //Variable for storing the last name

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

        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. */

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

- 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.
```

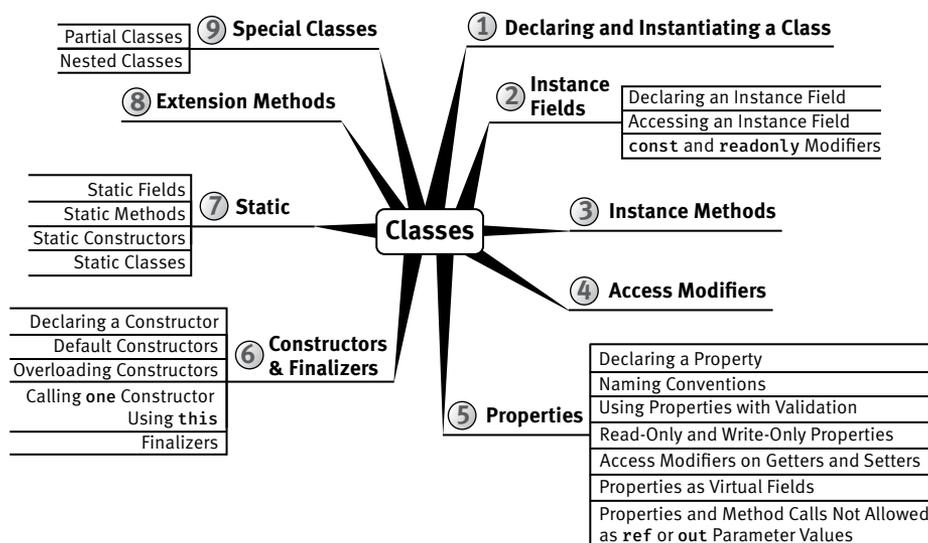
Although it might have been convenient to provide full code samples that you could copy into your own programs, doing so would detract from your learning a particular topic. Therefore, you need to modify the code samples before you can incorporate them into your programs. The core omission is error checking, 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 callout 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# 7.0*. 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 a chapter on the Common Language Infrastructure, which describes C# within the context of the development platform in which it operates. This chapter appears at the end because it is not C# specific and it departs from the syntax and programming style in the rest of the book. However, this chapter is suitable for reading at any time, perhaps most appropriately immediately following Chapter 1.

Here is a description of each chapter (in this list, chapter numbers shown in **bold** indicate the presence of C# 6.0–7.0 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 the nullable modifier and a C# 7.0-introduced

feature, tuples. 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.
- **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.
- **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 is perhaps the core feature missing from C# 1.0. This chapter fully covers this 2.0 feature. In addition, C# 4.0 added support for covariance and contravariance—something covered in the context of generics in this chapter.
- **Chapter 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 new 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 new `Enumerable` class. This class makes available an entirely new collection API known as the standard query operators and 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—*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 discusses how to work with threads and provides best practices to avoid the problems that plague multithreaded applications.
- **Chapter 20—*Thread Synchronization***: Building on the preceding chapter, this one demonstrates some of the built-in threading pattern support that can simplify the explicit control of multithreaded code.
- **Chapter 21—*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 22—*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# 5.0, 6.0, and 7.0 Topics***: These indexes provide quick references for the features added in C# 4.0 through 7.0. 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# 7.0* 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 (9781509303588) 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 sixth edition of the book, you can only imagine how much my family has sacrificed to allow me to write over the last 10 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.) A huge sorry and ginormous Thank You!

Over the years, many technical editors 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, Ian Davis, Doug Dechow, Gerard Frantz, Thomas Heavey, Anson Horton, Brian Jones, Shane Kercheval, Angelika Langer, Eric Lippert, John Michaelis, Jason Morse, Nicholas Paldino, Jon Skeet, Michael Stokesbary, Robert Stokesbary, John Timney, Neal Lundby, Andrew Comb, Jason Peterson, Andrew Scott, Dan Haley, Phil Spokas (who helped with portions of the writing in Chapter 22), and Kevin Bost.

Or course, Eric Lippert is no less than amazing. 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 is no longer. 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# 7.0*. Thanks, Eric! I can't imagine anyone better for the job. You deserve all the credit for raising the bar from good to great.

Like 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!

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 Trina Fletcher Macdonald, Anna Popick, Julie Nahil, and Carol Lallier. Trina deserves a special medal for putting up with the likes of me when she clearly was juggling myriad other more important things as well. Also, Carol's attention to detail was invaluable, and her ability to improve the writing and red-line potential writing faux pas (even catching them when they occurred in code listings) was so appreciated.



About the Author

Mark Michaelis is the founder of IntelliTect, a high-end software engineering and consulting company where he serves as the chief technical architect and trainer. Mark speaks at developer conferences and has written numerous articles and books. Currently, he is the Essential .NET columnist for *MSDN Magazine*.

Since 1996, Mark has been a Microsoft MVP for C#, Visual Studio Team System, and the Windows SDK. In 2007, he was recognized as a Microsoft Regional Director. He also serves on several Microsoft software design review teams, including C# and VSTS.

Mark holds a bachelor of arts in philosophy from the University of Illinois and a masters in computer science from the Illinois Institute of Technology.

When not bonding with his computer, Mark is busy with his family 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 Editor

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.

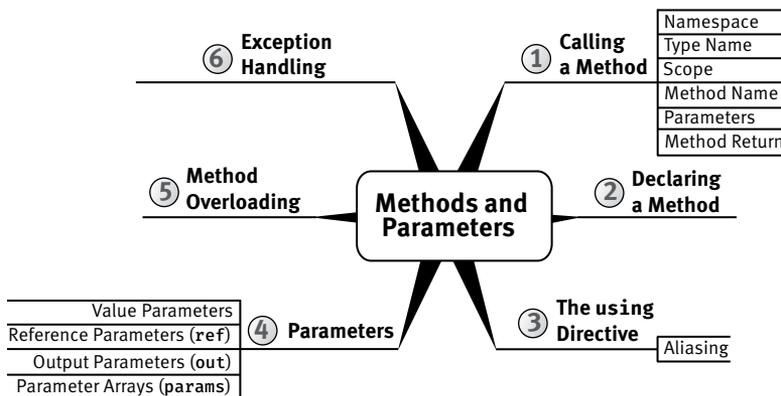


This page intentionally left blank

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 comprise 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 files, 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 stated. 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 the previous chapter, 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; the argument list 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
Your full name is Inigo Montoya.

```

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.

In earlier versions of Visual Studio, you can select a group of statements, right-click on it, and then select the Extract Method refactoring from the Refactoring section of the context menu to automatically move a group of statements to a new method. In Visual Studio 2015, the refactorings are available from the Quick Actions section of the context menu.

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 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 also 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 have an *unreachable end point*. That is, there must be no way for control to “fall off the end” of a method without it 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 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 GetUserInput(string prompt)
    {
        System.Console.Write(prompt);
        return System.Console.ReadLine();
    }
    static (string First, string Last) GetName()
    {
        string firstName, lastName;
        firstName = GetUserInput("Enter your first name: ");
```

```

        lastName = Get userInput("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 }!");
    }
}

```

End 7.0

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

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) =>
    $"{ firstName } { 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.

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 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 import Directive

Java allows for 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 the 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 means 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 also can 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

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

namespace declaration above or below the `EssentialCSharp` declaration, 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.

Begin 6.0

using static Directive

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 optional 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.Net;

class Program
{
    static int Main(string[] args)
    {
        int result;
        string targetFileName;
        string url;
        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");
                targetFileName = null;
                url = null;
                break;
            case 2:
                url = args[0];
                targetFileName = args[1];
                break;
        }

        if (targetFileName != null && url != null)
        {
            WebClient webClient = new WebClient();
            webClient.DownloadFile(url, targetFileName);
            result = 0;
        }
        else
        {
            Console.WriteLine(
                "Usage: Downloader.exe <URL> <TargetFileName>");
            result = 1;
        }
        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 on the command line which class to use for the `Main()` declaration. `csc.exe` includes an `/m` option to specify the fully qualified class name of `Main()`.

■ 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 entirely 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.

Begin 7.0

Output Parameters (out)

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

7.0

```
>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 (which would require referencing the `System.ValueTuple` NuGet package) but without C# 7.0 syntax.

7.0

■ NOTE

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

Read-Only Pass by Reference (`in`)

Begin 7.2

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 so that the value type could be passed by reference so that not only copy of the value type occurred but, in addition, the invoked method could not 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.

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)
```

7.0

```

{
    // 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 line 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: Object references shouldn't be garbage collected while they're still referenced, and 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; } }
```

Second, `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 workaround 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 = null; 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 parameters. 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.

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

```
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 != null)
    {
        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 most readable approach, it is generally not the fastest implementation. If

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

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. 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

```
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 != null)
    {
        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

Starting with C# 4.0, the language designers added support for **optional parameters**. By allowing the association of a parameter with a constant value as part of 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

```

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 be required in C# 3.0 and earlier. 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")

```

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.

4.0

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 made available in C# 4.0 is the use of **named arguments**. 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

```

class Program
{

```

```

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

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

```

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.

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, C# 4.0 now imposes 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.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.");
        }
        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 the Advanced Topic titled “General Catch” later in this chapter) optionally allows you to specify the data type of the exception, and 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.

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,

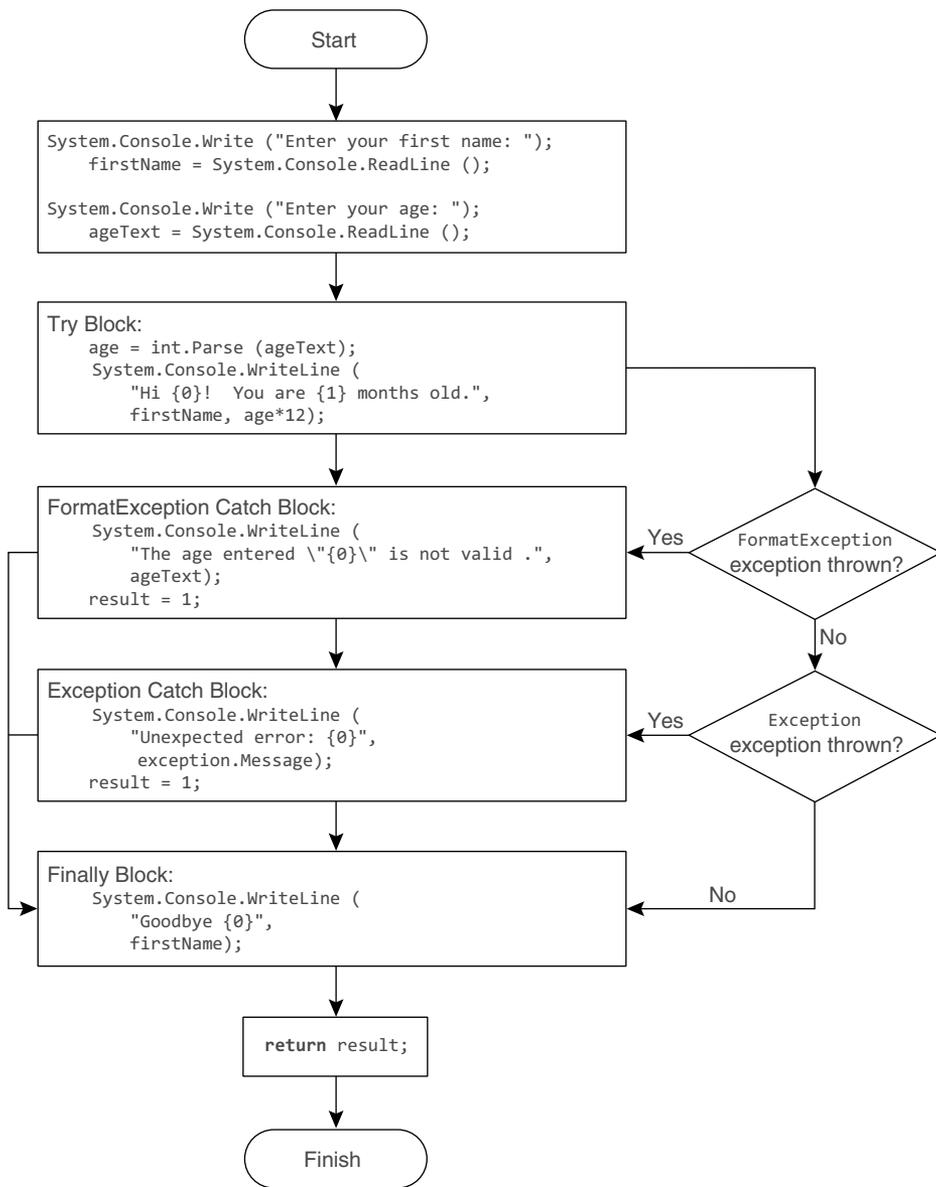


FIGURE 5.1: Exception-Handling Control Flow

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

Starting in C# 2.0, all objects thrown as exceptions derive from `System.Exception`. (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.

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.

TABLE 5.2: Common Exception Types (continued)

Exception Type	Description
<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.
<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

```

...
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){...}`. 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.)

The behavior starting in C# 2.0 varies slightly from the earlier C# behavior. In C# 2.0, if a language allows throwing non-System.Exception, the object of the thrown exception will be wrapped in a System.Runtime.CompilerServices.RuntimeWrappedException that does derive from System.Exception. Therefore, all exceptions, whether derived from System.Exception or not, will propagate into C# assemblies as if they were derived from System.Exception.

The result is that System.Exception catch blocks will catch all exceptions not caught by earlier blocks, and a general catch block, following a System.Exception catch block, will never be invoked. Consequently, following a System.Exception catch block with a general catch block in C# 2.0 or later will result in a compiler warning indicating that the general catch block will never execute.

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

```
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.⁶ In this case, the second catch block handles the exception and writes out 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.

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

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 make an effort to 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.⁷

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

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.

Additionally, 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.

Begin 2.0

■ ADVANCED TOPIC

Numeric Conversion with `TryParse()`

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. In the first release of C#, the only data type that enabled this behavior was a `double` method called `double.TryParse()`. However, this method is included with all numeric primitive types starting with the Microsoft .NET Framework 2.0. 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.");
}
```

With the Microsoft .NET Framework 4, a `TryParse()` method was also added to enum types.

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.

End 2.0

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

- . (dot) operator, 145, 920
- ?: (question mark, colon) conditional operator, 142–143
- ? . (question mark, dot) null-conditional operator, 144–146
- /// (forward slashes), XML comment delimiter, 447
- (hyphens), in identifier names, 14
- (minus sign)
 - arithmetic binary operator, 111–112
 - delegate operator, 583–584
 - precedence, 112
 - subtraction operator, overloading, 426–428
 - unary operator, 110–111, 428–430
- (minus sign, equal)
 - delegate operator, 583–584
 - minus assignment operator, 428
- (minus signs), decrement operator
 - C++ vs. C#, 124
 - decrement, in a loop, 121–124
 - description, 121
 - guidelines, 124
 - lock statement, 125
 - post-decrement operator, 122–124
 - postfix decrement operator, 123–124
 - pre-decrement operator, 123–124
 - prefix decrement operator, 123–124
 - race conditions, 125
 - thread safety, 125
 - unary operator, 428–430
- <> (angle brackets), in XML, 31
- & (ampersand) bitwise AND operator, 149–152, 151, 405–406
- && (ampersands) logical AND operator, 140, 428
- &= (ampersand, equal sign) bitwise AND assignment operator, 152–153
- * (asterisk) multiplication operator, 111–112, 426–428
- *= (asterisk, equal sign) multiplication assignment operator, 428
- @ (at sign)
 - coding verbatim strings, 57–58
 - keyword prefix, 15
- \ (backslashes), as literals, 58
- / (forward slash) division operator
 - description, 111–112
 - overloading, 426–428
 - precedence, 112
- / (forward slash) in XML, 31
- /= (slash, equal) division assignment operator, 428
- /unsafe switch, 912–913
- ^ (caret) bitwise XOR operator, 140–141, 149–152, 426–428
- ^= (caret, equal sign) bitwise XOR assignment operator, 152–153
- () (cast operator), overloading, 430–431
- () (parentheses), associativity/precedence, 430–431
- { } (curly braces)
 - C/C++ code style, 2
 - defining array literals, 93
 - formatting code, 20

- { } (curly braces) (*continued*)
 - forming code blocks, 133
 - methods definition, 16
 - omitting, 134
 - string interpolation, 27, 59
 - string formatting, 63
 - type definition, 15
 - in switch statements, 18
- [] (square brackets), array declaration, 92
- \$ (dollar sign), string interpolation, 26, 59
- \$@ (dollar sign, at sign), verbatim strings
 - with interpolation, 59–60
- " " (double quotes)
 - strings, 57–58
 - escape sequence for string literals, 58
- = (equal sign) assignment operator
 - vs. == (equality operator), C++ vs. C#, 138
 - assigning variables, 22
 - definition, 22
 - overloading, 424, 428
 - precedence, 112
- => (equal sign, greater than) lambda operator, 551, 554–555, 557
- == (equal equal sign) equality operator
 - as true/false evaluator, 53
 - overloading, 390, 424–425
 - in place of = (equal sign) assignment operator, 138
- ! (exclamation point)
 - logical NOT operator, 141
 - unary operator, 428–430
- != (exclamation point, equal sign) inequality operator
 - overloading, 390, 424–425
 - testing for inequality, 138–139
- < (less than sign) less than operator, 138, 424–426
- <= (less than, equal sign) less than or equal operator, 138, 424–426
- << (less than signs) shift left operator, 148–149, 426–428
- <<= (less than signs, equal) shift left assignment operator, 148–149
- > (greater than sign), greater than operator, 138, 424–425
- >= (greater than, equal sign), greater than or equal operator, 138, 424–425
- >> (greater than signs), shift right operator, 148–149, 426–428
- >>= (greater than signs, equal) shift right assignment operator, 148–149
- () (parentheses)
 - for code readability, 113–114
 - grouping operands and operators, 113–114
 - guidelines, 114
- % (percent sign) mod operator, 111–112, 426–428
- %= (percent sign, equal) mod assignment operator, 428
- + (plus sign)
 - addition operator, overloading, 426–428
 - arithmetic binary operator, 111–112
 - with char type data, 115
 - concatenating strings, 114–115
 - delegate operator, 583–584
 - determining distance between two characters, 116
 - with non-numeric operands, 114–115
 - precedence, 112
 - unary operator, 110–111
 - unary operator, overloading, 428–430
- += (plus sign, equal)
 - addition assignment operator, 428
 - delegate operator, 583–584
- ++ (plus signs) increment operator
 - C++ vs. C#, 124
 - increment, in a loop, 121–124
 - description, 121
 - guidelines, 124
 - lock statement, 125
 - post-increment operator, 122–124
 - postfix increment operator, 123–124
 - pre-increment operator, 123–124
 - prefix increment operator, 123–124
 - race conditions, 125
 - thread safety, 125
 - unary operator, 428–430
- ? (question mark) nullable modifier, 80–83, 492
- ?? (question marks) null-coalescing operator, 143–144
- ~ (tilde), bitwise complement operator, 153
 - unary operator, 428–430
- _ (underscore)
 - as digit separator, 50, 72–73

- in identifier names, 14
- line continuation character (Visual Basic), 18
- in variable names, 22
- tuple item discard, 87
- __ (two underscores), in keyword names, 15
- | (vertical bar) bitwise OR operator, 149–152, 151, 405–407, 426–428
- |= (vertical bar, equal sign) bitwise OR assignment operator, 152–153
- || (vertical bars) logical OR operator, 139–140, 428
- \ (single backslash character), character escape sequence, 55

A

- Abort(), 786–787
- Aborting threads, 786–787
- Abstract classes. *See also* Derivation.
 - defining, 338–340
 - definition, 338
 - derived from System.Object, 344–345
 - vs. interfaces, 376
 - polymorphism, 341–343
- Abstract members
 - defining, 338–340
 - definition, 338
 - "is a" relationships, 341
 - overriding, 341
 - virtual, 341
- Access modifiers. *See also* Encapsulation.
 - circumventing with reflection, 934
 - on getters and setters, 274–276
 - purpose of, 260
 - types of, 259
- Action delegates, 542–544
- Add()
 - appending items to lists, 685
 - inserting dictionary elements, 691–692
 - System.Threading.Interlocked class, 876–878
 - thread synchronization, 877
- add_OnTemperatureChange(), 599–600
- Addresses. *See* Pointers and addresses.
- Aggregate functions, 643–644
- AggregateException, 590, 803–807
- AggregateException.Flatten(), 822
- AggregateException.Handle(), 806, 822
- Aggregation
 - derivation, 323–325
 - interfaces, 371–372
 - multiple inheritance, interfaces, 371–372
- Aliasing
 - namespaces, 199–200. *See also* using directive.
 - types, 199–200
- AllocExecutionBlock(), 905
- AllowMultiple member, 746
- Alternative control flow statements, 130
- Ampersand, equal sign (&=) bitwise AND assignment operator, 152–153
- Ampersand (&) bitwise AND operator, 149–152, 151
- Ampersands (&&) logical AND operator, 140
 - overloading, 428
- AND operator, & (ampersand), bitwise AND operator 149–152, 151, 405–406
- AND operator, && (ampersands), logical AND operator, 140, 428
- Angle brackets (<>), in XML, 31
- Angle struct, 395–396
- Anonymous functions
 - definition, 551–552
 - guidelines, 566
 - type, 555
- Anonymous methods. *See also* Lambda expressions.
 - definition, 538, 551, 556
 - guidelines, 557
 - internals, 560–561
 - parameterless, 557–558
 - passing, 557
 - type association, 555
- Anonymous type arrays, initializing, 653–654
- Anonymous types. *See also* Implicit local variables; Tuples.
 - collection initializers, 653–654
 - description, 82–83
 - drawbacks, 652
 - generating, 652–653
 - history of, 646
 - immutability, 650–652
 - with LINQ, 649–650
 - in query expressions, 662

- Anonymous types (*continued*)
 - vs. tuples, 652
 - type safety, 650–652
- Antecedent tasks, 797–798
- Apartment-threading models, 894–895
- APIs (application programming interfaces)
 - calls from P/Invoke, wrappers, 909
 - definition, 38
 - deprecated, 751–752
 - as frameworks, 38
- Append(), 67
- AppendFormat(), 67
- Appending items to collections, 685
- Applicable method calls, 224
- Applications, compiling, 9
- Appointment, 315–316
- __arglist keyword, 15
- ArgumentException, 468
- ArgumentNullException, 467, 468
- ArgumentOutOfRangeException, 467, 468
- Arguments
 - calling methods, 183, 187
 - named, calling methods, 222
- Arity (number of type parameters), 503–504
- Array accessor, 97
- Array declaration
 - C++ vs. C#, 92
 - code examples, 92, 93, 94
 - description, 91
 - Java vs. C#, 92
- Array instance methods, 102–103
- Array types, constraint limitations, 516
- ArrayList type, 391–394
- Arrays. *See also* Collections; Lists; TicTacToe game.
 - accessing, 92, 97–98
 - anonymous type, initializing, 653–654
 - of arrays. *See* Jagged arrays.
 - assigning, 91, 93–97
 - binary search, 100–102
 - BinarySearch(), 100–102
 - buffer overruns, 99
 - changing the number of items in, 102
 - Clear(), 100–102
 - clearing, 100–102
 - cloning, 103
 - common errors, 105–106
 - converting collections to, 683
 - copying, 103
 - declaring, 91–93
 - default keyword, 91
 - default values, 95
 - defining array literals, 93
 - description, 90–92
 - designating individual items, 90
 - exceeding the bounds of, 99–100
 - GetLength(), 102–103
 - indexers, defining, 702–703
 - indexes, 90
 - instantiating, 93–95
 - iterating over, 607–608
 - jagged, 97, 98, 100
 - length, getting, 99–100, 102
 - length, specifying, 95
 - Length member, 99–100
 - multidimensional, 91, 95–96, 98
 - null, 705
 - number of dimensions. *See* Multidimensional arrays.
 - number of items, getting, 99
 - overstepping the bounds of, 99
 - palindromes, 103–104
 - rank, 92–93
 - redimensioning, 102
 - Reverse(), 104–105
 - reversing, 104
 - reversing strings, 103–104
 - searching, 100–102, 688–689
 - sorting, 100–102
 - strings as, 103–105
 - swapping data elements, 98
 - three-dimensional, 96–97
 - ToCharArray(), 104
 - two-dimensional, 93, 96, 98. *See also* TicTacToe game.
 - type defaults, 91
 - unsafe covariance, 530–531
 - zero items, 705
- as operator, 349–350
- AsParallel(), 620
- ASP.NET, 927
- AspNetSynchronizationContext, 842
- Assemblies, compiling, 9
- Assembly, definition, 9
- assembly attributes, 736–738
- Assert(), 118
- Assignment operator. *See* Equal sign (=) assignment operator.
- Association, 249–251, 292

- Associativity of operators, 112–117
 - Asterisk, equal sign (=) multiplication
 - assignment operator, 428
 - Asterisk (*) multiplication operator, 111–112, 426–428
 - async keyword
 - misconceptions about, 826
 - purpose of, 826, 828
 - task-based asynchronous pattern, 823–828
 - Windows UI, 842–844
 - async methods, return of `ValueTask<T>`, 828–830
 - Asynchronous continuations, 796–803
 - Asynchronous delays, 785
 - Asynchronous high-latency operations with the TPL, 819–823
 - Asynchronous lambdas, 833–835
 - Asynchronous methods
 - custom, implementing, 835–838
 - returning void from, 830–833
 - Asynchronous operations, 776, 781–783
 - Asynchronous tasks. *See* Multithreading, asynchronous tasks.
 - AsyncState, 795
 - At sign (@)
 - coding verbatim string literals, 58
 - inserting literal backslashes, 58
 - keyword prefix for keyword identifiers, 15
 - Atomic operations, threading problems, 778
 - Atomicity of reading and writing to variables, 867
 - AttachedToParent enum, 799
 - Attributes
 - adding encryption, 755–756
 - adding metadata about assemblies, 736–737
 - alias command-line options, 740
 - AllowMultiple member, 746
 - assembly, 736–737
 - checking for, 740
 - CIL for, 758
 - class, 738
 - custom, 738–739
 - custom, defining, 738–739
 - custom, retrieving, 739–740, 742
 - custom serialization, 755–756
 - decorating properties with, 735–736
 - definition, 721
 - deserializing objects, 755–756
 - duplicate names, 746–747
 - guidelines, 738, 740, 744, 746
 - initializing with a constructor, 740–744
 - vs. interfaces, 377
 - looking for, 740–741
 - method, 738
 - module, 738
 - named parameters, 746–747
 - naming conventions, 738
 - no-oping a call, 749–750
 - Parse(), 748
 - predefined, 748–749
 - pseudoattributes, 758
 - retrieving, 740–741
 - return, 737–738
 - serialization-related, 748–749, 752–754
 - setting bits or fields in metadata tables. *See* Pseudoattributes.
 - uses for, 735
 - warning about deprecated APIs, 751–752
 - AttributeUsageAttribute
 - decorating custom attributes, 745–747
 - predefined attributes, 748–749
 - Automatically implemented properties
 - description, 265–267
 - initializing, 266–267
 - internals, 276–278
 - NextId implementation, 296
 - read-only, 272, 303–304
 - Average(), 644
 - await keyword
 - misconceptions about, 826
 - non-Task<T> or values, 838–840
 - task-based asynchronous pattern, 823–828
 - Windows UI, 842–844
 - await operators
 - with catch or finally statements, 846
 - description, 844–846
 - multithreading with
 - System.Threading.Thread class, 785
- B**
- Backslashes (\), as literals, 58
 - Base classes
 - inheritance, 326
 - vs. interfaces, 354–355

- Base classes, overriding. *See also* Derivation.
 - accessing a base member, 336–337
 - base keyword, 336
 - brittle base class, 331–334
 - constructors, 337
 - fragile base class, 331–334
 - introduction, 326
 - new modifier, 330–334
 - override keyword, 327, 336–337
 - sealed modifier, 335
 - sealing virtual members, 335
 - virtual methods, 326–330
 - virtual modifier, 326–330
- base keyword, 336
- Base members, accessing, 336–337
- Base type, 244
- BCL (Base Class Library), 33, 927, 929, 943
- Binary digits, definition, 147
- Binary display, string representation of, 151
- Binary floating-point types, precision, 116
- Binary literals, 51
- Binary operators, 426–428
- Binary search of arrays, 100–102
- BinaryExpression, 568
- BinarySearch()
 - bitwise complement of, 688
 - searching a list, 688–689
 - searching arrays, 100–102
- BinaryTree<T>, 506–508, 704, 707–708
- Bits, definition, 147
- Bitwise complement of BinarySearch(), 688
- Bitwise complement operator, tilde (~), 153
- Bitwise operators
 - & (ampersand) AND operator, 149–152, 151, 405–406
 - &= (ampersand, equal sign) bitwise AND assignment operator, 152–153
 - ^ (caret) XOR operator, 140–141, 149–152
 - ^= (caret, equal sign) bitwise XOR assignment operator, 152–153
 - << (less than signs), shift left operator, 148–149
 - <<= (less than, equal signs), shift left assignment operator, 148–149
 - ~ (tilde), bitwise complement operator, 153
 - | (vertical bar) bitwise OR operator, 149–152, 151, 405–407
 - |= (vertical line, equal sign) bitwise OR assignment operator, 152–153
 - >> (greater than signs), shift right operator, 148–149
 - >>= (greater than, equal signs), shift right assignment operator, 148–149
- binary digits, definition, 147
- bits, definition, 147
- bytes, definition, 147
- introduction, 147–148
- logical operators, 149–152
- masks, 151
- multiplication and division with bit shifting, 149
- negative numbers vs. positive, 148
- shift operators, 148–149
- string representation of a binary display, 151
- two's complement notation, 148
- Block statements, 133. *See also* Code blocks.
- BlockingCollection<T>, 888
- bool (Boolean) types
 - description, 53–54
 - returning from lambda expressions, 552, 558
- Boolean expressions. *See also* Bitwise operators.
 - == (equal signs) equality operator, 138–139
 - != (exclamation point, equal sign) inequality operator, 138–139
 - < (less than sign), less than operator, 138
 - <= (less than, equal sign), less than or equal operator, 138
 - > (greater than sign), greater than operator, 138
 - >= (greater than, equal sign), greater than or equal operator, 138
- definition, 137
- equality operators, 138–139
- evaluating. *See* if statements.
- example, 137–138
- in if statements, 130
- relational operators, 138–139

- Boolean expressions, logical operators
 - . (dot) operator, 145
 - ?: (question mark, colon), conditional operator, 142–143
 - ?. (question mark, dot), null-conditional operator, 144–146
 - && (ampersands), logical AND operator, 140
 - ^ (caret), XOR operator, 140–141
 - ! (exclamation point), logical negation operator, 141
 - ?? (question marks), null-coalescing operator, 143–144
 - || (vertical lines), logical OR operator, 139–140
 - introduction, 139
 - Boolean values, replacing with enums, 400
 - Bounds of an array, overstepping, 99–100
 - Boxing
 - avoiding during method calls, 396–398
 - code examples, 391–392
 - introduction, 390–391
 - `InvalidCastException`, 393–394
 - performance, 393
 - subtle problems, 393–396
 - synchronizing code, 394–396
 - unboxing, 390–394
 - value types in the lock statement, 394–396
 - `Break()`, 856
 - break statement, 129, 165–166
 - Breaking parallel loop iterations, 855–856
 - Brittle base, 331–334
 - `BubbleSort()`, 538–547
 - byte type, 44
 - Bytes, definition, 147
- C**
- C language
 - pointer declaration, vs. C#, 914
 - similarities to C#, 2
 - C# language
 - case sensitivity, 2
 - compiler, 9
 - description, 943
 - C++ language vs. C#
 - = (assignment operator) vs. == (equality operator), 138
 - buffer overflow bugs, 99
 - declaring arrays, 102
 - delete operator, 248
 - deterministic destruction, 460, 933
 - explicit deterministic resource cleanup, 248
 - garbage collection, 932–934
 - global methods, 190
 - global variables and functions, 289
 - header files, 194
 - implicit deterministic resource cleanup, 248
 - implicit nondeterministic resource cleanup, 248
 - implicit overriding, 327
 - implicitly typed variables, 82
 - increment/decrement operators, 124
 - local variable scope, 137
 - `main()`, 17
 - method calls during construction, 330
 - multiple inheritance, 323
 - operator-only statements, 111
 - operator order of precedence, 124
 - order of operations, 114
 - partial methods, 194
 - pointer declaration, 914
 - preprocessing, 171
 - pure virtual functions, 341
 - similarities, 2
 - string concatenation at compile time, 59
 - switch statement fall-through, 164
 - `void*`, 82
 - void type, 68
 - Caching data in class collections, 626
 - `Calculate()`, 793
 - Call site, 203
 - Call stack, 202–203
 - Callback function, 537
 - Caller, 183
 - `CallerMemberName` parameter, 734
 - Calling
 - constructors, 279, 285–286
 - methods. *See* Methods, calling.
 - object initializers, 281–282
 - camelCase
 - tuple names, 86–87
 - variable names, 22
 - `Cancel()`, 812
 - Canceling
 - parallel loop iterations, 852–854
 - PLINQ queries, 859–861
 - tasks. *See* Multithreading, canceling tasks.

- CancellationToken property, 810–814, 855
- CancellationTokenSource property, 812, 854
- CancellationTokenSource.Cancel(), 812
- Capacity(), 683–686
- Captured variables, 561–563
- Capturing loop variables, 564–566
- Caret, equal sign (^=) bitwise XOR assignment operator, 152–153
- Caret (^) bitwise XOR operator, 140–141, 149–152, 426–428
- Cartesian products, 634, 675
- Casing
 - formats for identifiers, 14
 - local variables, 22
- Cast operator
 - defining, 319–320
 - definition, 69
 - overloading, 430–431
- Casting
 - between arrays of enums, 402–403
 - between base and derived types, 317–318
 - definition, 69
 - explicit cast, 69–70, 317–318
 - implicit conversion, 317–318
 - with inheritance chains, 318
 - inside generic methods, 523–524
 - type conversion without, 73–74
- Catch(), 472
- Catch blocks
 - catching different exception types, 469–471
 - description, 228–232
 - general, 234–235, 473–475
 - internals, 475
 - with no type parameter, 234–235
- catch clause, 803
- catch statements, await operators, 846
- Catching exceptions
 - catch blocks, 469–471
 - code sample, 227–228
 - conditional clauses, 470–471
 - definition, 227
 - description, 227–232, 469
 - different exception types, 469–471
 - exception conditions, 470–471
 - general catch blocks, 473–475
 - rethrowing existing exceptions, 471
 - switch statements, 469
 - when clauses, 470
- Central processing unit (CPU), definition. *See* CPU (central processing unit).
- Chaining
 - constructors, 285–286
 - inheritance, 316
 - multicast delegates, 586–587
 - tasks, 797–798
- Changing strings, 65–67
- char (character) types, 21, 54
- Checked block example, 71
- Checked conversions, 70–72
- Checking for null
 - guidelines, 582
 - multicast delegates, 580–582
- Chess game, declaring an array for, 93
- Child type, 244
- Church, Alonzo, 558
- CIL (Common Intermediate Language). *See also* CLI (Common Language Infrastructure).
 - compiling C# source code into, 32
 - compiling into machine code, 924, 930
 - CTS (Common Type System), 32, 939–940
 - custom attributes, 942
 - description, 943
 - disassembling, tools for, 34. *See also* ILDASM.
 - ILDASM, 34
 - late binding, 942
 - managed execution, 32–33
 - metadata, 941–942
 - objects, 940
 - sample output, 35–37
 - source language support, 939, 940–941
 - type structure, 32, 939–940
 - values, 940
- CIL disassembler. *See* ILDASM.
- class attributes, 738
- Class collections. *See also* IEnumerable interface; IEnumerable<T> interface.
 - cleaning up after iteration, 611–612
 - error handling, 612
 - iterating over using while(), 609
 - resource cleanup, 612
 - sharing state, 610

- Class collections, foreach loops
 - with arrays, 607–608
 - code example, 611–612
 - with IEnumerable interface, 612
 - with IEnumerable<T> interface, 608–610
 - modifying collections during, 613
- Class collections, sorting. *See also*
 - Standard query operators, sorting.
 - ascending order ThenBy(), 626–628
 - ascending order with OrderBy(), 626–628
 - descending order with OrderByDescending(), 628
 - descending order with ThenByDescending(), 628
- Class definition
 - definition, 15
 - guidelines, 15
 - naming conventions, 15
 - syntax, 15
- Class hierarchy, 244
- class keyword, 511
- Class libraries
 - adding NuGet packages, 436–439
 - definition, 432–433
- Class libraries, referencing
 - with Dotnet CLI, 434–436
 - with Visual Studio 2017, 435–436
- Class members, definition, 248
- Class type
 - combining with class or struct, 515
 - constraints, 509–510
- class vs. struct, 715
- Classes
 - abstract. *See* Abstract classes.
 - adding instance methods, 299
 - association, 249–251, 292
 - association with methods, 183
 - base. *See* Base classes.
 - within classes. *See* Nested, classes.
 - declaring, 245–247
 - definition, 246–247
 - derived. *See* Derivation.
 - fields, 249–251
 - guidelines, 246
 - identifying support for generics, 731–732
 - inextensible, 298
 - instance fields, 249–251
 - instance methods, 251–252
 - instances of. *See* Objects.
 - instantiating, 245–247
 - vs. interfaces, 375–376
 - member variables, 249–251
 - nested, 304–306
 - partial, 307–308
 - polymorphism, 245
 - private members, 260
 - refactoring, 314–315
 - sealed, 325
 - spanning multiple files, Java vs. C#, 9. *See also* Partial methods.
 - splitting across multiple files. *See* Partial methods.
 - static, 297–298
 - uses for, 243
- Clear(), 100–102, 173
- Clearing arrays, 100–102
- CLI (Common Language Infrastructure). *See also* CIL (Common Intermediate Language); VES (Virtual Execution System).
 - assemblies, 936–938
 - compilers, 925–928
 - contents of, 925
 - definition, 924
 - description, 924–925, 943
 - implementations, 925–928
 - managed execution, 32–33
 - manifests, 936–938
 - modules, 936–938
 - xcopy deployment, 938
- Clone(), 103
- Cloning arrays, 103
- Closed over variables, 561–563
- Closures, 564
- CLR (Common Language Runtime), 944. *See also* Runtime.
- CLS (Common Language Specification), 940–941
 - description, 944
 - managed execution, 33
- CLU language, 706
- Code. *See* CPU (central processing unit).
- Code access security, 33
- Code blocks, 132–135
- Code readability
 - vs. brevity, 189
 - improving with whitespace, 19–20
 - matching caller variables with parameter names, 204
 - TAP language pattern, 844

- Code safety. *See* Unsafe code.
- Coding the publish-subscribe pattern
 - with multicast delegates
 - checking for null, 580–582
 - connecting publisher with subscribers, 578–579
 - defining subscriber methods, 576–577
 - defining the publisher, 578
 - delegate operators, 583–584
 - getting a list of subscribers, 590
 - guidelines, checking for null, 582
 - invoking a delegate, 579–580
 - method returns, 590
 - multicast delegate internals, 586–587
 - new delegate instances, 582
 - passing by reference, 590
 - removing delegates from a chain, 583
 - sequential invocation, 584–586
 - thread safe delegate invocation, 582
- Cold tasks, 792
- Collect(), 450
- Collection classes
 - dictionary collections, 690–695
 - linked list collections, 701
 - list collections, 682–686
 - queue collections, 700
 - sorted collections, 697–698
 - sorting lists, 686–687
 - stack collections, 699–700
- Collection initializers
 - with anonymous types, 653–654
 - basic requirements, 605–606
 - definition, 604
 - description, 282–283
 - for dictionaries, 606
 - initializing anonymous type arrays, 653–654
 - initializing collections, 605
- Collection interfaces, customizing
 - appending items to, 685
 - comparing dictionary keys, 695–696
 - converting to arrays, 683
 - counting collection elements, 683
 - dictionary class vs. list, 680–683
 - finding even elements, 690
 - finding multiple items, 689–690
 - generic hierarchy, 681
 - inserting new elements, 691–693
 - lists vs. dictionaries, 680–683
 - order of elements, 687–688
 - removing elements, 686
 - search element not found, 689
 - searching arrays, 688–689
 - searching collections, 688–689
 - specifying an indexer, 682
- Collection interfaces with standard query operators
 - caching data, 626
 - counting elements with Count(), 621–622
 - deferred execution, 622–626
 - definition, 613
 - filtering with Where(), 616–617, 622–626
 - guidelines, 622
 - projecting with Select(), 618–619
 - queryable extensions, 644–645
 - race conditions, 620–621
 - running LINQ queries in parallel, 620–621
 - sample classes, 614–616
 - sequence diagram, 625
 - table of, 644
- Collections. *See also* Anonymous types; Arrays; Class collections; Lists.
 - discarding duplicate members, 675–676
 - empty, 705
 - filtering, 658
 - projecting, 658
 - returning distinct members, 675–676
- Collections, customizing
 - accessing elements without modifying the stack, 699
 - appending items to, 685
 - counting elements of, 683
 - empty, 705
 - FIFO (first in, first out), 700
 - finding even elements, 690
 - finding multiple items, 689–690
 - indexers, defining, 702–703
 - inserting new elements, 691–693, 699
 - LIFO (last in, first out), 699–700
 - order of elements, 687–688
 - removing elements, 686
 - requirements for equality comparisons, 695–696
 - search element not found, 689
 - searching, 688–689
- Collections, iterating over with while, 609

- Collections, sorting. *See also* Standard query operators, sorting.
 - by file size, 669–670
 - by key, 697–698
 - with query expressions, 668–669
 - by value, 697–698
- Collections of collections, join operations, 640–641
- COM DLL registration, 939
- COM threading model, controlling, 894–895
- Combine()
 - combining delegates, 584
 - event internals, 600
 - parameter arrays, 212–214
 - vs. Swap() method, 206
- CommandLine, 304–306
- CommandLineAliasAttribute, 740–741
- CommandLineInfo, 725–730, 735
- CommandLineSwitchRequiredAttribute, 738–739
- Comments
 - vs. clear code, 30
 - delimited, 30
 - guidelines, 31
 - multi-line, 173
 - overview, 28–31
 - preprocessor directives as, 173
 - single-line, 30
 - types of, 30
 - XML delimited, 30
 - XML single-line, 30
- Common Intermediate Language (CIL). *See* CIL (Common Intermediate Language).
- The Common Language Infrastructure Annotated Standard*, 32
- Common Language Infrastructure (CLI). *See* CLI (Common Language Infrastructure).
- Common Language Runtime (CLR), 944. *See also* Runtime.
- Common Language Specification (CLS). *See* CLS (Common Language Specification).
- Common Type System (CTS), 32
- Compare(), 54, 537
- CompareExchange(), 876–877
- CompareExchange<T>, 876–877
- CompareTo(), 509, 686–687
- Comparing
 - dictionary keys, 695–696
 - for equality, float type, 117–120
- Comparison operators, 424–425
- ComparisonHandler delegate, 545
- Compatible method calls, 224
- Compile(), 568
- Compilers
 - AOT (ahead of time), compilation, 942–943
 - C# compilation to machine code, 929–931
 - C# language, 9
 - DotGNU Portable NET, 926
 - JIT (just-in-time) compiler, 930
- Compiling
 - applications, 3
 - assemblies, 9
 - into CIL, 32
 - jitting, 930
 - just-in-time, 32, 928
 - NGEN tool, 930
- Complex memory models, threading problems, 779–780
- Composite formatting, 27–28
- Compress(), 354–355
- Concat(), 643
- Concatenating strings, 114–115
- Concrete classes, 338, 341
- Concurrent collection classes, 888–889
- Concurrent operations, definition, 776
- ConcurrentBag<T>, 888
- ConcurrentDictionary<T>, 889
- ConcurrentQueue<T>, 888
- ConcurrentStack<T>, 888
- Conditional
 - clauses, catching exceptions, 470–471
 - expressions, guidelines, 143
 - logical operators, overloading, 428
- ConditionalAttribute, 749–751
- Conditions, 130
- ConnectionState vs. ConnectionStates, 400
- Consequence statements, 130
- Console input, 24–25
- Console output
 - comments, types of, 30
 - composite formatting, 27
 - format items, 27–28
 - format strings, 27

- Console output (*continued*)
 - formatting with string interpolation, 26
 - with `System.Console.Write()`, 26–28
- `ConsoleListControl`, 355–360, 364
- `const` field, encapsulation, 301
- `const` keyword, 125
- Constant expressions, 125
- Constant locals, 125
- Constants
 - declaring, 126
 - definition, 125
 - guidelines, 125
 - vs. variables, guidelines, 125
- Constraints on type parameters. *See also*
 - Contravariance; Covariance.
 - class type constraints, 510–511
 - constructor constraints, 512–513
 - generic methods, 514, 522–523
 - inheritance, 513–515
 - interface type constraints, 510
 - introduction, 505–508
 - listing, 508
 - multiple constraints, 512
 - non-nullable value types, 511–512
 - reference types, 511–512
- Constraints on type parameters, limitations
 - array types, 516–517
 - combining class type with `class`, 515
 - combining class type with `struct`, 515
 - on constructors, 517–519
 - delegate types, 516–517
 - enumerated types, 516–517
 - operator constraints, 515–516
 - OR criteria, 516
 - restricting inheritance, 514
 - sealed types, 516–517
- Construction initializers, 285–286
- Constructor constraints, 512–513
- Constructors
 - calling, 279
 - calling one from another, 285–286
 - centralizing initialization, 286–287
 - chaining, 285–286
 - collection initializers, 282–283
 - constraints, 517–519
 - declaring, 278–280
 - default, 280–281
 - exception propagation from, 460
 - expression-bodied member
 - implementation, 285
 - finalizers, 283
 - in generic types, declaring, 501
 - guidelines, 285
 - initializing attributes with, 740–744
 - introduction, 278
 - new operator, 279, 280
 - object initializers, 281–282
 - overloading, 283–285
 - overriding base classes, 337
 - static, 294–296
- `Contains()`, 688–689, 699–700
- `ContainsKey()`, 693
- `ContainsValue()`, 693
- Context switch, definition, 776
- Context switching, 777
- Contextual keywords, 13, 718
- Continuation clauses, query expressions, 673–674
- Continuation tasks, 796–803
- `continue` statement
 - description, 167–169
 - guidelines, 163
 - syntax, 128
- `ContinueWith()`, 796–798, 806–807, 821
- Contracts vs. inheritance, 368–369
- Contravariance
 - definition, 528
 - delegates, 559
 - enabling with `in` modifier, 528–530
- Control flow. *See also* Flow control.
 - guidelines, 158, 159
 - misconceptions, 826
 - task continuation, 796–803
 - within tasks, 827
- Control flow statements. *See also specific statements.*
 - alternative statements, 130
 - block statements. *See* Code blocks.
 - Boolean expressions, evaluating. *See* `if` statement.
 - `break`, 129
 - code blocks, 132–135. *See also* Scope.
 - combining. *See* Code blocks.
 - conditions, 130
 - consequence, 130
 - `continue`, 128
 - declaration spaces, 135–136

- definition, 109
 - do while, 128, 153–156
 - for, 128, 156–159
 - foreach, 128, 159–161
 - goto, 129
 - if, 127, 130, 168–169
 - if/else, examples, 130, 132
 - indentation, 134
 - nested if, 130–132
 - scopes, 135–136. *See also* Code blocks.
 - switch, 129, 161–165
 - true/false evaluation, 130
 - while, 127, 153–156
 - Conversion operator
 - consequences of using, 431–432
 - overloading, 430–432
 - Converting
 - collections to arrays, 683
 - enums to and from strings, 403–404
 - between interfaces and implementing types, 366
 - types. *See* Types, conversions between.
 - Cooler objects, 576–577
 - Cooperative cancellation, definition, 810
 - Coordinate, 430–431
 - Copying
 - arrays, 103
 - Copy(), 294, 299–300
 - DirectoryInfoExtension.Copy(), 294, 299–300
 - reference types, 383
 - CopyTo(), 683
 - CoreCLR, 925
 - Count(), 621–622, 644
 - Count property, 622, 683
 - CountdownEvent, 888
 - Counting
 - class collection elements with Count(), 621–622
 - collection elements, 683
 - lines within a file, example, 215–220
 - CountLines(), 183
 - Covariance
 - definition, 524–525
 - delegates, 559
 - enabling with out modifier, 526–528
 - guidelines, 531
 - introduction, 524–525
 - preventing, 525
 - type safety, 530–531
 - unsafe covariance in arrays, 530–531
 - Covariant conversion
 - definition, 524–525
 - restrictions, 527–528
 - .cpp file, C++ vs. C#, 194
 - CPU (central processing unit), 774
 - Create(), 504–505
 - Create() factory, 763–764
 - .cs file extension, 8
 - CTS (Common Type System), 32, 939–940, 944
 - Curly braces ({})
 - in the body of constructs, 17
 - defining array literals, 93
 - formatting code, 20
 - forming code blocks, 133
 - in methods, 16
 - omitting, 134–135
 - string interpolation, 27, 59
 - as string literals, 63
 - in switch statements, 18
 - CurrentTemperature property, 578–579
 - Custom asynchronous methods, 835–838
 - Custom dynamic objects, 766–769
 - Custom serialization attributes, 755–756
 - Customizing
 - collection interfaces. *See* Collection interfaces, customizing.
 - collections. *See* Collections, customizing.
 - events, 600–601
 - IEnumerable interface. *See* Iterators.
 - IEnumerable<T> interface. *See* Iterators.
 - IEnumerator<T> interface. *See* Iterators.
 - Customizing, exceptions
 - defining, 479–481
 - guidelines, 481
 - InnerException property, 481
 - serializable exceptions, 482
- ## D
- Data
 - on interfaces, 355
 - retrieval from files, 257–259
 - Data persistence, 256–257
 - Data types. *See* Types.
 - DataStorage, 256–259

- Deadlocks. *See also* Thread synchronization.
 - avoiding, 879–880
 - causes of, 880
 - non-reentrant locks, 880
 - prerequisites for, 880
 - thread synchronization, 881
 - threading problems, 780
- Deallocating memory, finalizers, 454
- Debugging with preprocessor directives, 173
- decimal type, 46–47
- Declaration spaces, 135–136
- Declaring
 - arrays, 92–93
 - classes, 245–247
 - constants, 126
 - constructors, 278–280
 - delegate types, 542–545
 - events, 593–594
 - finalizers, 453–454
 - generic classes, 496
 - instance fields, 249–250
 - local variables, 20–21
 - methods. *See* Methods, declaring.
 - properties, 262–265
- Deconstruct(), 287–289
- Deconstructors, 287–289
- Decrement(), 125, 877–878
- Decrement operator. *See* Increment/decrement operators (++, --).
- Default constructors, 280–281
- default keyword, 95
- default operator, 388–389
- Default types, 48
- default(bool) keyword, 95
- DefaultIfEmpty(), 639–640
- default(int) keyword, 95
- Deferred execution
 - implementing, 667
 - query expressions, 663–666
 - standard query operators, 622–626
- #define preprocessor directive, 172–174
- Delay(), 785, 893–894
- Delaying code execution. *See* Thread synchronization, timers.
- delegate keyword, 542, 586
- Delegate operators, 583–584. *See also* specific operators.
- Delegate types, 516–517, 540–545
- Delegates
 - = (minus sign, equal sign), delegate operator, 583–584
 - BubbleSort() example, 538–547
 - contravariance, 559
 - covariance, 559
 - creating with a statement lambda, 551–554
 - deferred execution, 667
 - definition, 537, 540
 - executing unsafe code, 920–922
 - vs. expression trees, 570–571
 - general purpose, 542–544
 - guidelines, 557
 - immutability, 547
 - instantiating, 545–547
 - internals, 547–550
 - invocation sequence diagram, 585
 - mapping to function pointers, 910
 - method group conversion, 546
 - method groups, 546
 - method names as arguments, 546
 - multicast, 575, 580
 - nesting, 545
 - with the null-conditional operator, 146–147
 - passing with expression lambdas, 554
 - structural equality, 558–560
 - synchronous, 791
 - System.Action, 542–544
 - System.Func, 542–544
 - vs. tasks, 791
 - thread safety, 582
- delete operator, C++ vs. C#, 248
- Delimited comments, 30
- DenyChildAttach enum, 799
- Deprecated APIs, 751–752
- Dequeue(), 700
- Dereferencing pointers, 917–919
- Derivation
 - abstract classes. *See* Abstract classes.
 - aggregation, 323–325
 - casting between base and derived types, 317–318
 - casting with inheritance chains, 318
 - classes derived from System.Object, 344–345
 - data conversion with the as operator, 349–350

- defining custom conversions, 319–320
- determining the underlying type, 345–346
- explicit cast, 317–318
- extension methods, 322–323
- implicit conversion, 317–318
- inheritance chains, 316
- "is a" relationships, 317–318
- is operator, 345–346
- multiple inheritance, simulating, 323–325
- overriding base classes. *See* Base classes, overriding.
- private access modifier, 319–320
- protected access modifier, 321–322
- refactoring a class, 314–315
- sealed classes, 325
- single inheritance, 323–325
- Derived types, 244–245
- Deserialize(), 754
- Deserializing
 - document objects, 754
 - documents, 756–758
 - objects, 755–756
- Deterministic finalization with the using statement, 454–457
- Diagnostics.ConditionalAttribute, 749–751
- Diagramming multiple inheritance, 373
- Dictionaries, indexers, 702–703
- Dictionary classes
 - customized collection sorting, 686–687
 - definition, 690–691
 - diagrams, 691
 - hash codes, 693, 696–697
 - inserting elements, 691
 - list collections, 683–686
 - vs. lists, 680–683
 - removing elements, 693
- Dictionary collection classes, 680–682
- Dictionary collections, 690–695
- Dictionary keys, comparing, 695–696
- Dictionary<>, initializing, 606
- Dictionary<TKey, TValue>, 606, 690–695
- Directives. *See* Preprocessor directives.
- DirectoryCountLines(), 216
- Directory.GetFiles(), 661, 669
- DirectoryInfoExtension.Copy(), 294, 299–300
- DirectoryInfo.GetFiles(), 299–300, 632
- DirectoryInfo.Move(), 299–300
- Disassembling CIL, tools for, 34. *See also* ILDASM; *specific tools.*
- Discarding duplicate collection members, 675–676
- Discards, 87
- DispatcherSynchronizationContext, 842
- Disposable tasks, 816
- Dispose(), 455–460
- Distinct(), 644, 675–677
- Dividing by zero, 119–120
- Division with bit shifting, 149
- DLL (dynamic link Library), 9
 - .dll file extension, 9
- do while loops, 128, 153–156
- Documentation
 - deserializing, 756–758
 - serializing, 756–758
 - tools for, 448–449
 - versioning, 756–758
 - XML, 448–449
- Dollar sign, at sign (\$@), string interpolation, 59–60
- Dollar sign (\$), string interpolation, 59
- Dot (.) operator, 145, 920
- DotGNU Portable NET, 926
- Dotnet CLI
 - dotnet.exe command, 4–5
 - referencing class libraries, 434–436
- Double quotes (" "), escape sequence for string literals, 58
- double type, precision, 116
- double.TryParse(), 238–239
- Dropping namespaces. *See* using directive.
- Dump(), 365
- Duplicate names for attributes, 746–747
- dynamic as System.Object, 763
- Dynamic binding, 764–765
- dynamic directive, 761–763
- Dynamic member invocation, 762
- dynamic principles and behaviors, 761–763
- dynamic type. *See* Programming with dynamic objects.

E

- E-mail domain, determining, 168
- Editors recommended for C#, 3. *See also*
 - Microsoft Visual Studio 2017;
 - Visual Studio Code.
- `#elif` preprocessor directive, 172–173
- Eliminating namespaces. *See* using directive.
- `else` clauses, 130
- `#else` preprocessor directive, 172–173
- Empty collections, 705
- `Empty<T>`, 705
- Encapsulation. *See also* Access modifiers.
 - circumventing with reflection, 934
 - `const` field, 301
 - definition, 248
 - description, 248
 - information hiding, 259–261
 - introduction, 243
 - public constants, permanent values, 302
 - read-only fields, 302–304
 - `readonly` modifier, 302–304
 - of types, 439
- Encryption for documents, 755–756
- `#endif` preprocessor directive, 172–173
- `#endregion` preprocessor directive, 176–178
- `Enqueue()`, 700
- `Enter()`, 394, 868–870, 874–875
- `EntityBase`, 510–511
- `EntityBase<T>`, 514
- `EntityDictionary`, 511–513
- `EntityDictionary<T>`, 510–511
- `EntityDictionary<TKey, TValue>`, 512–513
- Enumerated types, constraint limitations, 516–517
- Enums
 - casting between arrays of, 402–403
 - characteristics of, 399–400
 - conversion to and from strings, 403–404
 - defining, 400
 - definition, 399–400
 - as flags, 405–409
 - `FlagsAttribute`, 407–409
 - joining values, 405–407
 - replacing Boolean values, 400
 - type compatibility, 402–403
 - underlying type, 400
- Enums, guidelines
 - creating enums, 401
 - default type, 401
 - enum flags, 407
 - string conversions, 405
- Equal sign (=) assignment operator
 - vs. `==` (equality operator), C++ vs. C#, 138
 - assigning variables, 22
 - definition, 22
 - overloading, 424, 428
 - precedence, 112
- Equal sign, greater than (`=>`) lambda operator, 551, 554–555, 557
- Equal signs (`==`) equality operator
 - overloading, 424–425
 - in place of `=` (equal sign) assignment operator, 138
- Equal signs (`==`), true/false evaluator, 53
- Equality operators, 138
- `Equals()` equality operator
 - implementing, 420–423
 - overriding, 415–423
 - requirements for equality comparisons, 695–696
- `Equals()` method, overloading, 390
- Equi-joins, 638
- Error handling. *See also* Exception handling.
 - APIs, 903–905
 - class collections, 612
 - multicast delegates, 587–590
 - `P/Invoke`, 903–905
- Error messages, disabling/restoring, 175–176
- `#error` preprocessor directive, 172, 174–175
- Errors. *See also* Exception handling.
 - emitting with directives, 174–175
 - infinite recursion error, 217
 - reporting. *See* Exception handling.
- Escape sequences
 - `\` (single backslash character), 55
 - displaying a smiley face, 56
 - list of, 55–56
 - `\n`, newline character, 55, 56
 - `\t`, tab character, 55
 - for Unicode characters, 56
- `EssentialCSharp.sln` file, 10
- `Even()`, 690

- event keyword, 591, 593–594
- Event notification
 - with multiple threads, 878–879
 - thread-safe, 879
- Events
 - coding conventions, 594–596
 - customizing, 600–601
 - declaring, 593–594
 - encapsulating the publication, 592–593
 - encapsulating the subscription, 591–592
 - generics and delegates, 597–598
 - guidelines, 596, 598
 - internals, 598–600
 - registering listeners for, 802–803
- Exception class inheritance, 233
- Exception conditions, 470
- Exception handling. *See also* Errors; *specific exceptions.*
 - with `AggregateException`, parallel loop iterations, 851–852
 - appropriate use of, 237–238
 - basic procedures, 225–226
 - catch blocks, 228–232, 234–235
 - catching exceptions, 227–232
 - catching exceptions from `async void` methods, 830–833
 - common exception types, 232–233. *See also specific types.*
 - control flow, 229
 - examples, 225–228
 - for expected situations, 237–238
 - finally blocks, 228–232
 - general catch blocks, 234–235
 - guidelines, 232, 235, 237, 238, 468, 475–479
 - handling order, 230
 - Java vs. C#, 473
 - multiple exception types, 465–468
 - numbers as strings vs. integers, 226
 - numeric conversion, 238–239
 - `Parse()`, 238–239
 - reporting errors. *See* throw statement; Throwing exceptions.
 - task-based asynchronous pattern, 821–822
 - throwing exceptions, 226–232, 235–238. *See also* throw statements.
 - trapping errors, 226–232
 - try blocks, 228–232
 - `TryParse()`, 238–239
 - unhandled exceptions, 226–232
- Exception propagation from constructors, resource cleanup, 460
- `ExceptionDispatchInfo.Throw()`, 472
- Exceptions, custom. *See also* Errors.
 - defining, 479–481
 - guidelines, 481
 - serializable exceptions, 482
- `Exchange<T>`, 877
- Exclamation point (!)
 - logical NOT operator, 141
 - unary operator, overloading, 428–430
- Exclamation point, equal sign (!=)
 - inequality operator
 - overloading, 390, 424–425
 - testing for inequality, 138–139
- Exclamation point notation, 533
- Excluding/including code with preprocessor directives, 172–173
- `ExecuteSynchronously` enum, 800
- Execution time, definition, 33
- `Exit()`, 394, 868–870, 874–875
- Exiting a switch section, guidelines, 164
- Explicit cast, 69–70
- Explicit deterministic resource cleanup, C++ vs. C#, 248
- Explicit implementation of interfaces, 362–363, 364–365
- Explicit member implementation, 362–363
- Explicitly declared parameter types, 553
- Expression bodied methods, 194
- Expression lambdas, 554–556
- Expression trees
 - `BinaryExpression`, 568
 - building LINQ queries, 570–571
 - `Compile()`, 568
 - deferred execution, 667
 - definition, 538, 566
 - vs. delegates, 570–571
 - examining, 571–573
 - lambda expressions as data, 566–568
 - `LoopExpression`, 568
 - `MethodCallExpression`, 568
 - `NewExpression`, 568
 - as object graphs, 568–569
 - `ParameterExpression`, 568
 - `UnaryExpression`, 568
- Extensible Markup Language (XML). *See* XML.

- Extension methods
 - definition, 299
 - derivation, 322–323
 - inheritance, 322–323
 - on interfaces, 369–371
 - introduction, 299–300
 - reflection support for, 762–763
 - requirements, 300
- extern methods, 899
- External functions
 - calling with P/Invoke, 906–908
 - declaring with P/Invoke, 898–899
- F**
- F-reachable (finalization) queue, resource cleanup, 458
- Factory methods, generic types, 504–505
- FailFast(), 468, 468
- false, unary operator, overloading, 428–430
- Fat arrow notation. *See* Lambda operator.
- FCL (Framework Class Library), 929, 944
- Fibonacci calculator, 153–154
- Fibonacci numbers, 154
- Fibonacci series, 154
- Fields
 - accessing properties, 64
 - declaring as volatile, 875–876
 - getter/setter methods, 261–262
 - guidelines, 267–268
 - identifying owner of, 252–253
 - marking as private, 261–262
 - nonstatic. *See* Instance fields.
 - static, 289–292
 - virtual, properties as, 273–274
- FIFO (first in, first out), 700
- File extensions, 8. *See also specific extensions.*
- FileInfo collections, projecting, 669–670
- FileInfo object, 618–619
- FileInfo.Directory(), 632
- Filename matching class name, Java vs. C#, 9
- Files
 - data persistence, 256–257
 - data retrieval, 257–259
 - storage and loading, 256–259
- FileSettingsProvider, 369
- FileStream property, 461–463
- Filtering collections
 - definition, 658
 - filtering criteria, 667–668
 - predicates, 667–668
 - query expressions, 667–668
- Finalization
 - guidelines, 459–460
 - resource cleanup, 453–460
- Finalization (f-reachable) queue, resource cleanup, 458
- Finalizers
 - deallocating memory, 454
 - declaring, 453–454
 - description, 283, 453
 - deterministic finalization with the using statement, 454–457
- Finally blocks, 230–232
- finally statements, await operators, 846
- FindAll(), 689–690
- Finding
 - even elements in collections, 690
 - multiple items in collections, 689–690
- Fixed statement, 915–916
- Fixing (pinning) data, 915–916
- Flags, enums, 405–409
- FlagsAttribute
 - code sample, 747–748
 - custom serialization, 755–756
 - Deserialize(), 754
 - deserializing document objects, 754
 - effects on ToString() and Parse(), 407–409
 - ISerializable interface, 755–756
 - no-oping a call, 749
 - non-serializable fields, 754
 - predefined attributes, 748–749
 - serialization-related attributes, 748–749
 - serializing document objects, 754
 - System.Diagnostics
 - .ConditionalAttribute, 749–751
 - System.NonSerializable, 754
 - System.ObsoleteAttribute, 751–752
 - System.SerializableAttribute, 752–754
 - versioning serialization, 756–758
- Flatten(), 822
- Flattening a sequence of sequences, 674–675
- flNewProtect, 902

- float type
 - negative infinity, 119
 - negative zero, 120
 - overflowing bounds of, 119
 - positive infinity, 119
 - positive zero, 120
 - unexpected inequality, 117
- Floating-point types
 - binary float, 47
 - decimal, 46–47
 - double, 45–46
 - for financial calculations. *See* decimal type.
 - float, 45–46
- Flow control. *See also* Control flow.
 - definition, 774
 - introduction, 126–129
 - statements. *See* Control flow statements.
- for loops
 - CIL equivalent for, 611–612
 - description, 156–159
 - parallel, 867–868
- for statements, 128, 156–159
- foreach loops, class collections
 - code example, 611–612
 - with `IEnumerable` interface, 612
 - with `IEnumerable<T>` interface, 608–610
 - interleaving loops, 610
 - iterating over arrays, 607–608
 - modifying collections during, 613
- foreach statement, 128, 159–161
- `ForEach<T>()`, 854
- Formal declaration, methods. *See* Methods, declaring.
- Formal parameter declaration, 191
- Formal parameter list, 191
- `Format()`, 60
- Format items, 27
- Format strings, 27
- `FormatMessage()`, 903
- Formatting
 - numbers as hexadecimal, 52
 - with string interpolation, 26
 - strings, 63
- Forward slash (/) division operator
 - description, 111–112
 - overloading, 426–428
 - precedence, 112
- Forward slash (/) in XML, 31
- Forward slashes (///), XML comment delimiter, 447
- Fragile base, 331–334
- Framework Class Library (FCL), 929
- Frameworks, definition, 38
- from clause, 659–660, 674–675
- `FromCurrentSynchronizationContext()`, 840–842
- Full outer join, definition, 629
- Func delegates, 542–544
- Function pointers, mapping to delegates, 910
- Functions
 - global, C++ vs. C#, 289
 - pure virtual, 341
- G**
- Garbage collection
 - `Collect()`, 450
 - introduction, 449–450
 - managed execution, 33
 - in .NET, 450–451
 - object references, 210
 - resource cleanup, 458–460
 - root references, 450
 - strong references, 451
 - weak references, 451–452
- Garbage collector, definition, 248
- `GC.ReRegisterFinalize()`, 461
- General catch blocks, 234–235, 473–475
- General purpose delegates, 542–544
- Generic classes
 - declaring, 496
 - type parameters, 496
 - undo, with a generic Stack class, 494–496
- Generic delegates, events, 597–598
- Generic internals
 - CIL representation, 532–533
 - instantiating based on reference types, 534–535
 - instantiating based on value types, 533–534
 - introduction, 531–532
- Generic methods
 - casting inside, 523–524
 - constraints, 514
 - constraints on type parameters, 514–515
 - guidelines, 524
 - introduction, 519–520
 - specifying constraints, 522–523
 - type inference, 520–522

- Generic types
 - arity (number of type parameters), 503–504
 - benefits of, 497
 - constraints. *See* Constraints on type parameters.
 - constructors, declaring, 501
 - Create(), 504–505
 - factory methods, 504–505
 - finalizers, declaring, 501
 - generic classes, 494–497
 - guidelines, 506
 - implementing, 499–500
 - interfaces, description, 498–499
 - interfaces, implementing multiple versions, 499–500
 - introduction, 493–494
 - multiple type parameters, 502–503
 - nesting, 505
 - overloading a type definition, 504
 - parameter arrays, 505
 - parameterized types, 493
 - specifying default values, 501–502
 - structs, 498–499
 - System.ValueTuple class, 503–505
 - Tuple class, 503–505
 - type parameter naming guidelines, 498
 - ValueTuple class, 503–505
 - variadic, 505
- Generic types, generic classes
 - declaring, 496
 - type parameters, 496
 - undo, with a generic Stack class, 494–496
- Generic types, reflection on
 - classes, identifying support for
 - generics, 731–732
 - determining parameter types, 731–733
 - methods, identifying support for
 - generics, 731–732
 - type parameters for generic classes or methods, 731–732
 - typeof operator, 731
- Generics, C# without generics
 - multiple undo operations, 488–489
 - nullable value types, 492–493
 - System.Collections.Stack class, 488–491
 - type safety, 491
 - using value types, 491
- get_PropertyName, 65
- get keyword, 264
- GetCurrentProcess(), 899
- GetCustomAttributes(), 740–741
- GetDynamicMemberNames(), 769
- GetEnumerator(), 613–616, 707
- GetFiles(), 183, 299, 632, 661, 669
- GetFirstName(), 264
- GetFullName(), 189–194
- GetGenericArguments(), 732–733
- GetHashCode(), 389–390, 413–415, 696–697
- GetInitials(), 189–192
- GetInvocationList(), 590
- GetLastError API, 903
- GetLastName(), 264
- GetLength(), 102
- GetName(), 251–252
- GetProperties(), 723–724
- GetResponse(), 818
- GetResponseAsync(), 821
- GetReverseEnumerator(), 718–719
- GetSetting(), 367–369
- GetSummary(), 341
- GetSwitches(), 741–744
- Getter/setter methods, 261–262, 274–276
- Getter/setter properties, declaring, 262–265
- GetType(), 555, 723–724
- GetUserInput(), 189–194, 202
- GetValue(), 729
- GhostDoc tool, 449
- Global methods, C++ vs. C#, 190
- Global variables and functions, C++ vs. C#, 289
- goto statement, 129, 169–171
- Greater than, equal sign (>=), greater than or equal operator, 138, 424–425
- Greater than sign (>), greater than operator, 138, 424–425
- Greater than signs (>>), shift right operator, 148–149, 426–428
- Greater than signs, equal (>>=) shift right assignment operator, 148–149
- GreaterThan(), 547
- group by clause, 671
- group clause, 659
- GroupBy(), 636–637
- Grouping query results, 670–673
- GroupJoin(), 637–640

H

- .h file, C++ vs. C#, 194
- Handle(), 806, 822
- Hardcoding values, 48–50
- HasFlags(), 405
- Hash codes, dictionary classes, 693, 696–697
- Hash tables, balancing, 413–415
- Header files, C++ vs. C#, 194
- Heaps, 79–80
- Heater objects, 576–577
- HelloWorld program
 - assemblies, 9
 - compiling source code, 3–4, 4–5, 9
 - creating a project, 8–9
 - creating source code, 3–4, 4–5
 - declaring local variables, 20–21
 - with Dotnet CLI, 4–5, 10
 - editing source code, 3
 - executing, 4, 4–5, 9
 - getting started, 2–3
 - in an IDE (integrated development environment), 6–8
 - libraries, 8–9
 - .NET framework, choosing, 3
 - project files, 8–9
 - running source code, 3
 - with Visual Studio 2017, 6–8
- HelloWorld.dll file, 9
- Hexadecimal numbers
 - as binary literal numbers, 51
 - formatting numbers as, 52
 - notation, 50–51
 - specifying as values, 51
- HideScheduler enum, 800
- Hill climbing, 850–851
- Hot tasks, 792
- Hungarian notation, 14
- Hyper-Threading, definition, 774
- Hyphens (-), in identifier names, 14

I

- I/O-bound latency, definition, 772
- IAngle interface, 395–396
- IAngle.MoveTo interface, 395
- ICollection<T> interface, 682–683
- IComparable interface, 389, 686–687
- IComparable<string>.CompareTo(), 686–687
- IComparable<T>, 509, 686–687

- IComparer<T>, 686–687
- Id property, 795
- IDE (integrated development environment)
 - debugging support, 7
 - HelloWorld program, 6–8
- Identifier names
 - (hyphens), 14
 - _ (underscore), 14
- Identifiers
 - camelCase, 14
 - casing formats, 14
 - definition, 13
 - guidelines for, 14
 - keywords as, 15
 - naming conventions, 13–14
 - PascalCase, 14
 - syntax, 13–14
- IDictionary<TKey, TValue>, 680–682
- IDisposable interface
 - cleaning up after iterating, 611–612
 - resource cleanup, 455–460
 - Task support for, 816
- IDisposable.Dispose(), 455–460
- IDistributedSettingsProvider
 - interface, 374–375
- IEnumerable interface
 - class diagram, 608
 - CopyTo(), 683
 - Count(), 683
 - customizing. *See* Iterators.
 - foreach loops, 612
 - foreach loops, class collections, 612
- IEnumerable<T> interface
 - class diagram, 608
 - customizing. *See* Iterators.
 - foreach loops, 608–610
- IEnumerator<T> interface, customizing. *See* Iterators.
- IEqualityComparer<T> interface, 695–696
- if/else statements, examples, 130, 132
- #if preprocessor directive, 172–173
- if statements, 127, 130, 168–169
- IFileCompression interface, 354–355
- IFormattable interface, 389, 397–398, 512, 515
- IL. *See* CIL (Common Intermediate Language).
- IL Disassembler. *See* ILDASM.

- IL (intermediate language), 943
- ILDASM (IL Disassembler), 34. *See also*
 - Obfuscators.
- ICollection interface, 355–360, 370–371
- ICollection<T>, 680–682
- ILMerge.exe utility, 938
- Immutability
 - delegates, 547
 - strings, 65–67
 - value types, 385
- Immutable strings, 24
- Implicit conversion, 72–73
- Implicit deterministic resource cleanup,
 - C++ vs. C#, 248
- Implicit implementation of interfaces,
 - 363–365
- Implicit local variables. *See also*
 - Anonymous types.
 - declaring as anonymous, 647–648, 651
 - history of, 646
 - var type, 647–649, 651
- Implicit member implementation, inter-
 - faces, 363–365
- Implicit nondeterministic resource
 - cleanup, C++ vs. C#, 248
- Implicit overriding, Java vs. C#, 327
- Implicitly typed local variables, 69, 81–82
- Importing types from namespaces. *See*
 - using directive.
- Imports directive, 196
- in modifier, 528–530
- in type parameter, 528–530
- Increment(), 125, 877–878
- Increment/decrement operators (++ , --)
 - C++ vs. C#, 124
 - decrement, in a loop, 121–124
 - description, 121
 - lock statement, 125
 - post-increment operator, 122–124
 - postfix increment operator, 123–124
 - pre-increment operator, 123–124
 - prefix increment operator, 123–124
 - race conditions, 125
 - thread safety, 125
- Indentation, flow control statement, 134
- IndexerNameAttribute, 703
- Indexers
 - arrays, 702–703
 - defining, 702–704
 - dictionaries, 702–703
 - inserting new elements, 692–693
 - naming, 703
 - specifying, 682
- IndexOf(), 688–689
- Inextensible classes, 298
- Inference, type parameters in generic
 - methods, 521–522
- Infinite recursion error, 217
- Information hiding. *See* Encapsulation.
- Inheritance
 - base classes, 326
 - base type, 244
 - chaining, 316
 - child type, 244
 - constraint limitations, 515
 - constraints on type parameters,
 - 513–515
 - vs. contracts, 368–369
 - derived types, 244–245
 - extension methods, 322–323
 - interfaces, 366–369
 - introduction, 243–245
 - “is a kind of” relationships, 314
 - multiple, C++ vs. C#, 323
 - multiple, simulating, 323–325
 - parent type, 244
 - private members, 319–320
 - purpose of, 314
 - single, 323–325
 - specializing types, 245
 - subtypes, 244
 - super types, 244
 - value types, 389–390
- Initialize(), 268–269, 287
- Initializing. *See also* Collection
 - initializers.
 - anonymous type arrays, 653–654
 - collections, 605
 - collections that don’t support
 - ICollection<T>, 606
 - a Dictionary<>, 606
 - versions of, 606
- Initializing attributes with a constructor,
 - 740–744
- inner classes, Java, 307
- Inner join, 629, 632–635
- InnerException property, 481
- InnerExceptions property, 590, 806,
 - 822, 852
- Insert(), 67

- Inserting
 - elements in dictionary classes, 691
 - new elements in collections, 688–689
- Instance fields. *See also* Static, fields.
 - accessing, 250–251
 - declaring, 249–250
 - definition, 249–251
- Instance methods
 - adding to a class, 299
 - definition, 60
 - introduction, 251–252
- Instantiating
 - arrays, 93–97
 - classes, 245–247
 - delegates, 545–547
 - interfaces, 361
- Instantiation, 17, 247
- int (integer) type, 21, 44, 225
- Integer literals, determining type of, 49–50
- Integers, type for, 44–45
- Integral types, 115
- Interface type constraints, 510
- Interfaces
 - vs. abstract classes, 353–354, 376
 - aggregation, 371–372
 - vs. attributes, 377
 - vs. base classes, 354–355
 - vs. classes, 375–376
 - contracts vs. inheritance, 368–369
 - converting between interfaces and implementing types, 366
 - data, 355
 - defining, 355
 - deriving one from another, 366–369, 374–375
 - extension methods, 369–371
 - inheritance, 366–369
 - instantiating, 361
 - introduction, 354–355
 - method declarations in, 355
 - naming conventions, 355
 - polymorphism, 355–360
 - post-release changes, 374–375
 - purpose of, 354–355
 - value types, 389–390
 - versioning, 374–375
- Interfaces, generic types
 - description, 498–499
 - finalizers, declaring, 501
 - implementing multiple versions, 499–500
- Interfaces, implementing and using
 - accessing methods by name, 365
 - code example, 356–359
 - explicit implementation, 362–365
 - explicit member implementation, 362–363
 - guidelines, 364–365
 - implicit implementation, 363–365
 - implicit member implementation, 363–365
 - mechanism relationships, 365
 - overview, 354–355
 - semantic relationships, 365
- Interfaces, multiple inheritance
 - aggregation, 371–372
 - diagramming, 373
 - implementing, 369, 371–372
 - working around single inheritance, 371–372
- internal access modifiers on type
 - declarations, 439–440
- internal accessibility modifier, 441
- Interpolating strings, 59–62
- Intersect(), 644
- into keyword, 673–674
- IntPtr, 901
- InvalidAddressException, 479
- InvalidCastException, 394
- Invoke(), 580–582
- IObsolete interface, 377
- IOrderedEnumerable<T> interface, 627
- IPairInitializer<T> interface, 530
- IProducerConsumerCollection<T>, 889
- IQueryable<T> interface, 644–645
- IReadableSettingsProvider interface, 366–369
- IReadOnlyPair<T> interface, 526–528
- “Is a kind of” relationships, 314
- “Is a” relationships, 317–318, 341
- is operator
 - pattern matching, 346–347
 - verifying underlying types, 345–346
- IsAlive property, 784
- IsBackground property, 783
- IsCancellationRequested property, 812, 854
- IsCompleted property, 795, 856
- IsDefined(), 407
- ISerializable interface, 482, 755–756
- ISettingsProvider interface, 367–369, 374–375

- IsInvalid, 906
 - IsKeyword(), 664
 - Items property, 500
 - Iterating over
 - arrays, 607–608
 - class collections using
 - IEnumerable<T>. *See*
 - IEnumerable<T> interface.
 - class collections using while(), 609
 - a collection, using while, 609
 - maintaining state during, 610
 - properties of objects in a collection. *See* Reflection.
 - Iterators
 - canceling iteration, 715–716
 - contextual keywords, 718
 - creating your own, 705
 - defining, 706
 - functional description, 716–718
 - guidelines, 715
 - multiple in one class, 718–719
 - nested, 714–715
 - origins of, 705–706
 - recursive, 714–715
 - reserved keywords, 718
 - return statement, 708–709
 - returning values from, 707–709
 - state, 709–711
 - struct vs. class, 715
 - syntax, 707
 - yield break statement, 715–716
 - yield return statement, 708–709, 718
 - ITrace interface, 365
 - IWritableSettingsProvider interface, 369
- J**
- Jagged arrays
 - declaring, 98
 - definition, 97
 - getting the length of, 100
 - Java vs. C#
 - classes spanning multiple files, 9
 - declaring arrays, 102
 - exception specifiers, 473
 - filename matching class name, 9
 - generics, 535
 - implicit overriding, 327
 - implicitly typed variables, 82
 - importing namespaces with wildcards, 196
 - inner classes, 307
 - main(), 17
 - partial class, 9
 - similarities to C#, 2
 - var, 82
 - virtual methods by default, 326
 - JIT (just-in-time) compiler, 32, 930
 - Jitting, 32, 930. *See also* Compilers; Compiling; Just-in-time compilation.
 - Join(), 632–635, 783
 - Join operations. *See* Standard query operators, join operations.
 - Jump statements
 - break statement, 165–166
 - continue statement, 167–169
 - goto statement, 169–171
 - if statement, 168–169
 - Just-in-time compilation, 32
- K**
- KeyNotFoundException, 693
 - Keys property, 695
 - Keywords. *See also specific keywords.*
 - contextual, 13
 - definition, 11
 - as identifiers, 15
 - incompatibilities, 13
 - list of, 12
 - placement, 11
 - reserved, 13, 15
 - syntax, 11–13, 15
 - Kill(), 837
- L**
- Lambda calculus, 558
 - Lambda expressions. *See also* Anonymous methods.
 - captured variables, 561–563
 - capturing loop variables, 564–566
 - closed over variables, 561–563
 - closures, 564
 - as data, 566–568
 - definition, 538, 550–551
 - explicitly declared parameter types, 553
 - expression lambdas, 551
 - GetType(), 555
 - guidelines, 553
 - internals, 560–561
 - lifetime of captured variables, 563
 - name origin, 558

- notes and examples, 555–556
- outer variable CIL implementation, 563–564
- outer variables, 561–563
- predicate, definition, 555, 616
- returning a bool, 552, 558
- sequence of operations, 625
- statement lambdas, 551–554
- typeof() operator, 555
- Lambda operator
 - => (equal sign, greater than) lambda operator, 551, 554–555, 557
- Lambdas, asynchronous, 833–835
- Language contrast. *See specific languages.*
- Language interoperability, 33
- Last in, first out (LIFO), 699–700
- LastIndexOf(), 688–689
- Late binding, 942
- Latency
 - asynchronous high-latency operations with the TPL, 819–823
 - definition, 772
 - synchronous high-latency operations, 817–818
- Latitude, 430–431
- Lazy initialization, 461–462
- Lazy loading, 462–463
- LazyCancellation enum, 801
- Left-associative operators, 113
- Left outer join, definition, 629
- Length
 - arrays, 99–100
 - strings, 64–65
- Length member, 64–65, 99–100
- Less than, equal sign (<=) less than or equal operator, 138, 424–425
- Less than sign (<) less than operator, 138, 424–425
- Less than signs, equal (<=) shift left assignment operator, 148–149
- Less than signs (<<) shift left operator, 148–149, 426–428
- let clause, 669–670
- Libraries
 - class, 432–433
 - definition, 9, 432
 - file extension, 9
 - TPL (Task Parallel Library). *See Pseudoattributes.*
- Library implementation. *See CLS (Common Language Specification).*
- Lifetime of captured variables in lambda expressions, 563
- LIFO (last in, first out), 699–700
- Line-based, statements, Visual Basic vs C#, 18
- #line preprocessor directive, 172, 176
- Linked list collections, 701
- LinkedListNode<T>, 701
- LinkedList<T>, 701
- LINQ (Link Integrated Query)
 - anonymous types, 649–650
 - definition, 603
- LINQ queries
 - building with expression trees, 570–571
 - with query expressions. *See Query expressions with LINQ.*
 - running in parallel, 620–621
- Linq.ParallelEnumerable, 857–859
- Liskov, Barbara, 705–706
- List collections, 683–686
- Listeners, registering for events, 802–803
- Lists. *See also Collections.*
 - vs. dictionaries, 680–683
 - indexers, defining, 702–703
 - sorting, 686–687
- List<T>
 - covariance, 525
 - description, 683–686
- List<T>.Sort(), 686–687
- Literal values
 - case sensitivity, suffixes, 49–50
 - definition, 47–48
 - exponential notation, 50
 - specifying, 47–48
 - strings, 57
- Loading
 - files, 256–259
 - lazy, 462–463
- Local functions, 39, 540, 834–835
- Local variable scope, C++ vs. C#, 137
- Local variables
 - assigning values to, 22–23
 - casing, 22
 - changing values, 22–23
 - data types, specifying, 21
 - declaring, 20–21
 - guidelines for naming, 22
 - implicitly typed, 69, 81–82
 - naming conventions, 22, 191
 - unsynchronized, 867–868

- Localizing applications, Unicode
 - standard, 54
 - lock keyword, 870–872
 - lock objects, 873–874
 - Lock performance, locking on this
 - keyword, 874–875
 - lock statement, 394–396, 780
 - Lock synchronization, 870–872
 - Locking
 - guidelines, 875, 881
 - on this, `typeof`, and `string`, 874–875
 - threading problems, 780
 - `lockTaken` parameter, 870
 - Logical operators, 149–152. *See also* Boolean
 - expressions, logical operators.
 - Lollipops, in interface diagrams, 373
 - Long-running tasks, 815–816
 - long type, 44
 - Longitude, 430–431
 - `LongRunning` enum, 799
 - Loop variables, 156, 564–566
 - `LoopExpression`, 568
 - Loops
 - break statement, 129
 - continue statements, 128
 - with decrement operator, 121–124
 - do while, 128, 153–156
 - escaping, 165–166
 - for, 128, 156–159
 - foreach, 128, 159–161
 - goto, 129
 - guidelines, 158, 159
 - if, 127, 130
 - if/else, examples, 130, 132
 - iterations, definition, 155
 - nested if, 130–132
 - switch, 129
 - while, 127, 153–156
 - Loops, jump statements
 - break, 165–166
 - continue, 167–169
 - goto, 169–171
 - if, 168–169
 - `LowestBreakIteration`, 856
 - `lpfl0ldProtect`, 901
- M**
- Machine code, compiling, 924–925
 - `Main()`
 - activation frames, 202–203
 - `args` parameter, 17
 - call site, 203
 - call stack, 203
 - declaring, 16–17
 - definition, 16
 - disambiguating multiple `Main()`
 - methods, 202
 - invoking location, 203
 - multiple, disambiguating, 202
 - nonzero return code, 16–17
 - parameters, 200–202
 - passing command-line arguments to,
 - 200–202
 - returns from, 200–202
 - stack unwinding, 203
 - syntax, 16–17
 - `main()` method, Java vs. C#, 17
 - `__makeref` keyword, 15
 - `MakeValue()`, 512–513
 - Managed code, definition, 32
 - Managed execution
 - BCL (Base Class Library), 33
 - CIL (Common Intermediate Language), 32–34
 - CLI (Common Language Infrastructure) specification, 32–33
 - CLS (Common Language Specification), 33
 - code access security, 33
 - CTS (Common Type System), 32
 - definition, 31
 - execution time, 33
 - garbage collection, 33
 - just-in-time compilation, 32
 - language interoperability, 33
 - managed code, definition, 32
 - native code, definition, 32
 - platform portability, 33
 - runtime, definition, 32, 33
 - type safety, 33
 - unmanaged code, definition, 32
 - VES (Virtual Execution System), 32
 - Many-to-many relationships, definition, 629
 - Masks, 151
 - `Max()`, 297, 644
 - `MaxDegreeOfParallelism` property, 855
 - `Max<T>`, 519–520
 - `Me` keyword, 254
 - Member invocation, reflection, 725–730
 - Member names, retrieving, 769

- Member variables, 249–251
- MemberInfo, 729–730
- Memory, deallocating, 454. *See also* Garbage collector.
- Metadata
 - about assemblies, adding, 736–737
 - within an assembly, examining. *See* Reflection.
 - definition, 31
 - reflection, 942
 - for types, accessing with System.Type, 723–724. *See also* Reflection.
 - XML, 31
- Metadata tables, setting bits or fields in. *See* Pseudoattribute.
- method attributes, 738
- Method calls
 - avoiding boxing, 396–398
 - during construction, C++ vs. C#, 330
 - as ref or out parameter values, 276
 - vs. statements, 188–189
 - translating query expressions to, 676–678
- Method group conversion, delegates, 546
- Method names
 - as arguments, delegates, 546
 - calling, 187
- Method resolution, 224–225
- Method return type declaration, 192–194
- Method returns, multicast delegates, 590
- MethodCallExpression, 568
- MethodImplAttribute, 874
- MethodImplOptions.Synchronized(), 875
- Methods. *See also* Anonymous methods; *specific methods.*
 - accessing by name, on interfaces, 365
 - class association, 183
 - declaring in interfaces, 355
 - definition, 16, 182–183
 - derived from System.Object, 344–345
 - falling through, 192
 - global, C++ vs. C#, 190
 - guidelines for naming, 183
 - identifying support for generics, 731–732
 - instance, 251–252
 - naming conventions, 183
 - operational polymorphism, 218
 - overloading, 217–220
 - overriding, 328–330
 - partial, 308–311
 - passing as arguments, 557
 - return type declaration, 192–194
 - return values, 188
 - returning multiple values with tuples, 193–194
 - syntax, 16
 - uniqueness, 217–218
 - unreachable end point, 192
 - void, 193
- Methods, calling
 - applicable calls, 224
 - arguments, 183, 187
 - caller, 183
 - compatible calls, 224
 - method call example, 184
 - method name, 187
 - method resolution, 224–225
 - method return values, 188
 - named arguments, 222
 - namespaces, 184–186. *See also specific namespaces.*
 - recursively. *See* Recursion.
 - return values, 183
 - scope, 187
 - statements vs. method calls, 188–189
 - type name qualifier, 186–187
- Methods, declaring
 - example, 189–190
 - expression bodied methods, 194
 - formal parameter declaration, 191
 - formal parameter list, 191
 - method return type declaration, 192–194
 - refactoring, 190–191
 - return statements, example, 192–193
 - specifying no return value, 193
 - type parameter list, 191
 - void as a return type, 193
- Methods, extension
 - derivation, 322–323
 - on interfaces, 369–371
 - overview, 299–300
- Microsoft IL (MSIL), 943. *See also* CIL (Common Intermediate Language).
- Microsoft .NET Framework. *See* .NET Framework.
- Microsoft Silverlight, 926

- Microsoft Visual Studio 2017, 3
- Microsoft.CSharp.RuntimeBinder
 - .RuntimeBinderException, 762
- Min(), 297, 644
- Min(<T>), 519–520
- Minus sign (-)
 - arithmetic binary operator, 111–112
 - delegate operator, 583–584
 - precedence, 112
 - subtraction operator, overloading, 426–428
 - unary operator, 110–111
 - unary operator, overloading, 428–430
- Minus sign, equal (-=)
 - minus assignment operator, 428
 - delegate operator, 583–584
- Minus signs (--) decrement operator
 - C++ vs. C#, 124
 - decrement, in a loop, 121–124
 - description, 121
 - guidelines, 124
 - lock statement, 125
 - post-increment operator, 122
 - postfix increment operator, 123–124
 - pre-increment operator, 123–124
 - prefix increment operator, 123–124
 - race conditions, 125
 - thread safety, 125
- Minus signs (--) unary operator, overloading, 428–430
- Mod operator, percent sign (%), 111–112, 426–428
- module attributes, 738
- Monitor, 868–870
- Mono, 37, 926
- Montoya, Inigo, 2
- Move(), 299–300, 385
- MoveNext(), 710–711
- MSIL (Microsoft intermediate language), 943. *See also* CIL (Common Intermediate Language).
- MTA (Multithreaded Apartment), 895
- Multicast delegates
 - adding methods to, 586
 - chaining, 587
 - definition, 575, 580
 - error handling, 587–590
 - internals, 586–587
 - new delegate instances, 582
 - passing by reference, 590
 - removing delegates from a chain, 583
- Multidimensional arrays
 - assignment, 91
 - declaring, 91
 - indexes, 98
 - initializing, 95–96
- Multiple inheritance
 - C++ vs. C#, 323
 - simulating, 323–325
- Multiple inheritance, interfaces
 - aggregation, 371–372
 - diagramming, 373
 - implementing, 369, 371–372
 - working around single inheritance, 371–372
- Multiple Main() methods, disambiguating, 202
- Multiplication with bit shifting, 149
- Multithreaded Apartment (MTA), 895
- Multithreading
 - asynchronous operations, definition, 776
 - clock speeds over time, 772
 - concurrent operations, definition, 776
 - context switch, definition, 776
 - CPU (central processing unit), 774
 - flow of control, definition, 774
 - Hyper-Threading, definition, 774
 - I/O-bound latency, definition, 772
 - latency, definition, 772
 - multithreaded programs, definition, 774
 - parallel programming, definition, 776
 - performance guidelines, 778
 - PLINQ (Parallel LINQ). *See* Pseudoattributes.
 - PLINQ queries. *See* PLINQ queries, multithreading.
 - process, definition, 774
 - processor-bound latency, definition, 772
 - programs. *See* Pseudoattributes.
 - purpose of, 775–776
 - quantum, definition, 776
 - simultaneous multithreading, definition, 774
 - single-threaded programs, definition, 774
 - TAP (Task-based Asynchronous Pattern). *See* Pseudoattributes.
 - task, definition, 775
 - thread, definition, 774
 - thread pool, definition, 775

- thread safe code, definition, 774
- threading model, definition, 774
- time slice, definition, 776
- time slicing, definition, 776
- TPL (Task Parallel Library). *See* Pseudoattributes.
- Multithreading, asynchronous tasks
 - antecedent tasks, 797–798
 - associating data with tasks, 795
 - asynchronous continuations, 796–803
 - chaining tasks, 797–798
 - cold tasks, 792
 - composing large tasks from smaller one, 796–798
 - continuation tasks, 797–798
 - control flow, 796
 - creating threads and tasks, 790–791
 - hot tasks, 792
 - Id property, 795
 - introduction, 791–795
 - invoking, 791–795
 - multithreaded programming
 - complexities, 789–790
 - observing unhandled exceptions, 806–807
 - polling a `Task<T>`, 793–794
 - registering for notification of task behavior, 801–802
 - registering for unhandled exceptions, 803–807
 - registering listeners for events, 802–803
 - returning void from an asynchronous method, 830–833
 - synchronous delegates, 791
 - task continuation, 796–803
 - task identification, 795
 - task scheduler, 790–791
 - task status, getting, 794
 - `Task.Run()`, 792
 - tasks, definition, 790–791
 - tasks vs. delegates, 791
 - `Task.WaitAll()`, 793
 - `Task.WaitAny()`, 793
 - unhandled exception handling with `AggregateException`, 803–807
 - unhandled exceptions on a thread, 803–807
- Multithreading, canceling tasks
 - cooperative cancellation, definition, 810
 - disposable tasks, 816
 - long-running tasks, 810–814
 - obtaining a task, 814–815
- Multithreading, guidelines
 - aborting threads, 787
 - long-running tasks, 816
 - parallel loops, 849
 - performance, 777, 781
 - thread pooling, 789
 - `Thread.Sleep()`, 784–785
 - unhandled exceptions, 810
- Multithreading, parallel loop iterations
 - breaking, 855–856
 - canceling, 852–854
 - exception handling with `AggregateException`, 851–852
 - hill climbing, 850–851
 - introduction, 846–850
 - options, 854–855
 - TPL performance tuning, 850–851
 - work stealing, 850–851
- Multithreading, performance
 - context switching, 777
 - overview, 776–777
 - switching overhead, 777
 - time slicing costs, 777
- Multithreading, task-based asynchronous pattern
 - with `async` and `await`, 824–829
 - `async` and `await` with the Windows UI, 843–845
 - `async` keyword, purpose of, 827
 - `async void` method, 830–833
 - asynchronous high-latency operations
 - with the TPL, 820–824
 - asynchronous lambdas, 833–835
 - `await` keyword, 838–840
 - `await` operators, 844–846
 - awaiting non-`Task<T>` or values, 838–840
 - control flow misconceptions, 826
 - control flow within tasks, 827
 - custom asynchronous methods, 835–838
 - handling exceptions, 821–822
 - problems addressed by, 846
 - progress update, 837–838
 - synchronization context, 840–842
 - synchronous high-latency operations, 817–818
 - task drawbacks, overview, 817–823
 - task schedulers, 840–842

Multithreading, threading problems
 atomic operations, 778
 complex memory models, 779–780
 deadlocks, 780
 lock statement, 780
 locking leading to deadlocks, 780
 race conditions, 778–779

Multithreading, with
 System.Threading.Thread
 Abort(), 786–787
 aborting threads, 786–787
 asynchronous delays, 785
 asynchronous operations, 781–783
 await operator, 785
 checking threads for life, 784
 foreground threads vs. background, 783
 IsAlive property, 784
 IsBackground property, 783
 Join(), 783
 Priority property, 784
 putting threads to sleep, 784–785
 reprioritizing threads, 783
 Task.Delay(), 785
 thread management, 783–784
 thread pooling, 787–789
 ThreadAbortException, 786–787
 Thread.Sleep() method, putting
 threads to sleep, 784–785
 ThreadState property, 784
 waiting for threads, 783

N

\n, newline character, 55, 57
 Name property, 273–274
 Named arguments, calling methods, 222
 Named parameters, attributes, 746–747
 nameof operator
 properties, 270–271, 733–734
 throwing exceptions, 467, 468
 Namespaces
 aliasing, 199–200. *See also* using
 directive.
 calling methods, 184–186
 in the CLR (Common Language
 Runtime), 442
 common, list of, 185–186
 defining, 442–445
 definition, 184, 195
 dropping. *See* using directive.
 eliminating. *See* using directive.
 guidelines, 186, 445

importing types from. *See* using
 directive.
 introduction, 442–445
 naming conventions, 186, 442
 nested, 196, 443–444

Naming conventions
 avoiding ambiguity, 254
 class definition, 15
 identifiers, 13–14
 for interfaces, 355
 local variables, 22
 methods, 183
 namespaces, 186, 442
 parameters, 191, 223
 properties, 267
 type parameter, 498
 Native code, definition, 32
 NDoc tool, 449
 Negation operator. *See* NOT operator.
 Negative infinity, 119
 Negative numbers vs. positive, bitwise
 operators, 148
 Negative zero, 120
 Nested
 classes, 304–306
 delegates, 545
 generic types, 505–506
 if statements, 130–132
 iterators, 714–715
 namespaces, 196, 443–444
 types, 306
 using directives, 197–198
 .NET
 description, 943
 garbage collection, 450–451, 933–934
 .NET Compact Framework, 926
 .NET Core
 default .NET framework, 3
 description, 37, 925, 927–928
 .NET Framework
 description, 925
 vs. .NET framework, 927
 .NET framework
 choosing, 3
 default, 3. *See also* .NET Core.
 downloading, 3
 installing, 3
 vs. .NET Framework, 927
 predominant implementations, 37.
See also specific frameworks.
 .NET Micro Framework, 926

- .NET Native, 942–943
 - .NET standard, 928–929
 - .NET versions, mapped to C# releases, 39–40
 - new keyword, 94
 - New line, starting
 - /n (newline character), 55, 57, 64
 - /r/n (newline character), 64
 - strings, 55, 57
 - verbatim string literals, 57
 - WriteLine() method, 64
 - new modifier, 330–335
 - new operator
 - constructors, 279, 280
 - value types, 388
 - NewExpression, 568
 - NextId initialization, 295
 - No-oping a call, 749–750
 - Non-nullable value types, 511–512
 - Non-serializable fields, 754
 - NonSerializable, 754
 - Normalized data, 633
 - NOT operator, 141
 - NotImplementedException, 233
 - NotOnCanceled enum, 800
 - NotOnFaulted enum, 799
 - NotOnRanToCompletion enum, 799
 - nowarn option, 175–176
 - nowarn:<warn list> option, 175–176
 - NuGet packages, adding to class libraries, 436–439
 - null, checking for
 - arrays, 705
 - collections, 705
 - empty arrays or collections, 705
 - guidelines, 582
 - invoking delegates, 594
 - multicast delegates, 580–582
 - Null-conditional operator
 - delegates, 146–147
 - question mark, dot (?.), 144–146
 - short circuiting with, 144–146
 - null type
 - description, 67–68
 - use for, 67–68
 - Nullable modifier, 80–83
 - Nullable<T>, 493
 - NullReferenceException
 - invoking delegates, 594
 - throwing exceptions, 467, 468
 - Numbers, formatting as hexadecimal, 52
 - Numeric conversion, exception handling, 238–239
- ## O
- Obfuscators, 34. *See also* ILDASM (IL Disassembler).
 - Object graphs, expression trees as, 568–569
 - Object initializers
 - calling, 281–282
 - constructors, 281–282
 - definition, 281–282
 - object members, overriding, 416–419
 - Object-oriented programming,
 - definition, 242–243
 - Object types. *See* Types.
 - Objects. *See also* Constructors.
 - associations, 292
 - CIL (Common Intermediate Language), 940
 - definition, 246–247
 - destroying, 283
 - identity vs. equal object values, 416–419
 - instantiation, 247
 - Obsolete APIs. *See* Deprecated APIs.
 - ObsoleteAttribute, 751–752
 - Obtaining a task, 814–815
 - OfType<T>(), 643
 - One-to-many relationships, 630, 637–638
 - OnFirstNameChanging(), 310–311
 - OnLastNameChanging(), 310–311
 - OnlyOnCanceled enum, 799
 - OnlyOnFaulted enum, 800
 - OnlyOnRanToCompletion enum, 800
 - OnTemperatureChange event, 598–599
 - OnTemperatureChanged(), 577–578
 - Operational polymorphism, 218
 - OperationCanceledException, 814, 854, 859–861
 - Operator constraints, constraint limitations, 515–516
 - Operator-only statements, C++ vs. C#, 111
 - Operator order of precedence, C++ vs. C#, 124
 - Operators. *See also specific operators.*
 - arithmetic binary (+, -, *, /, %), 111–112
 - assignment (=), 22
 - associativity (()) parenthesis, 112–117
 - bitwise (&, |, ^), 149–152)
 - characters in arithmetic operations, 115

Operators (*continued*)

- comparison, 424–425
 - compound mathematical assignment
 - (+=, -=, *=, /=, %=), 120
 - compound bitwise assignment (&=, |=, ^=, >>=, <<=), 152
 - const keyword, 125
 - constant expressions, 125
 - constant locals, 125
 - definition, 109
 - left-associative, 113
 - operands, 110
 - operator-only statements, C++ vs. C#, 111
 - order of operations, 112, 114
 - plus and minus unary (+, -), 110–111
 - precedence, 112–114, 178–179
 - results, 110
 - right-associative, 113
 - uses for, 110
- Operators, increment/decrement (++ , --)
- C++ vs. C#, 124
 - decrement, in a loop, 121–124
 - description, 121
 - guidelines, 124
 - lock statement, 125
 - post-increment/post-decrement operator, 122
 - postfix increment/decrement operator, 123–124
 - pre-increment/pre-decrement operator, 123–124
 - prefix increment/decrement operator, 123–124
 - race conditions, 125
 - thread safety, 125
- Optional parameters, 220–224
- OR criteria, constraint limitations, 516
- OR operator, | (vertical bar), 405–407
- Order of operations, 112, 114. *See also* Precedence.
- OrderBy(), 626–629
- orderby clause, 668–669
- OrderByDescending(), 628
- out parameter, 206–209, 276, 526–528
- Out property, 740
- out vs. pointers, P/Invoke, 901–902
- Outer joins, 639–642
- Outer variables, lambda expressions, 561–563
- OutOfMemoryException, 468, 476

Output, passing parameters, 206–209

Overloading

- constructors, 283–285
 - equality operators on value types, 390
 - methods, 217–220
 - type definitions, 504
 - type definitions with tuples, 503–504
- Overloading operators. *See also specific operators.*
- = (equal sign) assignment operator, 423
 - binary operators, 426–428
 - binary operators combined with assignment operators, 428
 - cast operator, 430–431
 - comparison operators, 424–425
 - conditional logical operators, 428
 - conversion operators, 430
 - unary operators, 428–430
- override keyword, 327, 336–337

Overriding

- abstract members, 341
 - base classes. *See* Base classes, overriding.
 - base classes, virtual methods, 326–330
 - implicit, C++ vs. C#, 327
 - methods, 328–330
 - properties, 326–327
- Overriding object members
- Equals() equality operator, 415–423
 - GetHashCode(), 413–415
 - ToString(), 412–413

P

P/Invoke

- allocating virtual memory, 900
- calling external functions, 906–908
- declaring external functions, 898–899
- declaring types from unmanaged structs, 902–903
- description, 898
- error handling, 903–905
- function pointers map to delegates, 910
- guidelines, 905, 910
- out vs. pointers, 901–902
- parameter types, 899–901
- ref vs. pointers, 901–902
- SafeHandle, 905–906
- sequential layout, 902–903
- Win32 error handling, 903–905
- wrappers for API calls, 909

- PairInitializer<T> interface, 530
- Pair<T>, 525
- Palindromes, 103–104
- Parallel LINQ (PLINQ) queries,
 - multithreading. *See* PLINQ queries, multithreading.
- Parallel loop iterations. *See* Multithreading, parallel loop iterations.
- Parallel programming, definition, 776
- Parallel.For() loops, 855–856
- Parallel.ForEach() loops, 856
- Parallel.ForEach<T>(), 854
- ParallelOptions object, 855
- ParallelQuery<T>, 858, 861
- Parameter arrays, 212–214, 505
- Parameter names, identifying when
 - throwing exceptions, 467, 468
- Parameter types
 - determining, 731–732
 - explicitly declared, 553
 - P/Invoke, 899–901
- ParameterExpression, 568
- Parameterized types, 493
- Parameters
 - calling methods, 183, 187
 - guidelines, 191, 222–223
 - on the Main(), 200–202
 - matching caller variables with
 - parameter names, 203–204
 - method overloads, 222
 - names, generating. *See* nameof operator.
 - names, getting. *See* nameof operator.
 - naming conventions, 191, 223
 - optional, 220–224
 - reference types vs. value types, 204–205
 - specifying by name, example, 222–223
- Parameters, passing
 - out, 206–209
 - output, 206–209
 - ref type, 205–206
 - by reference, 205–206
 - by value, 203–204
- ParamName property, 468
- params keyword, 212–214
- Parent type, 244
- Parentheses (())
 - for code readability, 113–114
 - grouping operands and operators, 113–114
 - guidelines, 113
- Parse(), 73–76, 238–239, 404, 407–409, 748
- Partial classes, 9, 307–308
- Partial methods, C++ vs. C#, 194
- PascalCase
 - definition, 14
 - tuple names, 86–87
- Passing
 - anonymous methods, 557
 - command-line arguments to Main(), 200–202
 - delegates with expression lambdas, 554–555
 - methods as arguments, 557
 - by reference, multicast delegates, 590
- Passing, parameters
 - out, 206–209
 - output, 206–209
 - ref type, 205–206
 - by reference, 205–206
 - by value, 203–204
- Passing variable parameter lists, 212–213
- Pattern matching
 - with the is operator, 346–347
 - with a switch statement, 347–349
- Peek(), 699–700
- Percent sign, equal (=) mod assignment operator, 428
- Percent sign (%) mod operator, 111–112, 426–428
- Performance
 - effects of boxing, 393
 - locks. *See* Lock performance.
 - multithreading, 776–777, 780
 - runtime, 935–936
 - TPL (Task Parallel Library), 850–851
- Periods (...), download progress indicator, 821
- Pi, calculating, 846–850
- PiCalculator.Calculate(), 793
- PingButton_Click(), 843
- Ping.Send(), 843
- Platform interoperability. *See* P/Invoke; Unsafe code.
- Platform invoke. *See* P/Invoke.
- Platform portability, managed execution, 33
- PLINQ queries, multithreading
 - AggregateException, 861
 - canceling, 859–861
 - CancellationToken, 861

- PLINQ queries, multithreading (*continued*)
 - CancellationTokenSource, 861
 - introduction, 856–859
 - Linq.ParallelEnumerable, 857–859
 - OperationCanceledException, 815, 854, 859–861
 - ParallelQuery<T> object, 861
 - with query expressions, 858–859
 - TaskCancelledException, 861
- Plus sign (+)
 - addition operator, overloading, 426–428
 - arithmetic binary operator, 111–112
 - with char type data, 115
 - concatenating strings, 114
 - delegate operator, 583–584
 - determining distance between two characters, 116
 - with non-numeric operands, 114
 - precedence, 112
 - unary operator, 110–111
- Plus sign, equal (=)
 - plus assignment operator, 428
 - delegate operator, 583–584
- Plus sign (+), unary operator, overloading, 428–430
- Plus signs (++) increment operator
 - C++ vs. C#, 124
 - decrement, in a loop, 121–124
 - description, 121
 - guidelines, 124
 - lock statement, 125
 - post-increment operator, 122
 - postfix increment operator, 123–124
 - pre-increment operator, 123–124
 - prefix increment operator, 123–124
 - race conditions, 125
 - thread safety, 125
- Plus signs (++) unary operator, overloading, 428–430
- Pointers and addresses
 - accessing members of a referent type, 920
 - allocating data on the call stack, 917
 - assigning pointers, 914–915
 - dereferencing pointers, 917–919
 - fixing (pinning) data, 915–916
 - pointer declaration, 913–917
 - referent types, 913
 - unmanaged types, 913
 - unsafe code, 911–913
- Polling a Task<T>, 793–794
- Polymorphism. *See also* Inheritance; Interfaces.
 - abstract classes, 341–343
 - description, 245
 - interfaces, 355–360
- Pop(), 488–491, 699–700
- Positive infinity, 119
- Positive numbers vs. negative, bitwise operators, 148
- Positive zero, 120
- #pragma preprocessor directive, 172, 175–176
- Precedence, 112–117, 178–179. *See also* Order of operations.
- Precision
 - binary floating-point types, 116
 - double type, 116
 - float type, 117–120
- Predefined attributes, 748–749
- Predefined types, 43
- Predicates
 - definition, 555, 616
 - filtering class collections, 616
 - lambda expressions, 555, 667–668
- PreferFairness enum, 799
- Preprocessing, C++ vs. C#, 171
- Preprocessor directives. *See also* Control flow; Flow control.
 - as comments, 173
 - as debugging tool, 173
 - defining preprocessor symbols, 173–174
 - disabling/restoring warning messages, 175–176
 - emitting errors and warnings, 174–175
 - excluding/including code, 172–173
 - handling differences among platforms, 173
 - specifying line numbers, 176
 - summary of, 171–172. *See also specific directives.*
 - undefining preprocessor symbols, 173–174
 - visual code editors, 176–178
- Preprocessor symbols
 - defining with preprocessor directives, 173–174
 - undefining with preprocessor directives, 173–174

- The Princess Bride*, 2
 - `Print()`, 342
 - Priority property, 784
 - private access modifier, 259–261, 319–320, 441
 - Private fields, 261–262
 - private keyword, 261
 - Private members
 - accessing, 320–321
 - definition, 260
 - inheritance, 320
 - private protected accessibility modifier, 441
 - Procedural programming. *See* Structured programming.
 - Process, definition, 774
 - `Process.Kill()`, 837
 - Processor-bound latency, definition, 772
 - Program, accessing static fields, 291
 - Programming, object-oriented definition, 242–243
 - Programming with dynamic objects
 - dynamic binding, 764–765
 - dynamic directive, 761–763
 - dynamic member invocation, 762
 - dynamic principles and behaviors, 761–763
 - dynamic `System.Object`, 763
 - implementing a custom dynamic object, 766–769
 - introduction, 759
 - invoking reflection with dynamic, 759–761
 - reflection, support for extension methods, 762–763
 - retrieving member names, 769
 - signature verification, 762
 - vs. static compilation, 765–766
 - type conversion, 761–763
 - type safety, 760–761, 766
 - Progress update display, 837–838
 - Projecting collections
 - definition, 658
 - `FileInfo` collections, 669–670
 - with query expressions, 660–663
 - with `Select()`, 618–619
 - Properties
 - accessing with fields, 64
 - automatically implemented, 265–267, 272, 278, 296
 - automatically implemented, read-only, 303–304
 - declaring, 262–265
 - decorating with attributes, 735–736
 - definition, 262
 - getter and setter, 262–265
 - getting names of. *See* `CallerMemberName` parameter; `nameof` operator.
 - guidelines, 267–268, 272, 282
 - internal CIL code, 277–278
 - introduction, 261–262
 - `nameof` operator, 270–271, 733–734
 - naming conventions, 267
 - overriding, 326–327
 - read-only, 271–272
 - read-only automatically implemented, 303–304
 - as ref or out parameter values, 276
 - static, 296
 - validation, 268–270
 - as virtual fields, 273–274
 - write-only, 271–272
 - protected access modifier, 321–322, 441
 - protected internal accessibility modifier, 441
 - protected internal type modifier, 440–441
 - Protected members, accessing, 321–322
 - Pseudoattributes, 758
 - public access modifier, 259–261, 439–441
 - Publishing code, checking for null, 580–582
 - `Pulse()`, 870
 - Pure virtual functions, C++ vs. C#, 341
 - `Push()`, 488–491, 699–700
- ## Q
- Quantum, definition, 776
 - Query continuation clauses, 673–674
 - Query expressions with LINQ
 - anonymous types, 662
 - code example, 658–659
 - continuation clauses, 673–674
 - deferred execution, 663–667
 - definition, 657
 - discarding duplicate members, 675–676
 - filtering collections, 667–668
 - flattening a sequence of sequences, 674–675

- Query expressions with LINQ (*continued*)
 - from clause, 659–660
 - group by clause, 671
 - group clause, 659
 - grouping query results, 670–673
 - into keyword, 673–674
 - introduction, 658–660
 - let clause, 669–670
 - projecting collections, 660–663
 - range variables, 659
 - returning distinct members, 675–676
 - select clause, 659–660
 - sorting collections, 668–669
 - translating to method calls, 676–678
 - tuples, 662
 - where clause, 659–660
- Query operators. *See* Standard query operators.
- Question mark, colon (:) conditional operator, 142–143
- Question mark, dot (?.) null-conditional operator, 144–146
- Question mark (?) nullable modifier, 80–83, 492
- Question marks (??) null-coalescing operator, 143–144
- Queue collections, 700
- Queue<T>, 700

- R**
- Race conditions. *See also* Thread synchronization.
 - class collections, 620–621
 - threading problems, 778–779
- Range variables, 659
- Rank, arrays
 - declaring, 93
 - definition, 92
 - getting the length of, 99–100
- Read(), 25, 877
- Read-only
 - automatically implemented properties, 303–304
 - fields, encapsulation, 302–304
 - properties, 271–272
- Readability. *See* Code readability.
- ReadKey(), 25
- ReadLine(), 24–25
- readonly modifier
 - encapsulation, 302–304
 - guidelines, 304
- ReadToAsync(), 821
- ReadToEnd(), 818
- ReadToEndAsync(), 821
- Recursion. *See also* Methods, calling.
 - definition, 215
 - example, 215–217
 - infinite recursion error, 217
- Recursive iterators, 714–715
- ref parameter, properties as values, 276
- ref type parameters, passing, 205–206
- ref vs. pointers, 901–902
- Refactoring
 - classes, 314–315
 - methods, 190–191
- Reference
 - passing parameters by, 205–206
 - returning parameters by, 209–211
- Reference types
 - constraints on type parameters, 511–512
 - copying, 383
 - vs. value types, 204–206, 381–383
- ReferenceEquals(), 418
- Referencing other assemblies
 - changing the assembly target, 433
 - class libraries, 433–438
 - encapsulation of types, 439
 - internal access modifiers on type declarations, 439–440
 - protected internal type modifier, 440–441
 - public access modifiers on type declarations, 439–440
 - referencing assemblies, 434–436
 - type member accessibility modifiers, 441
- Referent types
 - accessing members of, 920
 - definition, 913
- Referential identity, 388
- Reflection
 - accessing using System.Type class, 723–724
 - circumventing encapsulation and access modifiers, 934
 - definition, 722
 - GetProperty(), 723–724
 - getting an object's public properties, 723–724
 - GetType(), 723–724
 - invoking with dynamic, 759–761

- member invocation, 725–730
- metadata, 942
- retrieving Type objects, 723–725
- support for extension methods, 762–763
- TryParse(), 729–730
- typeof(), 723–725
- uses for, 722
- Reflection, on generic types
 - classes, identifying support for
 - generics, 731–732
 - determining parameter types, 731–732
 - methods, identifying support for
 - generics, 731–732
 - type parameters for generic classes or methods, 732–733
 - typeof operator, 731
- __reftype keyword, 15
- __refvalue keyword, 15
- #region preprocessor directive, 172, 176–178
- Registering
 - listeners for events, 802–803
 - for notification of task behavior, 801–802
 - for unhandled exceptions, 808–810
- Relational operators, 138–139
- ReleaseHandle(), 906
- Remainder operator. *See* Mod operator.
- Remove(), 67
 - event internals, 600
 - removing delegates from chains, 584
 - removing dictionary elements, 693
 - System.Delegate, 686
- RemoveAt(), 686
- remove_OnTemperatureChange(), 599–600
- Removing
 - delegates from a chain, 583
 - dictionary elements, 693
 - elements from collections, 686
- Replace(), 67
- ReRegisterFinalize(), 461
- Reserved keywords, 13, 15, 718
- Reset(), 609
- Reset events, 884–887
- Resource cleanup. *See also* Garbage collection.
 - class collections, 612
 - exception propagation from constructors, 460
 - finalization, 453–460
 - finalization (f-reachable) queue, resource cleanup, 458
 - finalizers, 453–454
 - garbage collection, 458–460
 - guidelines, 459–460
 - with IDisposable, 455–460
 - invoking the using statement, 455–457
 - resurrecting objects, 461
- Result property, 794
- Results, 110
- Resurrecting objects, 461
- Rethrowing
 - existing exceptions, 471
 - wrapped exceptions, 483–484
- Retrieving
 - attributes, 740–741
 - member names, 769
 - Type objects, reflection, 723–725
- return attributes, 737–738
- return statements, 192–193, 708–709
- Return values
 - calling methods, 183
 - from iterators, 707–709
 - from Main(), 200–202
 - methods, 188
 - from the ReadLine(), 24–25
- Reverse(), 100–102, 104, 644
- Reversing
 - arrays, 104
 - strings, 104
- Right-associative operators, 113
- Right outer join, definition, 629
- Root references, 450
- Round-trip formatting, 52–53
- Run(), 792, 814–815
- RunContinuationAsynchronously
 - enum, 801
- Running applications, 3
- RunProcessAsync(), 835–837
- Runtime. *See also* VES (Virtual Execution System).
 - circumnavigation encapsulation and access modifiers, 934
 - definition, 32–33
 - garbage collection, 932–934
 - managed code, definition, 932
 - managed data, definition, 932
 - managed execution, definition, 932
 - performance, 935–936
 - platform portability, 935

- Runtime (*continued*)
 - reflection, 934, 942
 - type checking, 934
 - type safety, 934
- RuntimeBinderException, 762
- S**
- SafeHandle, 905–906
- Save(), 255–256
- sbyte type, 44
- Scope
 - calling methods, 187
 - flow control statements, 135–136
- SDK (software development kit), 3
- Sealed classes, 325
- sealed modifier, 335
- Sealed types constraint limitations, 516–517
- Sealing virtual members, 335
- Search element not found, 689
- Searching
 - arrays, 100–102, 688–689
 - collections, 688–689
 - lists, 688–689
- Select()
 - projecting class collection data, 618–619
 - vs. SelectMany(), 641
- select clause, 659–660, 667
- SelectMany()
 - calling, 640–641
 - creating outer joins, 639–642
 - vs. Select(), 641
- Semaphore, 887–888
- SemaphoreSlim, 887–888
- SemaphoreSlim.WaitAsync(), 887–888
- Semicolon (;), ending statements, 17–18
- Send(), 843
- SendTaskAsync(), 844
- SequenceEquals(), 644
- Sequential invocation, multicast delegates, 584–586
- Serializable objects, 482
- SerializableAttribute, 752–754
- Serialization(), 754
- Serialization, versioning, 756–758
- Serialization-related attributes, 748–749, 752–754
- Serializing
 - business objects into a database. *See* Reflection.
 - documents, 756–758
 - Serializing document objects, 754
 - set_PropertyName, 64
 - Set(), 884–887
 - set keyword, 264
 - SetName(), 252–253
 - SetResult(), 837
 - Setter properties. *See* Getter/setter properties.
 - Shared Source CLI, 926
 - Shift operators, 148–149
 - Short circuiting
 - with the null-conditional operator, 144–146
 - with the OR operator, 140
 - short types, 44, 45
 - SignalAndWait(), 883
 - Signature verification, 762
 - Simple assignment operator. *See* Equal sign (=) assignment operator.
 - Simultaneous multithreading, definition, 774
 - Single backslash character (\), escape sequence, 55
 - Single inheritance, 323–325, 371–372
 - Single-line comments, 30
 - Single-threaded programs, definition, 774
 - Slash, equal (/=) division assignment operator, 428
 - Sleep(), 784–785
 - Smiley face, displaying, 56
 - Sort(), 100–102, 686–687
 - SortedDictionary<T>, 697–698
 - SortedList<T>, 697–698
 - Sorting
 - in alphabetical order, 550
 - arrays, 100–102
 - class collections, 626–628. *See also* Standard query operators, sorting.
 - collections, 686–687, 697–698
 - integers in ascending order. *See* BubbleSort().
 - lists, 686–687
 - Sorting, collections. *See also* Standard query operators, sorting.
 - by file size, 669–670
 - by key, 697–698
 - with query expressions, 668–669
 - by value, 697–698

- Source code
 - compiling, 3–4, 4–5, 9. *See also*
 - Compilers.
 - creating, 3–4, 4–5
 - editing, 3
 - running, 3
 - used in this text, solution file, 10
- Specializing types, 245
- SqlException, 481
- Square brackets ([]), array declaration, 92
- Stack
 - allocating data on, 917
 - definition, 381
 - unwinding, 203
 - values, accessing, 699–700
- Stack, 494–496
- Stack collections, 699–700
- stackalloc data, 917
- Stack<int>, 533–534
- StackOverflowException, 233
- Stack<T>, 532–533, 699–700
- Standard query operators
 - AsParallel(), 620–621
 - caching data, 626
 - Concat(), 643
 - counting elements with Count(), 621–622
 - deferred execution, 622–626
 - definition, 613
 - Distinct(), 644
 - filtering with Where(), 616–617, 622–626
 - guidelines, 622, 678
 - Intersect(), 644
 - OfType(), 643
 - projecting with Select(), 618–619
 - queryable extensions, 644–645
 - race conditions, 620–621
 - Reverse(), 644
 - running LINQ queries in parallel, 620–621
 - sample classes, 614–616
 - sequence diagram, 625
 - SequenceEquals(), 644
 - sorting, 626–628
 - System.Linq.Enumerable method calls, 642–643
 - table of, 644
 - Union(), 643
- Standard query operators, join operations
 - Cartesian products, 634
 - collections of collections, 640–641
 - DefaultIfEmpty(), 639–640
 - equi-joins, 638
 - full outer join, 629
 - grouping results with GroupBy(), 636–637
 - inner join, 629, 632–635
 - introduction, 628–629
 - left outer join, 629
 - many-to-many relationships, 629
 - normalized data, 633
 - one-to-many relationships, 630
 - one-to-many relationships, with GroupJoin(), 637–638
 - outer joins, with GroupJoin(), 639–640
 - outer joins, with SelectMany(), 639–642
 - right outer join, 629
- Start(), 329–330, 803–804
- StartNew(), 814–815
- State
 - iterators, 709–711
 - sharing, class collections, 610
- Statement delimiters, 18–19
- Statement lambdas, 551–554
- Statements
 - combining. *See* Code blocks.
 - delimiters, 17–18
 - ending, 17–18
 - ending punctuation, 17–18
 - vs. method calls, 188–189
 - multiple, on the same line, 18, 20
 - splitting across multiple lines, 18–19
 - syntax, 18–19
 - without semicolons, 18
- STAThreadAttribute, 895
- Static
 - classes, 297–298
 - compilation vs. programming with dynamic objects, 765–766
 - constructors, 294–295
 - fields, 289–292. *See also* Instance fields; static keyword.
 - methods, 60–61, 293–294
 - properties, 296
 - static keyword, 289. *See also* Static, fields.

- Status property, 795
- Stop(), 329–330, 856
- Storage
 - disk, 700
 - files, 256–259
 - reclaiming. *See* Finalizers; Garbage collection.
- Store(), 256–259
- String interpolation
 - formatting with, 26
 - syntax prefixes, 59–60
- string keyword, 57, 874–875
- String methods
 - instance methods, 60
 - list of, 61–62
 - static methods, 60–61. *See also* using directive; using static directive.
- string type, 57, 226
- string.Compare(), 54
- string.Format(), 60
- string.join statement, 849
- Strings
 - @ (at sign), coding verbatim strings, 58
 - \$ (dollar sign), string interpolation, 59
 - \$@ (dollar sign, at sign), verbatim string interpolation, 59–60
 - "" (double quotes), coding string literals, 57–58
 - as arrays, 103–104
 - changing, 65–67
 - concatenation at compile time, vs. C++, 59
 - converting text to uppercase, 66
 - determining length of, 64–65
 - formatting, 63
 - having no value vs. empty, 68
 - immutability, 24, 65–67
 - interpolation, 59–62
 - length, 64–65
 - Length member, 64–65
 - literals, 57–59
 - \n, newline character, 55, 58
 - \r\n, newline character, 64
 - read-only properties, 65
 - representing a binary display, 151
 - setting to null, 67–68
 - starting a new line, 55, 57
 - type for, 57
 - verbatim string interpolation, 59–60
 - verbatim string literals, 58
 - void type, 68
- Strong references, garbage collection, 451–452
- struct keyword
 - vs. class, 715
 - constraints, 511–512
 - declaring a struct, 384
- StructLayoutAttribute, 902–903
- Structs
 - declaring, 384
 - default value for, 388–389
 - definition, 384
 - finalizer support, 388
 - generic types, 498–499
 - guidelines, 387
 - initializing, 385–387
 - referential identity, 388
- Structural equality, delegates, 558–560
- Structured programming, 181
- subscriber methods, 576–577
- Subtypes, 244
- Sum(), 644
- Super types, 244
- SuppressFinalize(), 458–460
- Surrogate pairs, 55
- Swap(), 206
- Swapping array data elements, 98
- switch statements
 - { } (curly braces), 18
 - catching exceptions, 469
 - code example, 161–162
 - fall-through, C++ vs. C#, 164
 - pattern matching, 347–349
 - replacing if statements, 163–164
 - syntax, 129
- Switching overhead, 777
- Synchronization. *See* Thread synchronization.
- Synchronization context, 840–842
- Synchronization types. *See* Thread synchronization, synchronization types.
- Synchronized(), 875
- Synchronizing
 - code, boxing, 394–396
 - local variables, 867–868
 - multiple threads with Monitor class, 868–870
 - threads. *See* Thread synchronization.

- Synchronous delegates, 791
- Synchronous high-latency operations, 817–818
- Syntax
 - class definition, 15
 - identifiers, 13–14
 - keywords, 11–13, 15
 - methods, 16–18
 - statement delimiters, 18–19
 - statements, 18–19
 - type definition, 15
 - variables, 20–21
- System, 185
- System.Action delegates, 542–544
- System.ApplicationException, 233, 468, 468
- System.ArgumentException, 233, 465, 479
- System.ArgumentNullException, 233
- System.ArithmeticException, 233
- System.Array, 516–517
- System.Array.Reverse(), 104
- System.ArrayTypeMismatchException, 233
- System.Attribute, 738
- System.AttributeUsageAttribute, 745–747
- System.Collection.Generic.List<T>
 - .FindAll(), 689–690
- System.Collections, 185
- System.Collections.Generic
 - .IEnumerable<T>. *See* IEnumerable<T> interface.
- System.Collections.Generics, 185
- System.Collections.Generic
 - .Stack<T>, 496, 609
- System.Collections.IEnumerable. *See* IEnumerable interface.
- System.Collections.Stack.Pop(), 488–491
- System.Collections.Stack.Push(), 488–491
- System.ComponentModel
 - .Win32Exception, 903–904
- System.Console.Clear(), 173
- System.Console.Read(), 25
- System.Console.ReadKey(), 25
- System.Console.ReadLine() method
 - calling, 183–184
 - reading from the console, 24–25
 - return values, 188
- System.Console.Write() method
 - calling, 183–184
 - return values, 188
 - starting a new line, 57, 64
 - writing to the console, 26–28
- System.Console.WriteLine()
 - outputting a blank line, 64
 - round-trip formatting, 52–53
 - starting a new line, 57, 64
 - writing to the console, 26–28
- System.Console.WriteLine() method
 - calling, 183–184
 - overloading ToString(), 412–413
 - return values, 188
- System.Convert, 73
- System.Data, 185
- System.Data.SqlClient
 - .SQLException(), 481
- System.Delegate
 - constraint limitations, 516–517
 - delegate internals, 547–550
 - multicast delegate internals, 586–587
- System.Delegate.Combine()
 - combining delegates, 584
 - event internals, 600
- System.Delegate.Remove()
 - event internals, 600
 - removing delegates from chains, 584
 - removing list elements, 686
- System.Diagnostics
 - .ConditionalAttribute, 749–751
- System.Diagnostics.Processor, 899
- System.Diagnostics.Trace.Write()
 - method, 412
- System.Drawing, 185
- System.Dynamic.DynamicObject, 766
- System.Dynamic
 - .IDynamicMetaObjectProvider interface, 766–769
- System.Enum, 517
- System.Enum.IsDefined(), 407
- System.Enum.Parse(), 404
- System.Environment.CommandLine, 17
- System.Environment.FailFast(), 468, 468
- System.EventArgs, 595–596
- System.EventHandler<T>, 598
- System.Exception, 232, 468, 468
- System.ExecutionEngineException, 468

- System.FormatException, 233
- System.Func delegates, 542–544
- System.GC object, 450
- System.GC.SuppressFinalize(), 458–460
- System.IndexOutOfRangeException, 233
- System.IntPtr, 901
- System.InvalidCastException, 233
- System.InvalidOperationException, 233, 470
- System.IO, 185
- System.IO.DirectoryInfo, 299–300
- System.IO.FileAttributes, 405
- System.Lazy<T>, 462–463
- System.Linq, 185
- System.Linq.Enumerable
 - aggregate functions, 644
 - Average(), 644
 - Count(), 644
 - GroupBy() method, grouping results, 636–637
 - GroupJoin(), 637–638
 - Join(), 632–635
 - Max(), 644
 - method calls, 642–643
 - Min(), 644
 - Select(), 618–619
 - Sum(), 644
 - Where(), 616–617
- System.MulticastDelegate, 547–550
- System.MultiCastDelegate, 516–517
- System.NonSerializable, 754
- System.NotImplementedException, 233
- System.NullReferenceException, 233
- System.Object, 344–345
- System.ObsoleteAttribute, 751–752
- System.OutOfMemoryException, 468, 476
- System.Reflection.MethodInfo
 - property, 547
- System.Runtime.CompilerServices
 - .CallSite<T>, 763–764
- System.Runtime.ExceptionServices
 - .ExceptionDispatchInfo
 - .Catch(), 472
- System.Runtime.ExceptionServices
 - .ExceptionDispatchInfo
 - .Throw(), 472
- System.Runtime.InteropServices
 - .COMException, 468
- System.Runtime.InteropServices
 - .SafeHandle, 906
- System.Runtime.InteropServices
 - .SEHException, 468
- System.Runtime
 - .SerializationException, 757–758
- System.Runtime.Serialization
 - .ISerializable, 755–756
- System.Runtime.Serialization
 - .OptionalFieldAttribute, 758
- System.Runtime.Serialization
 - .SerializationInfo, 755–756
- System.Runtime.Serialization
 - .StreamingContext, 755–756
- System.Security.AccessControl
 - .MutexSecurity objects, 882
- System.SerializableAttribute, 752–754, 758
- System.StackOverflowException, 233, 468
- System.String(), 57
- System.SystemException, 468, 479
- System.Text, 185
- System.Text.RegularExpressions, 185
- System.Text.StringBuilder, 67, 510
- System.Threading, 185
- System.Threading.AutoResetEvent, 884, 887
- System.Threading.Interlocked, 125
 - Add(), 877
 - CompareExchange(), 876–877
 - CompareExchange<T>, 877
 - Decrement(), 877
 - Exchange<T>, 877
 - Increment(), 877–878
 - Read(), 877
- System.Threading.Interlocked
 - methods, 876–877
- System.Threading.ManualResetEvent, 884–887
- System.Threading
 - .ManualResetEventSlim, 884–887
- System.Threading.Monitor, 868–870
- System.Threading.Monitor.Enter(), 394, 868–870, 874–875
- System.Threading.Monitor.Exit(), 394, 870, 874–875
- System.Threading.Monitor.Pulse(), 870
- System.Threading.Mutex, 882–883

System.Threading.Tasks, 185
 System.Threading.Tasks
 .TaskCanceledException, 813–814
 System.Threading.Thread, 774.
 See also Multithreading, with
 System.Threading.Thread.
 System.Threading.Timer, 894
 System.Threading.WaitHandle, 883
 System.Timers.Timer, 894
 System.Type class, accessing metadata,
 723–724
 System.UnauthorizedAccessException,
 483
 System.ValueTuple, 88–90, 503–505
 System.ValueType, 389–390
 System.WeakReference, 452
 System.Web, 185
 System.Windows, 186
 System.Windows.Forms.Timer, 894
 System.Windows.Threading
 .DispatcherTimer, 894
 System.Xml, 186

T

TAP language pattern, code readability,
 844
 TAP (Task-based Asynchronous Pattern).
 See Multithreading, task-based
 asynchronous pattern.
 Target property, 547
 Task-based Asynchronous Pattern (TAP).
 See Multithreading, task-based
 asynchronous pattern.
 Task Parallel Library (TPL). *See* TPL (Task
 Parallel Library).
 Task schedulers, 790–791, 840–842
 TaskCanceledException, 813–814
 TaskCompletionSource.SetResult(),
 836
 TaskCompletionSource<T> object,
 835–837
 TaskContinuationOptions enums,
 799–801
 Task.ContinueWith(), 796–798, 806–807,
 841–842, 845
 TaskCreationOptions.LongRunning
 option, 815–816
 Task.Delay(), 785, 893–894
 Task.Factory.StartNew(), 814–815
 Task.Run(), 814–815

Tasks
 antecedent, 797–798
 associating data with, 795
 asynchronous. *See* Multithreading,
 asynchronous tasks.
 canceling. *See* Multithreading,
 canceling tasks.
 chaining, 797–798
 cold, 792
 composing large from smaller,
 796–798
 continuation, 796–803
 control flow within, 827
 creating, 790–791
 definition, 775, 790–791
 vs. delegates, 791
 disposable, canceling, 816
 drawbacks, 817–823
 hot, 792
 identification, 795
 long-running, canceling, 810–814
 registering for notification of
 behavior, 801–802
 status, getting, 795
 TaskScheduler property, 840–842, 855
 TaskScheduler
 .UnobservedTaskException
 event, 807
 Task<T>, 793–794
 Task.WaitAll(), 793
 Task.WaitAny(), 793
 Temporary storage pool. *See* Stack.
 Ternary operators, definition, 142
 TextNumberParser.Parse(), 466
 textToUpper(), 66
 ThenBy(), 626–628
 ThenByDescending(), 628
 Thermostat, 577–578
 this keyword
 avoiding ambiguity, 253–256
 definition, 252–253
 identifying field owner, 252–253
 locking, 874–875
 with a method, 254–255
 passing in a method call, 255–256
 in static methods, 294
 Thread management, 783–784
 Thread pool, definition, 775
 Thread pooling, definition, 787–789
 Thread safe code, definition, 774

- Thread safe delegate invocation, 582
- Thread-safe event notification, 879
- Thread safety
 - definition, 867
 - delegates, 582
 - thread synchronization, 867
- Thread synchronization. *See also*
 - Deadlocks; Race conditions; Synchronization.
 - best practices, 879–881
 - deadlocks, 881
 - monitors, 868–870
 - multiple threads, 868–870
 - with no `await` operator, 872–873
 - timers, 893–894
 - using a monitor, 868–870
- Thread synchronization, synchronization types
 - concurrent collection classes, 888–889
 - reset events, 884–887
- Thread synchronization, thread local storage
 - definition, 889–890
 - `ThreadLocal<T>`, 890–891
 - `ThreadStaticAttribute`, 891–893
- Thread synchronization, uses for
 - atomicity of reading and writing to variables, 867
 - declaring fields as `volatile`, 875–876
 - event notification with multiple threads, 878–879
 - guidelines, 875
 - lock keyword, 870–872
 - lock objects, 873–874
 - lock synchronization, 870–872
 - locking guidelines, 875
 - locking on `this`, `typeof`, and `string`, 874–875
 - with `MethodImplAttribute`
 - with `MethodImplOptions`
 - `.Synchronized()` method, 875
 - multiple threads and local variables, 867–868
 - sample pseudocode execution, 866
 - synchronizing local variables, 867–868
 - synchronizing multiple threads with `Monitor` class, 868–870
 - with `System.Threading.Interlocked` methods, 876–877
 - thread-safe event notification, 879
 - thread safety, 867
 - thread safety, definition, 867
 - torn read, definition, 867
 - unsynchronized local variables, 867–868
 - unsynchronized state, 864–865
 - `volatile` keyword, 875–876
- `ThreadAbortException`, 786–787
- Threading model, definition, 774
- `ThreadLocal<T>`, 890–891
- Threads
 - aborting, 786–787
 - checking for life, 784
 - creating, 790–791
 - definition, 774
 - foreground vs. background, 783
 - putting to sleep, 784–785
 - reprioritizing, 784
 - unhandled exceptions, 807–810
 - waiting for, 783
- `Thread.Sleep()`, putting threads to sleep, 784–785
- `ThreadState` property, 784
- `ThreadStaticAttribute`, 891–893
- Three-dimensional arrays, initializing, 96
- `Throw()`, 472
- throw statements, 235–238
- `ThrowIfCancellationRequested()`, 814
- Throwing exceptions. *See also* Catching exceptions; Exception handling.
 - `ArgumentNullException`, 468
 - `ArgumentOutOfRangeException`, 468
 - checked and unchecked conversions, 484–486
 - code sample, 466
 - description, 226–232
 - guidelines, 483
 - identifying the parameter name, 467, 468
 - `nameof` operator, 467, 468
 - `NullReferenceException`, 468
 - rethrowing, 471
 - rethrowing wrapped exceptions, 483–484
 - throw statement, 235–238
 - without replacing stack information, 471–472
- TicTacToe game. *See also* Arrays.
 - checking player input, 161–162
 - conditional operators, 142–143

- declaring an array for, 93
- determining remaining moves, 160
- `#endregion` preprocessor directive,
 - example, 176–178
- escaping out of, 165–166
- `if/else` example, 130
- initializing, 95–96
- nested `if` statements, 130–132
- `#region` preprocessor directive,
 - example, 176–178
- tracking player moves, 166–167
- Tilde (~) unary operator, overloading, 428–430
- Tilde (~) bitwise complement operator, 153
- Time slicing, definition, 776
- Time slicing costs, 777
- Timers. *See* Thread synchronization, timers.
- `TKey` parameter, 512
- `ToArray()`, 624
- `ToCharArray()`, 104
- `ToDictionary()`, 624
- `ToList()`, 624
- `ToLookup()`, 624
- Torn read, definition, 867
- `ToString()`, 74, 397–398, 407–409, 412–413, 652–653, 748
- TPL (Task Parallel Library). *See also* Multithreading, parallel loop iterations.
 - asynchronous high-latency operations, 819–823
 - performance tuning, 850–851
- Trapping errors, 226–232. *See also* Exception handling.
- `TrimToSize()`, 683–686, 700
- True/false evaluations. *See also* Boolean expressions.
 - `==` (equal equal) syntax, 53
 - flow control statements, 130
- `true` unary operator, overloading, 428–430
- Try blocks, 228–232
- `TryGetMember()`, 767
- `TryGetPhoneButton()`, 206–209
- `TryParse()`, 74–76, 238–239, 729–730
- `TryParse<T>()`, 404
- `TrySetMember()`, 767
- `Tuple`, 503–505
 - `Tuple.Create()`, 504–505
- Tuples. *See also* Anonymous types.
 - vs. anonymous types, 652
 - code sample, 84–85
 - combining data elements, 83
 - definition, 83
 - discarding, 87
 - naming conventions, 86
 - overloading type definitions, 503–504
 - in query expressions, 662
 - returning multiple values with, 193–194
 - syntax, 86
 - `System.ValueTuple... Type`, 88–90
- `TValue` parameter, 512
- Two-dimensional arrays
 - declaring, 93
 - initializing, 96, 98
- Two's complement notation, 148
- Type categories, reference types
 - definition, 78–80
 - description, 67–68, 78–80
 - heaps, 79–80
 - memory area of the referenced data, 79–80
- Type categories, value types
 - ? (question mark), nullable modifier, 80–83
 - description, 67–68, 78
- Type definition
 - casing, 15
 - naming conventions, 15
 - overloading, 504
 - syntax, 15
- Type objects, retrieving, 723–725
- Type parameter list, 191
- Type parameters, 504
 - constraints on. *See* Constraints on type parameters.
 - generic classes, 496
 - for generic classes or methods, 732–733
 - naming guidelines, 498
- Type parameters in generic methods
 - constraints, 522–523
 - inference, 521–522
- Type safety
 - covariance, 530–531
 - managed execution, 33
 - programming with dynamic objects, 760–761, 766

- Type.ContainsGenericParameters
 - property, 731–732
 - typeof(), 723–725
 - typeof keyword, locking, 874–875
 - typeof operator, 555, 731
 - Types. *See also specific types.*
 - aliasing, 199–200. *See also using directive.*
 - anonymous, 82–83
 - array defaults, 95
 - bool (Boolean), 53–54
 - char (character), 21, 54
 - compatibility, enums, 400
 - conversion, programming with
 - dynamic objects, 761–763
 - data conversion with the as operator, 349–350
 - declaring on the fly. *See Anonymous types.*
 - definition, 21
 - encapsulating, 439
 - extending. *See Inheritance.*
 - implicitly typed local variables, 69
 - inference, generic methods, 520–522
 - integral, 115
 - name qualifier, calling methods, 186–187
 - null, 67–68
 - predefined, 43
 - string, 57
 - underlying, determining, 345–346
 - Unicode standard, 54
 - void, 67–68
 - well formed. *See Well-formed types.*
 - Types, conversions between. *See also*
 - Overloading operators; Overriding object members.
 - cast operator, 69
 - casting, 69
 - checked block example, 71
 - checked conversions, 70–72
 - defining custom conversions, 319–320
 - explicit cast, 69–70
 - implicit conversion, 72–73
 - numeric to Boolean, 72
 - overflowing a float value, 119–120
 - overflowing an integer value, 70
 - Parse(), 73–76
 - System.Convert class, 73
 - ToString(), 74
 - TryParse(), 74–76
 - unchecked block example, 71–72
 - unchecked conversions, 70–72
 - without casting, 73–74
 - Types, fundamental numeric. *See also*
 - Literal values.
 - byte, 44
 - C# vs. C++ short type, 45
 - defaults, 48
 - floating-point types. *See Floating-point types.*
 - formatting numbers as hexadecimal, 52
 - hardcoding values, 48–50
 - hexadecimal notation, 50–51
 - int (integer), 21, 44
 - integer literals, determining type of, 49–50
 - integers, 44–45
 - keywords associated with, 44
 - long, 44
 - sbyte, 44
 - short, 44
 - uint, 44
 - ulong, 44
 - ushort, 44
- ## U
- uint type, 44
 - ulong type, 44
 - UML (Unified Modeling Language), 373
 - Unary operators
 - definition, 141
 - overloading, 428–430
 - UnaryExpression, 568
 - UnauthorizedAccessException, 483, 852
 - Unboxing, 390–394
 - Unchecked block example, 71–72
 - Unchecked conversions, 70–72
 - Uncompress(), 354–355
 - #undef preprocessor directive, 172–174
 - Underscore (_)
 - as digit separator, 50
 - discarding tuples, 87
 - in identifier names, 14
 - line continuation character, 18
 - in variable names, 22
 - Underscores (__) in keyword names, 15
 - Undo(), 488–490
 - Undo, with a generic Stack, 494–496

- Unexpected inequality, float type, 117–120
 - Unhandled exceptions
 - error messages, 226–231
 - handling with `AggregateException`, 803–807
 - observing, 806–807
 - registering for, 808–810
 - on a thread, 807–810
 - `UnhandledException` event, 808
 - Unicode standard
 - character representation, 54–56
 - localizing applications, 54
 - Unified Modeling Language (UML), 373
 - `Union()`, 643
 - Unity, 37
 - Universal Windows Applications, 928, 930
 - Unmanaged code, definition, 32
 - Unmanaged types, 913
 - Unmodifiable. *See* `Immutable`.
 - `UnobservedTaskException` event, 807
 - Unsafe code. *See also* `P/Invoke`; Pointers and addresses.
 - definition, 897
 - description, 911–913
 - executing by delegate, 920–922. *See also* `P/Invoke`.
 - unsafe
 - code blocks, 911–912
 - modifier, 911
 - statement, 911
 - Unsynchronized local variables, 867–868
 - Unsynchronized state, 864–865
 - `Unwrap()`, 821
 - Uppercase, converting text strings to, 66
 - ushort type, 44
 - using directives
 - dropping namespaces, 62–63
 - example, 62–63, 195
 - importing types from namespaces, 196
 - nested namespaces, 196
 - nesting, 197–198
 - wildcards, Java vs. C#, 196
 - using statement
 - deterministic finalization, 454–457
 - resource cleanup, 455–457
 - using static directive
 - abbreviating a type name, 198–199
 - dropping namespaces, 62–63
 - example, 62–63, 198
 - UWP (Windows 10 Universal Windows Platform) applications. *See* Universal Windows Applications.
- ## V
- Validating properties, 268–270
 - Value, passing parameters by, 203–204
 - value keyword, 264
 - Value parameters, 203–204
 - Value type conversion
 - to an implemented interface. *See* Boxing.
 - to its root base class. *See* Boxing.
 - Value types
 - custom types. *See* `Enums`; `Structs`.
 - default operator, 388–389
 - guidelines, 381, 390
 - immutability, 385
 - inheritance, 389–390
 - interfaces, 389–390
 - introduction, 380–381
 - new operator, 388
 - vs. reference types, 204–206, 381–383
 - stack, 381
 - temporary storage pool. *See* `Stack`.
 - Values
 - CIL (Common Intermediate Language), 940
 - hardcoding, 48–50
 - Values property, 695
 - `ValueTuple` class, 503–505
 - var type, 647–649, 651
 - Variable names
 - camelCase, 22
 - underscore (`_`) in, 22
 - Variable parameter lists, passing, 212–213
 - Variables
 - assigning values to, 22–23
 - changing string values, 24
 - declaring, 21–22
 - definition, 20
 - global, C++ vs. C#, 289
 - local. *See* Local variables.
 - naming conventions, 22
 - setting to null, 67
 - syntax, 20–21
 - type, 21
 - using, 23
 - Variadic generic types, 505

- Verbatim string literals
 - description, 58
 - displaying a triangle, 58
 - starting a new line, 57
 - Verbatim strings
 - coding, 57, 58
 - escape sequence, 58
 - VerifyCredentials(), 372
 - Versioning, 756–758
 - Versioning interfaces, 374–375
 - Versioning serialization, 756–758
 - Vertical bar, equal sign (=) bitwise OR assignment operator, 152–153
 - Vertical bar (|) bitwise OR operator, 149–152, 151, 426–428
 - Vertical bars (||) OR operator, 139–140, 428
 - VES (Virtual Execution System). *See also* CIL (Common Intermediate Language); CLI (Common Language Infrastructure); Runtime.
 - definition, 924
 - description, 944
 - managed execution, 32
 - Virtual abstract members, 341
 - Virtual Execution System (VES). *See* VES (Virtual Execution System).
 - Virtual fields, properties as, 273–274
 - Virtual functions, pure, 341
 - Virtual members, sealing, 335
 - Virtual memory, allocating with P/Invoke, 900
 - Virtual methods
 - custom dynamic objects, 766–769
 - Java vs. C#, 473
 - overriding base classes, 326–330
 - virtual modifier, 326–330
 - VirtualAllocEx(), 900
 - VirtualAllocEx() API, 900–901
 - VirtualMemoryManager, 899
 - VirtualMemoryPtr, 906
 - Visual Basic vs C#
 - changing the number of items in an array, 102
 - implicitly typed variables, 82
 - importing namespaces, 196
 - line-based, statements, 18
 - Me keyword, 254
 - Redim statement, 102
 - redimensioning arrays, 102
 - Variant, 82
 - void type, 69
 - Visual code editors, 176–178
 - Visual Studio 2017, 435–436
 - Visual Studio Code, 3
 - void methods, 193
 - void type
 - C++ vs. C#, 68
 - dereferencing, 918
 - description, 68
 - no value vs. empty string, 68
 - in partial methods, 310–311
 - as a return, 193
 - returning from an asynchronous method, 830–833
 - strings, 68
 - use for, 68
 - volatile keyword, 875–876
- ## W
- Wait(), 884–887
 - WaitAll(), 883–884
 - WaitAny(), 883–884
 - WaitAsync(), 887–888
 - WaitForExit(), 875
 - WaitHandle, 816
 - WaitOne(), 883–887
 - Warning messages, disabling/restoring, 175–176
 - #warning preprocessor directive, 172, 174–175
 - Weak references, garbage collection, 451–452
 - WebRequest.GetResponseAsync(), 821
 - Well-formed types
 - determining whether two objects are equal, 420–423
 - implementing Equals() equality operator, 420–423
 - lazy initialization, 461–463
 - object identity vs. equal object values, 416–419
 - overriding Equals() equality operator, 420–423
 - Well-formed types, garbage collection. *See also* Resource cleanup.
 - Collect(), 450
 - introduction, 449–450
 - in .NET, 450–451

- root references, 450
- strong references, 451
- weak references, 451–452
- Well-formed types, namespaces
 - in the CLR (Common Language Runtime), 442
 - guidelines, 445
 - introduction, 442–445
 - naming conventions, 442
 - nesting, 443–444
- Well-formed types, overloading operators
 - binary operators, 426–428
 - binary operators combined with assignment operators, 428
 - cast operator, 430–431
 - conditional logical operators, 428
 - conversion operators, 430–432
 - unary operators, 428–430
- Well-formed types, overriding object members
 - Equals() equality operator, 415–423
 - GetHashCode(), 413–415
 - ToString(), 412–413
- Well-formed types, referencing other assemblies
 - changing the assembly target, 433
 - class libraries, 433–438
 - encapsulation of types, 439
 - internal access modifiers on type declarations, 439–440
 - protected internal type modifier, 440–441
 - public access modifiers on type declarations, 439–440
 - referencing assemblies, 434–436
 - type member accessibility modifiers, 441. *See also specific modifiers.*
- Well-formed types, resource cleanup
 - exception propagation from constructors, 460
 - finalization, 453–460
 - finalization (f-reachable) queue, 458
 - garbage collection, 458–460
 - guidelines, 459–460
 - with IDisposable, 455–460
 - invoking the using statement, 455–457
 - resurrecting objects, 461
- Well-formed types, XML comments
 - associating with programming constructs, 446–447
 - generating an XML documentation file, 448–449
 - guidelines, 449
 - introduction, 445–446
 - single-line, 30
- XML documentation file, 448–449
- XML (Extensible Markup Language), 31

Y

- yield break statement, 715–716
- yield return statement
 - contextual keyword, 13
 - early prototype implementations, 13
 - implementing `BinaryTree<T>`, 708–709

- within a loop, 712–714
- requirements, 719–720
- returning iterator values, 708–709

Z

- `ZipCompression`, 365



Index of 7.0 Topics

Symbols

- _ (underscore)
 - in C# 7.2, 51
 - as digit separator, 50
 - discarding out parameter, 208
 - discarding tuples, 87
- ==, != (equals, exclamation point, equal sign) implementing on tuple (ValueTuple) in C# 7.3, 424

A

- Anonymous types
 - drawbacks, 652
 - vs. tuples, 652
- async keyword
 - ValueTask<T> return, 829–830
- async Main method, 828
 - in C# 7.1, 828, 872
- async methods, return of ValueTask<T>, 828–830
- await operators, with catch or finally statements, 846

B

- Binary literals, 51

C

- C# 7.1
 - async Main methods, 828, 872
 - default without type parameter, 389, 502
 - inferring tuple element names, 85, 87

C# 7.2

- _ (underscore), 51
- digit separator before and after numeric literal, 51
- pass read-only value type by reference, 209
- private protected, 441
- readonly struct, 385

C# 7.3

- ==, != (equals, exclamation point, equal sign) implementing on tuple (ValueTuple), 424

- camelCase, tuple names, 86–87
- catch statements, await operators, 846
- CIL (Common Intermediate Language)
 - disassembling, tools for, 34
 - ILDASM, 34
 - sample output, 35–37
- Class libraries, referencing with Dotnet CLI, 436

Constructors

- expression-bodied member implementation, 283
- finalizers, 283

- Create(), tuple instantiation, 504–505

D

- Deconstruct(), 287–288
- Deconstructors, 287–288
- Default without type parameter in C# 7.1, 389, 502
- Digit separator before and after numeric literal in C# 7.2, 51

Disassembling CIL, tools for, 34
 Discards, 87
 Discarding out parameter, 208
 Dotnet CLI, referencing class libraries, 436

E

Exception handling, catching exceptions
 from `async void` methods, 830–833
 Expression bodied members
 constructors, 285
 finalizers, 283
 properties, 264–265

F

Finalizers with expression bodied
 members, 283
 finally statements with `await`
 operators, 846
 Frameworks, definition, 38

G

Garbage collection, return by reference, 210
 Generic types
 `Create()` tuple, 504–505
 `System.ValueTuple`, 503–505
 `Tuple`, 503–505
 `ValueTuple`, 503–505
`GroupBy()`, with tuples, 636–637
`GroupJoin()`, with tuples, 637–640

H

Hexadecimal numbers as binary literal
 numbers, 51

I

ILDASM (IL Disassembler), 34
 Inferring tuple element names in C# 7.1,
 85, 87
 Inner join, with tuples 632–635
`is` operator, pattern matching, 346–347

J

`Join()` with tuples, 632–635

L

LINQ with tuples, 632–641
 Local functions, 39, 540, 834–835

M

Methods, returning multiple values with
 tuples, 193–194

Multithreading, task-based
 asynchronous pattern
 with `async` and `await` returning
 `ValueTuple`, 824–829
 `async Main` method, 830–833
 asynchronous lambdas with local
 functions, 834–835

N

.NET Core, description, 37
 .NET Framework, 37
 .NET frameworks, predominant
 implementations, 37
 .NET versions, mapped to C# releases,
 39–40

O

One-to-many relationships with tuples,
 637–638

out

 discarding out parameter, 208
 no longer declaring beforehand, 75,
 206–209

Outer joins with tuples, 639–642

Output, passing parameters,
 206–209

Overloading type definitions with
 tuples, 503–504

P

Parameters, passing
 discarding out parameter, 208
 out, 75, 206–209
 output, 75, 206–209
 returning multiple values with tuples,
 206–209

PascalCase, tuple names, 86–87

Pass read-only value type by reference in
 C# 7.2, 209

Passing, parameters
 out, 75, 206–209
 output, 75, 206–209

Pattern matching

 introduction, 165
 with the `is` operator, 346–347
 with a `switch` statement, 347–349

`private protected`
 accessibility modifier, 441
 in C# 7.2, 441

Properties with expression bodies,
 264–265

R

readonly struct in C# 7.2, 385
 ref return, 209–211
 Return by reference, 209–211

S

Select() vs. SelectMany() with tuples, 641
 Standard query operators with tuples, collections of collections, 640–641
 equi-joins, 638
 grouping results with GroupBy(), 636–637
 inner join, 632–635
 normalized data, 633
 one-to-many relationships, with GroupJoin(), 637–638
 outer joins, with GroupJoin(), 639–640
 outer joins, with SelectMany(), 639–642
 switch statements with pattern matching, 164, 347–349
 System.Linq.Enumerable with tuples
 GroupBy() method, grouping results, 636–637
 GroupJoin(), 637–638
 Join(), 632–635
 System.ValueTuple, 88–90, 503–505

T

Throwing exceptions with expressions, 466, 471

TryParse(), 74–76
 Tuple, 503–505
 Tuple.Create(), 504–505
 Tuples
 vs. anonymous types, 652
 code sample, 84–85
 combining data elements, 83
 definition, 83
 discarding, 87
 Equals() with tuples, 423–424
 GetHashCode() with tuples, 424
 naming conventions, 86
 returning multiple values with, 193–194
 syntax, 86
 System.ValueTuple Type, 88–90, 503–504
 Type definition overloading, with ValueTuple, 503–504

U

Underscore (_) as digit separator, 50
 discarding out parameter, 208
 discarding tuples, 87
 Unity, 37

V

ValueTuple, 503–505
 ValueTask<T>, returning from an asynchronous method, 828–830



Index of 6.0 Topics

Symbols

- `$@` (dollar sign, at sign), string interpolation, 59–60
- `?.` (question mark, dot) null-conditional operator, 144–146
- `.` (dot) operator, 145

A

- APIs (application programming interfaces)
 - definition, 38
 - as frameworks, 38
- `ArgumentException`, 468
- `ArgumentNullException`, 467, 468
- `ArgumentOutOfRangeException`, 467, 468
- Automatically implemented properties
 - initializing, 266–267
 - `NextId` implementation, 296
 - read-only, 272, 304

B

- Boolean expressions, logical operators
 - `?.` (question mark, dot), null-conditional operator, 144–146
 - `.` (dot) operator, 145

C

- `CallerMemberName` parameter, 734
- Catching exceptions
 - conditional clauses, 470–471
 - exception conditions, 470–471
- Checking for null, multicast delegates, 580–581

- CIL (Common Intermediate Language)
 - disassembling, tools for, 34
 - ILDASM, 34
 - sample output, 35–37
- Classes, static, 297
- Collection initializers, basic requirements, 605–606
- Conditional clauses, catching exceptions, 470–471
- Console output, formatting with string interpolation, 26

D

- default operator, 388–389
- Delegates, with the null-conditional operator, 146–147
- Disassembling CIL, tools for, 34
- Dollar sign, at sign (`$@`), string interpolation, 59–60
- Dot (`.`) operator, 145

E

- Exception handling, guidelines, 468, 477–478

F

- `FailFast()`, 468, 468
- Formatting with string interpolation, 26
- Frameworks, definition, 38

I

- ILDASM (IL Disassembler), 34
- Immutability, value types, 385

M

Managed execution, CIL (Common Intermediate Language), 32–34
 Method declaration, example, 189–190
 Mono, 37
 Move(), 385
 Multicast delegates, checking for null, 580–581

N

nameof operator
 properties, 270–271, 733–734
 throwing exceptions, 467, 468
 .NET Core description, 37
 .NET Framework, 37
 .NET frameworks, predominant implementations, 37
 .NET versions, mapped to C# releases, 39–40
 new operator, value types, 388
 Null-conditional operator
 delegates, 146–147
 question mark, dot (?.), 144–146
 short circuiting with, 144–146
 NullReferenceException, throwing exceptions, 467, 468

O

Obfuscators, 34
 OutOfMemoryException, 468

P

Parameter name, identifying when throwing exceptions, 467, 468
 ParamName property, 468
 Properties
 automatically implemented, 272, 296
 automatically implemented, read-only, 303–304
 guidelines, 272
 nameof operator, 270–271
 read-only, 272
 read-only automatically implemented, 303–304
 static, 296

Q

Question mark, dot (?.) null-conditional operator, 144–146

R

Read-only
 automatically implemented properties, 303–304
 properties, 272
 readonly modifier, guidelines, 304
 Referential identity, 388

S

Short circuiting with the null-conditional operator, 144–146
 Static
 classes, 297
 properties, 296
 String interpolation
 formatting with, 26
 syntax prefixes, 59–60
 struct keyword, declaring a struct, 384
 Structs
 declaring, 384
 default value, 387
 default value for, 388–389
 definition, 384
 finalizer support, 388
 guidelines, 387
 initializing, 385–387
 referential identity, 388
 System.ApplicationException, 468, 468
 System.Environment.FailFast(), 468, 468
 System.Exception, 468, 468
 System.ExecutionEngineException, 468
 System.OutOfMemoryException, 468
 System.Runtime.InteropServices.COMException, 468
 System.Runtime.InteropServices.SEHException, 468
 System.StackOverflowException, 468
 System.SystemException, 468

T

Throwing exceptions
 ArgumentNullException, 468
 ArgumentOutOfRangeException, 468
 code sample, 466
 identifying the parameter name, 467, 468
 nameof operator, 467, 468
 NullReferenceException, 468

U

Unity, 37

using directives

 dropping namespaces, 62–63

 example, 62–63

using static directive

 abbreviating a type name, 198–199

 dropping namespaces, 62–63

 example, 62–63, 198

V

Value types

 default operator, 388–389

 immutability, 385

 new operator, 388

W

when clauses, catching exceptions, 470

X

Xamarin, 37



Index of 5.0 Topics

A

APIs (application programming interfaces)
definition, 38
as frameworks, 38

C

Capturing loop variables, 564–566
Catch(), 472
Catching exceptions, rethrowing existing
exceptions, 471
CIL (Common Intermediate Language)
disassembling, tools for, 34
ILDASM, 34
sample output, 35–37
Closures, 564

D

Delay(), 893–894
Disassembling CIL, tools for, 34

E

ExceptionDispatchInfo.Throw(), 472

F

Frameworks, definition, 38

I

ILDASM (IL Disassembler), 34

L

Lambda expressions
capturing loop variables, 564–566

closures, 564

outer variable CIL implementation,
563–564

Loop variables, 564–566

M

Managed execution, CIL (Common
Intermediate Language), 32–34
Mono, 37

N

.NET Core, description, 37
.NET Framework, 37
.NET frameworks, 37
.NET versions, mapped to C# releases,
39–40

O

Obfuscators, 34

R

Rethrowing existing exceptions, 471
Run(), 814–815

S

Semaphore, 887–888
SemaphoreSlim, 887–888
SemaphoreSlim.WaitAsync(), 887–888
StartNew(), 814–815
System.Runtime.ExceptionServices
.ExceptionDispatchInfo
.Catch(), 472

- System.Runtime.ExceptionServices
 - .ExceptionDispatchInfo
 - .Throw(), 472
- System.Threading.Timer, 894
- System.Timers.Timer, 894
- System.Windows.Forms.Timer, 894
- System.Windows.Threading
 - .DispatcherTimer, 894

T

- Task.Delay(), 893–894
- Task.Factory.StartNew(), 814–815
- Task.Run(), 814–815
- Thread synchronization
 - with no await operator, 872–873

- timers, 893–894
- Throw(), 472
- Throwing exceptions
 - rethrowing, 471
 - without replacing stack information, 471–472

U

- Unity, 37

W

- WaitAsync(), 887–888

X

- Xamarin, 37

Credits

Item	Title	Attribution
Figure 1.1	The New Project dialog	Courtesy of Microsoft Corporation.
Figure 1.2	Dialog that shows the Program.cs file	Courtesy of Microsoft Corporation.
Figure 4.5	Collapsed Region in Microsoft Visual Studio .NET	Courtesy of Microsoft Corporation.
Figure 10.2	The Project Menu	Courtesy of Microsoft Corporation.
Figure 10.3	The Browse Filter	Courtesy of Microsoft Corporation.
Figure 10.4	XML Comments as Tips in Visual Studio IDE	Courtesy of Microsoft Corporation.
Output 12.1	Output of a Program Similar to the Etch A Sketch Game	Courtesy of Microsoft Corporation.
Output 12.2	Implementing Undo with a Generic Stack Class	Courtesy of Microsoft Corporation.
Figure 18.2	BinaryFormatter Does Not Encrypt Data	Courtesy of Microsoft Corporation.
Figure 19.1	Clock Speeds over Time	Graph compiled by Herb Sutter Used with permission. Original at www.gotw.ca .
Figure 19.3	CancellationTokenSource and CancellationToken Class Diagrams	Courtesy of Microsoft Corporation.