The Addison-Wesley Microsoft Technology Series

Essential C# 12.0

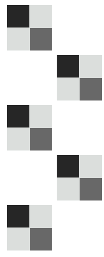
"Welcome to one of the most venerable and trusted franchises you could dream of in the world of C# books—and probably far beyond!"

—From the Foreword by **Mads Torgersen,** *Principal Architect, Microsoft*

MARK MICHAELIS
with
KEVIN BOST, *Technical Editor*

IntelliTect

FREE SAMPLE CHAPTER |



Essential C# 12.0

Mark Michaelis
with Kevin Bost,
Technical Editor

Addison-Wesley
Hoboken, New Jersey

Cover image: Iam_Anuphone/Shutterstock

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: 2023941980

Copyright © 2024 Pearson Education, Inc.

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

ISBN-13: 978-0-13-821951-2
ISBN-10: 0-13-821951-6
$PrintCode

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

Thanks!

Also, to my friends and colleagues at IntelliTect. Thanks for filling in for me when I was writing rather than doing my job and for helping with the myriad of details to improve the content and keep a code base like this running smoothly. Thanks especially for making the EssentialCSharp.com website a reality.

Pearson's Commitment to Diversity, Equity, and Inclusion

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

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

Our ambition is to purposefully contribute to a world where:

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

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

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

Contents at a Glance

Contents vii
Foreword xv
Preface xvii
Acknowledgments xxxi
About the Author xxxiii

1 Introducing C# 1
2 Data Types 49
3 More with Data Types 93
4 Operators and Control Flow 137
5 Parameters and Methods 217
6 Classes 293
7 Inheritance 385
8 Interfaces 443
9 Introducing Structs and Records 487
10 Well-Formed Types 547
11 Exception Handling 601
12 Generics 623
13 Delegates and Lambda Expressions 683
14 Events 727
15 Collection Interfaces with Standard Query Operators 755
16 LINQ with Query Expressions 809
17 Building Custom Collections 833

18 Reflection, Attributes, and Dynamic Programming 881

19 Introducing Multithreading 933

20 Programming the Task-Based Asynchronous Pattern 975

21 Iterating in Parallel 1021

22 Thread Synchronization 1041

23 Platform Interoperability and Unsafe Code 1077

24 The Common Language Infrastructure 1107

Index 1131

Index of 8.0 Topics 1187

Index of 9.0 Topics 1190

Index of 10.0 Topics 1191

Index of 11.0 Topics 1192

Index of 12.0 Topics 1193

Contents

Foreword xv

Preface xvii

Acknowledgments xxxi

About the Author xxxiii

1 Introducing C# 1

Hello, World 2

C# Syntax Fundamentals 12

Working with Variables 24

Console Input and Output 28

Managed Execution and the Common Language Infrastructure 38

Multiple .NET Frameworks 44

Summary 48

2 Data Types 49

Type Name Forms 50

Fundamental Numeric Types 52

More Fundamental Types 63

Conversions between Data Types 84

Summary 92

3 More with Data Types 93

Categories of Types 93

Declaring Types That Allow `null` 96

Implicitly Typed Local Variables 102

Tuples 103

Arrays 112

Summary 134

4 Operators and Control Flow 137

Operators 137

Introducing Flow Control 156

Code Blocks ({}) 161

Code Blocks, Scopes, and Declaration Spaces 164

Boolean Expressions 166

Programming with null 173

Bitwise Operators (<<, >>, |, &, ^, ~) 181

Control Flow Statements, Continued 187

Jump Statements 200

C# Preprocessor Directives 206

Summary 215

5 Parameters and Methods 217

Calling a Method 218

Declaring a Method 225

Local Functions 232

Using Directives 233

Returns and Parameters on Main Method 242

Top-Level Statements 246

Advanced Method Parameters 247

Recursion 261

Method Overloading 264

Optional Parameters 267

Basic Error Handling with Exceptions 272

Summary 291

6 Classes 293

Declaring and Instantiating a Class 298

Instance Fields 302

Instance Methods 305

Using the this Keyword 306

Access Modifiers 314

Properties 316

Constructors 333
Non-Nullable Reference Type Properties with Constructors 346
Nullable Attributes 354
Deconstructors 357
Static Members 359
Extension Methods 370
Encapsulating the Data 372
Nested Classes 376
Partial Classes 379
Summary 384

7 Inheritance 385

Derivation 386
Overriding the Base Class 397
Abstract Classes 410
All Classes Derive from `System.Object` 417
Type Checking 419
Pattern Matching 423
Avoid Pattern Matching When Polymorphism Is Possible 438
Summary 440

8 Interfaces 443

Introducing Interfaces 444
Polymorphism through Interfaces 446
Interface Implementation 451
Converting between the Implementing Class and Its Interfaces 457
Interface Inheritance 458
Multiple Interface Inheritance 461
Extension Methods on Interfaces 461
Versioning 464
Extension Methods versus Default Interface Members 480
Interfaces Compared with Abstract Classes 482
Interfaces Compared with Attributes 484
Summary 484

9 Introducing Structs and Records 487

Reference Equality versus Value Equality 493
Structs 494
Record Classes 500
Record Class Inheritance 503
Records 504
Overriding `object` Members 513
Customizing Record Behavior 521
Boxing 523
Enums 532
Summary 544

10 Well-Formed Types 547

Operator Overloading 548
Referencing Other Assemblies 557
Encapsulation of Types 564
Defining Namespaces 567
XML Comments 571
Garbage Collection and Weak References 576
Resource Cleanup 580
Lazy Initialization 596
Summary 598

11 Exception Handling 601

Multiple Exception Types 601
Catching Exceptions 604
Rethrowing an Existing Exception 607
General Catch Block 609
Guidelines for Exception Handling 610
Defining Custom Exceptions 614
Rethrowing a Wrapped Exception 618
Summary 622

12 Generics 623

C# without Generics 624
Introducing Generic Types 630
Constraints 646

Generic Methods 663
Covariance and Contravariance 669
Generic Internals 676
Summary 681

13 Delegates and Lambda Expressions 683

Introducing Delegates 684
Declaring Delegate Types 688
Lambda Expressions 698
Statement Lambdas 699
Expression Lambdas 702
Anonymous Methods 705
Delegates Do Not Have Structural Equality 707
Outer Variables 710
Static Anonymous Functions 712
Expression Trees 716
Summary 724

14 Events 727

Coding the Publish–Subscribe Pattern with Multicast Delegates 728
Understanding Events 743
Summary 753

15 Collection Interfaces with Standard Query Operators 755

Collection Initializers 756
What Makes a Class a Collection: `IEnumerable` 759
Standard Query Operators 766
Anonymous Types with LINQ 796
Summary 806

16 LINQ with Query Expressions 809

Introducing Query Expressions 810
Query Expressions Are Just Method Invocations 829
Summary 831

17 Building Custom Collections 833

More Collection Interfaces 834
Primary Collection Classes 837

Providing an Indexer 859
Returning null or an Empty Collection 862
Iterators 863
Summary 879

18 Reflection, Attributes, and Dynamic Programming 881

Reflection 881
nameof Operator 894
Attributes 895
Programming with Dynamic Objects 920
Summary 931

19 Introducing Multithreading 933

Multithreading Basics 935
Asynchronous Tasks 943
Canceling a Task 965
Working with System.Threading 972
Summary 973

20 Programming the Task-Based Asynchronous Pattern 975

Synchronously Invoking a High-Latency Operation 975
Asynchronously Invoking a High-Latency Operation Using the TPL 979
The Task-Based Asynchronous Pattern with async and await 984
Introducing Asynchronous Return of ValueTask<T> 991
Asynchronous Streams 994
IAsyncDisposable and the await using Declaration and Statement 998
Using LINQ with IAsyncEnumerable 999
Returning void from an Asynchronous Method 1001
Asynchronous Lambdas and Local Functions 1006
Task Schedulers and the Synchronization Context 1013
async/await with the Windows UI 1015
Summary 1019

21 Iterating in Parallel 1021

Executing Loop Iterations in Parallel 1021
Running LINQ Queries in Parallel 1032

Summary 1039

22 Thread Synchronization 1041

Why Synchronization? 1042

Timers 1073

Summary 1076

23 Platform Interoperability and Unsafe Code 1077

Platform Invoke 1078

Pointers and Addresses 1093

Executing Unsafe Code via a Delegate 1104

Summary 1105

24 The Common Language Infrastructure 1107

Defining the Common Language Infrastructure 1107

CLI Implementations 1109

.NET Standard 1113

Base Class Library 1113

C# Compilation to Machine Code 1114

Runtime 1116

Assemblies, Manifests, and Modules 1121

Common Intermediate Language 1124

Common Type System 1125

Common Language Specification 1125

Metadata 1126

.NET Native and Ahead of Time Compilation 1127

Summary 1128

Index 1131

Index of 8.0 Topics 1187

Index of 9.0 Topics 1190

Index of 10.0 Topics 1191

Index of 11.0 Topics 1192

Index of 12.0 Topics 1193

This page intentionally left blank

Foreword

Welcome to one of the most venerable and trusted franchises you could dream of in the world of C# books—and probably far beyond! Mark Michaelis's *Essential C#* book has been a classic for years, but it was yet to see the light of day when I first got to know Mark.

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

It was pretty quiet in the hands-on labs area, though, where people could try out the technology preview themselves with nice scripted 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 may have met a lot of people for the first time at that conference—people with whom I have since formed great relationships. But to be honest, I don't remember them—it's all a blur. The only one I remember is Mark. Here is why: When I asked him if he was liking the new stuff, he didn't just join the rave. He was totally level-headed: *"I don't know yet. I haven't made up my mind about it."* He wanted to absorb and understand the full package, and until then he wasn't going to let anyone tell him what to think.

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

These are the same qualities that make Mark such a great writer. He goes right to the essence and communicates with great integrity, no sugarcoating, and a keen eye for practical value and real-world problems. Mark has a great gift of providing clarity and elucidation, and no one will help you get C# 12.0 like he does.

Enjoy!

—Mads Torgersen
Principal Architect, 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# 12.0 and .NET 8.
- It serves as a timeless reference even after you gain proficiency with the language.

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

Many topics are not covered in this book. You won't find coverage of topics such as ASP.NET, Entity Framework, Maui, 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, *Essential C# 12.0* 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 more advanced areas, given a strong foundation in the C# langauge.

Target Audience for This Book

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

- *Beginners:* If you are new to programming, this book serves as a resource to help transition you from an entry-level programmer to a C# developer who is comfortable with any C# programming task that's thrown your way. This book not only teaches you syntax but also trains you in good programming practices that will serve you throughout your programming career.
- *Structured programmers:* Just as it's best to learn a foreign language through immersion, learning a computer language is most effective when you begin using it before you know all the intricacies. In this vein, the 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, and Visual Basic programmers fall into this category. Many of you are already completely comfortable with semicolons and curly braces. A brief glance at the code in Chapter 1 reveals that, at its core, C# is like other C- and C++-style languages that you already know.
- *C# professionals:* For those already versed in C#, this book provides a convenient reference for less frequently encountered syntax. Furthermore, it provides insight into language details and subtleties that are seldom addressed. Most important, it presents the guidelines and patterns for programming robust and maintainable code. This book also aids in the task of teaching C# to others. With the emergence of C# 3.0 through 12.0, some of the most prominent enhancements are

  1. String interpolation (see Chapter 2)
  2. Implicitly typed variables (see Chapter 3)
  3. Tuples (see Chapter 3)
  4. Nullable reference types (see Chapter 3)
  5. Pattern matching (see Chapter 4)
  6. Extension methods (see Chapter 6)
  7. Partial methods (see Chapter 6)
  8. Default interface members (see Chapter 8)
  9. Anonymous types (see Chapter 12)
  10. Generics (see Chapter 12)

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

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

Features of This Book

Essential C# 12.0 is a language book that adheres to the core C# Language Specification. To help you understand the various C# constructs, it provides numerous examples demonstrating each feature. Accompanying each concept are guidelines and best practices, ensuring that code compiles, avoids likely pitfalls, and achieves maximum maintainability.

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

Website

The interactive website for the book, available at https://essentialcsharp.com, provides the online chapters, which allows full text search. Shortly after this writing, I expect the website to provide interactive code editing and client-side compilation of

many of the code listings. This will allow you to focus on the language rather than getting distracted by installations or dotnet setup issues.

Source Code Download

In addition to the EssentialCSharp.com website, all the source code is available on GitHub at https://github.com/IntelliTect/EssentialCSharp so it can be downloaded or cloned locally to your computer. This enables you to work through the code samples as is or modify them and see the effects. This is a great way to learn the intricacies of the language.

C# Coding Guidelines

One of the more significant enhancements included in *Essential C# 12.0* is the C# coding guidelines, as shown in the following example taken from Chapter 17:

Guidelines

DO ensure that equal objects have equal hash codes.

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

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

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

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

Code Samples

The code snippets in most of this text can run on most implementations of the Common Language Infrastructure (CLI), but the focus is on the .NET 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# 8.0, 9.0, 10.0, 11.0, or 12.0 is called out in the C# version indexes at the end of the book.

Here is a sample code listing.

Listing 1.21: Commenting Your Code

```
public class CommentSamples
{
    public static void Main()
    {
        string firstName; // Variable for storing the first name
        string lastName;  // Variable for storing the last name

        Console.WriteLine("Hey you!");

        Console.Write /* No new line */ ("Enter your first name: ");
        firstName = Console.ReadLine();

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

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

        Console.WriteLine("Your full name is {1}, {0}.",
            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 */
```

```
Console.Write /* No new line */ ("Enter your first name: ");
```

- 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.

Listing 2.23: Error; `string` Is Immutable

```
Console.Write("Enter text: ");
string text = Console.ReadLine();

// UNEXPECTED: Does not convert text to uppercase
text.ToUpper();

Console.WriteLine(text);
```

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

```
// ...
```

Output 1.7

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

Your full name is Inigo Montoya.
```

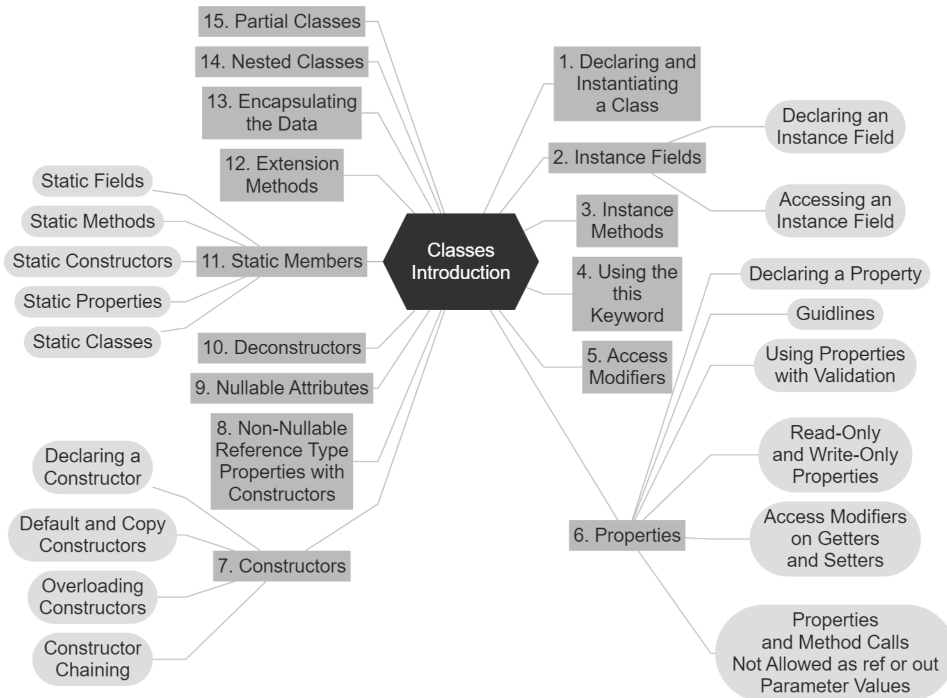
- Console output is the output from a particular listing that appears following the listing. User input for the program appears in boldface.

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.

Mind Maps

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



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

Helpful Notes

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

- Beginner Topics provide definitions or explanations targeted specifically toward entry-level programmers.
- Advanced Topics enable experienced developers to focus on the material that is most relevant to them.
- Callout notes highlight key principles in boxes so that readers easily recognize their significance.
- Language Contrast sidebars identify key differences between C# and its predecessors to aid those familiar with other languages.

How This Book Is Organized

At a high level, software engineering is about managing complexity, and it is toward this end that I have organized *Essential C# 12.0*. Chapters 1–5 introduce structured programming, which enable you to start writing simple functioning code immediately. Chapters 6–10 present the object-oriented constructs of C#. Novice readers should focus on fully understanding this section before they proceed to the more advanced topics found in the remainder of this book. Chapters 12–14 introduce additional complexity-reducing constructs, handling common patterns needed by virtually all modern programs. This leads to dynamic programming with reflection and attributes, which is used extensively for threading and interoperability in the chapters that follow.

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

Here is a description of each chapter (in this list, chapter numbers shown in **bold** indicate the presence of C# 10.0–12.0 material).

- ***Chapter 1****, Introducing C#:* After presenting the C# “Hello World” program, this chapter proceeds to dissect it. This should familiarize readers with the look and feel of a C# program and provide details on how to compile and debug

their own programs. It also touches on the context of a C# program's execution and its intermediate language.

- *Chapter 2, Data Types:* Functioning programs manipulate data, and this chapter introduces the primitive data types of C#.
- *Chapter 3, More with Data Types:* This chapter includes coverage of two type categories: value types and reference types. From there, it delves into implicitly typed local variables, tuples, the nullable modifier, and the C# 8.0–introduced feature, nullable reference types. It concludes with an in-depth look at a primitive array structure.
- *Chapter 4, Operators and Control Flow:* To take advantage of the iterative capabilities in a computer, you need to know how to include loops and conditional logic within your program. This chapter also covers the C# operators, data conversion, and preprocessor directives.
- ***Chapter 5**, Methods and Parameters:* This chapter investigates the details of methods and their parameters. It includes passing by value, passing by reference, and returning data via an out parameter. In C# 4.0, default parameter support was added, and this chapter explains how to use default parameters.
- ***Chapter 6**, Classes:* Given the basic building blocks of a class, this chapter combines these constructs to form fully functional types. Classes form the core of object-oriented technology by defining the template for an object.
- *Chapter 7, Inheritance:* Although inheritance is a programming fundamental to many developers, C# provides some unique constructs, such as the new modifier. This chapter discusses the details of the inheritance syntax, including overriding.
- ***Chapter 8**, Interfaces:* This chapter demonstrates how interfaces are used to define the versionable interaction contract between classes. C# includes both explicit and implicit interface member implementation, enabling an additional encapsulation level not supported by most other languages. With the introduction of default interface members, there is a section on interface versioning in C# 8.0.
- ***Chapter 9**, Introducing Structs and Records:* C# 9.0 introduced the concepts of records for structs and expanded it to reference types in C# 10. 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#. In defining how to create custom structures, this chapter also describes the idiosyncrasies they may introduce.

- *Chapter 10, Well-Formed Types:* This chapter discusses more advanced type definition. It explains how to implement operators, such as + and casts, and describes how to encapsulate multiple classes into a single library. In addition, the chapter demonstrates defining namespaces and XML comments and discusses how to design classes for garbage collection.
- *Chapter 11, Exception Handling:* This chapter expands on the exception-handling introduction from Chapter 5 and describes how exceptions follow a hierarchy that enables creating custom exceptions. It also includes some best practices on exception handling.
- *Chapter 12, Generics:* Generics are perhaps the core feature missing from C# 1.0. This chapter fully covers this 2.0 feature. In addition, C# 4.0 added support for covariance and contravariance—something covered in the context of generics in this chapter.
- ***Chapter 13**, Delegates and Lambda Expressions:* Delegates begin clearly distinguishing C# from its predecessors by defining patterns for handling events within code. This virtually eliminates the need for writing routines that poll. Lambda expressions are the key concept that make C# 3.0's LINQ possible. This chapter explains how lambda expressions build on the delegate construct by providing a more elegant and succinct syntax. This chapter forms the foundation for the collection API discussed next.
- *Chapter 14, Events:* Encapsulated delegates, known as events, are a core construct of the Common Language Runtime. Anonymous methods, another C# 2.0 feature, are also presented here.
- *Chapter 15, Collection Interfaces with Standard Query Operators:* The simple and yet elegantly powerful changes introduced in C# 3.0 begin to shine in this chapter as we take a look at the extension methods of the Enumerable class. This class makes available a collection API known as the standard query operators, which is discussed in detail here.
- *Chapter 16, LINQ with Query Expressions:* Using standard query operators alone results in some long statements that are hard to decipher. However, query expressions provide an alternative syntax that matches closely with SQL, as described in this chapter.
- *Chapter 17, Building Custom Collections:* In building custom APIs that work against business objects, it is sometimes necessary to create custom collections. This chapter details how to do this and in the process introduces contextual keywords that make custom collection building easier.

- ***Chapter 18**, Reflection, Attributes, and Dynamic Programming:* Object-oriented programming formed the basis for a paradigm shift in program structure in the late 1980s. In a similar way, attributes facilitate declarative programming and embedded metadata, ushering in a new paradigm. This chapter looks at attributes and discusses how to retrieve them via reflection. It also covers file input and output via the serialization framework within the Base Class Library. In C# 4.0, a new keyword, *dynamic*, was added to the language. This removed all type checking until runtime, a significant expansion of what can be done with C#.
- *Chapter 19, Introducing Multithreading:* Most modern programs require the use of threads to execute long-running tasks while ensuring active response to simultaneous events. As programs become more sophisticated, they must take additional precautions to protect data in these advanced environments. Programming multithreaded applications is complex. This chapter introduces how to work with tasks, including canceling them, and how to handle exceptions executing in the task context.
- *Chapter 20, Programming the Task-Based Asynchronous Pattern:* This chapter delves into the task-based asynchronous pattern with its accompanying async/await syntax. It provides a significantly simplified approach to multithreaded programming. In addition, the C# 8.0 concept of asynchronous streams is included.
- *Chapter 21, Iterating in Parallel:* One easy way to introduce performance improvements is by iterating through data in parallel using a Parallel object or with the Parallel LINQ library.
- *Chapter 22, Thread Synchronization:* Building on the preceding chapter, this chapter demonstrates some of the built-in threading pattern support that can simplify the explicit control of multithreaded code.
- ***Chapter 23**, Platform Interoperability and Unsafe Code:* Given that C# is a relatively young language, far more code is written in other languages than in C#. To take advantage of this preexisting code, C# supports interoperability—the calling of unmanaged code—through P/Invoke. In addition, C# provides for the use of pointers and direct memory manipulation. Although code with pointers requires special privileges to run, it provides the power to interoperate fully with traditional C-based application programming interfaces.
- *Chapter 24, The Common Language Infrastructure:* Fundamentally, C# is the syntax that was designed as the most effective programming language on top of the underlying Common Language Infrastructure. This chapter delves into how C# programs relate to the underlying runtime and its specifications.

- *Indexes of C# 8.0-12.0:* These indexes provide quick references for the features added in C# 8.0 through 12.0. 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, mark.michaelis.net

X (formerly Twitter): @Intellitect, @MarkMichaelis

Register your copy of *Essential C# 12.0* 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 (9780138219512) 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.

This page intentionally left blank

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 eighth edition of the book, you can only imagine how much my family has sacrificed to allow me to write over the last 18 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.)

In writing *Essential C# 12.0,* I am very grateful that Pearson gave permission for IntelliTect to host the manuscript on the website, essentialcsharp.com. I could never have done this without the IntelliTect team, however. I am so grateful to be surrounded by such amazing software engineers who produce such excellence autonomously from me. Many, many hours were spent parsing the manuscript, formatting it, editing the content, and creating both the website output and the manuscript for print. Thanks so much to the following contributors that made this edition possible: Benjamin Michaelis, Kevin Bost, Daniel Olvera, Jenny Curry, Mikaella Croskrey, Grant Woods, Zack Ward, Josh Willis, Andrew Scott, Anré "Ray" Tanner, Artem Chetverikov, Casey White, Austen Frostad, Tom Clark, Joseph Riddles, John Evans, Kelsey McMahon, Casey Schadewitz, Cameron J. Osborn, Nicole Glidden, and Elizabeth Pauley. Special thanks to Kevin Bost and my son Benjamin Michaelis. In addition to their code contributions, they helped manage the entire web tooling project, making the website a reality.

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

early editions, which no one else thought to mention. This attention to detail, combined with his unswerving demand for excellence, truly establishes *Essential C# 12.0* as a quintessential C# book for those looking to focus on the language.

Of course, Eric Lippert is no less amazing as well. His grasp of C# is truly astounding, and I am very appreciative of his edits, especially when he pushed for perfection in terminology. His improvements to the C# 3.0 chapters were incredibly significant, and in the second edition my only regret was that I didn't have him review all the chapters. However, that regret has since been mitigated: Eric painstakingly reviewed every *Essential C# 4.0* chapter and even served as a contributing author for *Essential C# 5.0* and *Essential C# 6.0*. I am extremely grateful for his role as a technical editor in the editions leading up to *Essential C# 12.0*. Thanks, Eric! I can't imagine anyone better for the job. You deserve all the credit for raising the bar from good to great.

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

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

Thanks to everyone at Pearson/Addison-Wesley for their patience in working with me in spite of my frequent focus on everything else except the manuscript. Thanks also to Malobika Chakraborty and Julie Nahil for helping me through the entire process from proposal to production.

About the Author

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

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

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

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

About the Technical Editor

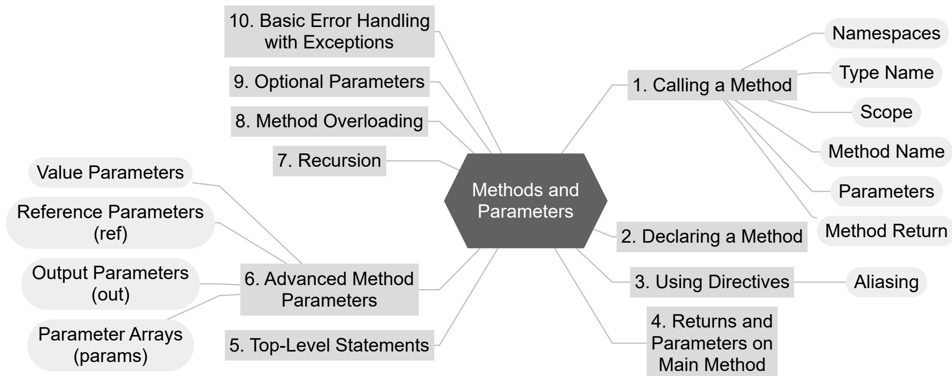
Kevin Bost is a successful Microsoft MVP and senior software architect at IntelliTect. He has been instrumental in building several innovative products, including System.CommandLine, Moq.AutoMocker, and ShowMeTheXAML. When not at work, Kevin can be found online, mentoring other developers on YouTube (youtube.keboo.dev) and maintaining the popular Material Design in XAML toolkit (http://materialdesigninxaml.net/). He also enjoys board games, Ultimate Frisbee, and riding his motorcycle.

This page intentionally left blank

5

Parameters and Methods

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, almost all of the statements in the programs you have written have appeared together in a list without any grouping outside of what the statement or code block provided. When programs become any more complex, this approach quickly becomes difficult to maintain and complicated to read through and understand.

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

LISTING 5.1: Grouping Statements into Methods

```
public class Program
{
    public static void Main()
    {
        int lineCount;
```

```
        string files;

        DisplayHelpText();
        files = GetFiles();
        lineCount = CountLines(files);
        DisplayLineCount(lineCount);
    }

    // ...
    // ...
}
```

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

BEGINNER TOPIC

What Is a Function?

Functions are virtually identical to methods, used for grouping together a sequence of statements to perform a particular action or compute a particular result. In fact, frequently, *method* and *function* are used interchangably. Technically, however, methods are always associated with a type—such as `Program` in Listing 5.1. In contrast, functions have no such association. A function doesn't necessarily have an owning type.

Guidelines

DO give methods names that are verbs or verb phrases.

A method provides a means of grouping related code together. Methods can receive data via **arguments** that are passed to the method **parameters**. Parameters are variables used for passing data from the **caller** (the code containing the method call or **invocation**) to the called or **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

```
public static void Main()
{
    string? firstName;
    string? lastName;

    Console.WriteLine("Hey you!");

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

    Console.Write("Enter your last name: ");
    lastName = Console.ReadLine();

    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 saw in type names (https://essentialcsharp.com/type-name-forms) in Chapter 2, methods may also be called with only the last 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
`System`	Contains the fundamental types and types for conversion between types, mathematics, program invocation, and environment management.
`System.Collections.Generics`	Contains strongly typed collections that use generics.
`System.Data.Entity`	A framework designed to store and retrieve data from a relational database through code generation that maps entities into tables.
`System.Drawing`	Contains types for drawing to the display device and working with images. Note, however, it is mostly Windows specific.

TABLE 5.1: Common Namespaces (continued)

Namespace	Description
`System.IO`	Contains types for working with directories and manipulating, loading, and saving files.
`System.Linq`	Contains classes and interfaces for querying data in collections using a Language Integrated Query.
`System.Text`	Contains types for working with strings and various text encodings and for converting between those encodings.
`System.Text.RegularExpressions`	Contains types for working with regular expressions.
`System.Threading`	Contains types for multithreaded programming.
`System.Threading.Tasks`	Contains types for task-based asynchrony.
`System.Windows`	Contains types for creating rich user interfaces when working with UI technologies such as Windows Presentation Foundation (WPF), or Windows UI Library (WinUI), leveraging Extensible Application Markup Language (XAML) for declarative design of the UI.
`System.Xml.Linq`	Contains standards-based support for XML processing using an object query language called LINQ (See Chapter 15).

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

Guidelines

DO use PascalCasing for namespace names.

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

Type Name

Calls to static methods require the type name qualifier as long as the target method is not within the same type.[1] (As discussed later in the chapter, a using static directive allows you to omit the type name.) For example, a call expression of `Console.WriteLine()` found in the method `HelloWorld.Main()` requires the type, `Console`, to be specified. However, just as with the namespace, C# allows the omission of the type name from a method call whenever the method is a member of the type containing the call expression. (Examples of method calls such as this appear in Listing 5.4.) The type name is unnecessary in such cases because the compiler infers the type from the location of the call. If the compiler can make no such inference, the name must be provided as part of the method call.

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

Scope

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

Method Name

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

---

1. Or base class.

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 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 `Console.WriteLine()`, the method call `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 `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

```
public static void Main()
{
    Console.Write("Enter your first name: ");
    Console.WriteLine($"Hello { Console.ReadLine() }!");
}
```

Instead of assigning the returned value to a variable and then using that variable as an argument to the call to `Console.WriteLine()`, Listing 5.3 calls the `Console.ReadLine()` method within the call to `Console.WriteLine()`. At

execution time, the `Console.ReadLine()` method executes first, and its return value is passed directly into the `Console.WriteLine()` method, rather than into a variable.

Not all methods return data. Both versions of `Console.Write()` and `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 `Console.WriteLine($"Hello { 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

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

```
public class IntroducingMethods
{
    public static void Main()
```

```
    {
        string firstName;
        string lastName;
        string fullName;
        string initials;

        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)
    {
        Console.Write(prompt);
        return Console.ReadLine() ?? string.Empty;
    }

    static string GetFullName(
          string firstName, string lastName) =>
              $"{ firstName } { lastName }";


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

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

Output 5.1

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

```
Enter your last name: Montoya
Hello Inigo Montoya! Your initials are I. M.
```

Listing 5.4 declares five methods. 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.

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. The statements that appear independent from a method and the methods that appear independent from a class are even conforming to this rule because the C# complier generates the surrounding method and class for these seemingly independent constructs. For more information, see "Top-Level Statements" later in this chapter.

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 adding comments to a block of code, because the method name serves to describe what the implementation does. Listing 5.4 is easy to grasp at a glance by just viewing the `Main()` method and not worrying about the details of each called method's implementation.

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

Formal Parameter Declaration

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

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

Guidelines

DO use camelCasing for parameter names.

Method Return Type Declaration

In addition to `GetUserInput()`, `GetFullName()`, and the `GetInitials()` methods requiring parameters to be specified, each of these methods includes a **method return type**. You can tell that a method returns a value because a data type appears immediately before the method name in the method declaration. Each of these method examples specifies a `string` return type. Unlike 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[2] contain one or more `return` statements that return control to the caller. A `return` statement consists of the `return` keyword followed by an expression that computes the value the method is returning. For example, the `GetInitials()` method's `return` statement is `return $"{ firstName[0] }. { lastName[0] }.";`. The expression (an interpolated string in this case) following the `return` keyword must be compatible with the stated return type of the method.

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

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

```
public class Program
{
    // ...

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

2. For example, you could throw an exception instead.

```
    // ...
}
```

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

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

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

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

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

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

Listing 5.6: Returning Multiple Values Using a Tuple

```
public class Program
{
    static string GetUserInput(string prompt)
    {
```

```
        Console.Write(prompt);
        return Console.ReadLine() ?? string.Empty;
    }
    static (string First, string Last) GetName()
    {
        string firstName, lastName;
        firstName = GetUserInput("Enter your first name: ");
        lastName = GetUserInput("Enter your last name: ");
        return (firstName, lastName);
    }
    static public void Main()
    {
        (string First, string Last) name = GetName();
        Console.WriteLine($"Hello { name.First } { name.Last }!");
    }
}
```

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

Expression Bodied Methods

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

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

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

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

Language Contrast: C++—Header Files

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

Local Functions

C# 7.0 introduced the ability to declare functions within methods. In other words, rather than always declaring a method within a class, the method (technically a function) could be declared within another method (see Listing 5.7).

Listing 5.7: Declaring a Local Function

```
public static void Main()
{
    string GetUserInput(string prompt)
    {
        string? input;
        do
        {
            Console.Write(prompt + ": ");
            input = Console.ReadLine();
        }
        while(string.IsNullOrWhiteSpace(input));
        return input!;
    };

    string firstName = GetUserInput("First Name");
    string lastName = GetUserInput("Last Name");
    string email = GetUserInput("Email Address");

    Console.WriteLine($"{firstName} {lastName} <{email}>");
    //...
```

This language construct is called a **local function**. The reason to declare it like this is that the function's scope is limited to invocation from within the method where it is declared. Outside of the method's scope, it is not possible to call the local function. In addition, the local function can access local variables in the method that are declared before the local function. Alternatively, this can explicitly be prevented by adding the `static` to the beginning of the local function declaration (see "Static Anonymous Functions" (https://essentialcsharp.com/outer-variables) in Chapter 13).

Using Directives

Fully qualified namespace names can become quite long and unwieldy. It is possible, however, to import all the types from one or more namespaces so that they can be used without full qualification.

Using Directive Overview

Rather than a declarative that applies to the entire project, C# includes a using directive that applies only to the current file. For example, Listing 5.8 (with Output 5.2) doesn't prefix `RegEx` with `System.Text.RegularExpressions`. The namespace may be omitted because of the `using System.Text.RegularExpressions` directive that appears at the top of the listing.

Listing 5.8: Using Directive Example

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

public class Program
{
    public static void Main()
    {
        const string firstName = "FirstName";
        const string initial = "Initial";
        const string lastName = "LastName";

        // Explaining regular expressions is beyond the
        // scope of this book.
```

```
        // See https://www.regular-expressions.info/ for
        // more information.
        const string pattern = $"""
            (?<{firstName}>\w+)\s+((?<{
            initial}>\w)\.\s+)?(?<{
            lastName}>\w+)\s*
            """;

        Console.WriteLine(
            "Enter your full name (e.g. Inigo T. Montoya): ");
        string name = Console.ReadLine()!;

        // No need to qualify RegEx type with
        // System.Text.RegularExpressions because
        // of the using directive above
        Match match = Regex.Match(name, pattern);

        if (match.Success)
        {
            Console.WriteLine(
                $"{firstName}: {match.Groups[firstName]}");
            Console.WriteLine(
                $"{initial}: {match.Groups[initial]}");
            Console.WriteLine(
                $"{lastName}: {match.Groups[lastName]}");
        }
    }
}
```

Output 5.2

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

A using directive such as `using System.Text` does not enable you to omit `System.Text` from a type declared within a **nested namespace** such as `System.Text.RegularExpressions`. For example, if your code accessed the `RegEx` type from the `System.Text.RegularExpressions` namespace, you would have to either include an additional `using System.Text.RegularExpressions` directive or fully qualify the type as `System.Text.RegularExpressions.RegEx`, not just `RegularExpressions.RegEx`. In short, a using directive does not import types from any nested namespaces. Nested

namespaces, which are identified by the period in the namespace, always need to be imported explicitly.

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

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

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

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

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 from the manuscript. Other namespace directives are generally 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.

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.

Begin 10.0

Implicit Using Directives

Readers will recall from Chapter 1 there was an `ImplicitUsings` element within the `.csproj` file for C# 10.0 and later:

```
<ImplicitUsings>enable</ImplicitUsings>
```

In addition, the source code consistently refers to things like `Console`, even though the fully qualified name is `System.Console`. The ability to abbreviate is afforded by the `ImplicitUsings` element, which tells the compiler to infer the namespace automatically rather than require the programmer to provide it. Using directives, therefore, allows the use of type identifiers without specifying their fully qualified name. To accomplish this, the complier generates global using directives whenever the `ImplicitUsings` element is set to `enable`.

Global Using Directives

If you search the subdirectory of a C# 10.0 (or higher) project, you will notice there is a file with the extension `.GlobalUsing.g.cs`, generally found in an `obj` folder subdirectory, and it contains multiple global using directives such as those found in Listing 5.9.

Listing 5.9: Implicit Usings Generated Global Using Declaratives[3]

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

3. Although removed from this listing for elucidation purposes, the generated global using statements prefix the namespace with "global::" as in `global using global::System`. Discussion of namespace alias qualifiers appears later in the section.

Each of these lines[4] is a global using directive that tells the compiler to imply the namespace qualifier for any type that appears within the namespace specified. For example, `global using global::System` allows implicit using directives like `Console.WriteLine("Hello! My name is Inigo Montoya!")` rather than the fully qualified name `System.Console.WriteLine(...)` because `Console` is defined in the `System` namespace.

Of course, you can provide your own global using declaratives. Say, for example, you frequently are using the `System.Text.StringBuilder` class (initially introduced in Chapter 2). Instead of always referring to it by the fully qualified name, you can provide a global using directive for the `System.Text` namespace and then just refer to the type using the abbreviation `StringBuilder` (see Listing 5.10), also still relying on implicit usings for `System.Console`.

LISTING 5.10: Implicit Using Directives with `StringBuilder`

```
// The global using directive imports all types from
// the specified namespace into the project
global using System.Text;
// ...
public class Program
{
    public static void Main()
    {
        // See Chapter 6 for explanation of new();
        StringBuilder name = new();

        Console.WriteLine("Enter your first name: ");
        name.Append(Console.ReadLine()!.Trim());

        Console.WriteLine("Enter your middle initial: ");
        name.Append( $" { Console.ReadLine()!.Trim('.').Trim() }." );

        Console.WriteLine("Enter your last name: ");
        name.Append($" { Console.ReadLine()!.Trim() }");

        Console.WriteLine($"Hello {name}!");
    }
}
```

4. Ignoring the comment that shows the code was generated by the compiler.

The global using directives applies to the entire project regardless of in which file it appears. And, although C# allows the same global using declarative multiple times, doing so is redundant. For this reason, it is preferable to place all the global using directives into a single file named `Usings.cs` by convention. Collocating them here also provides a well-known location to place them or remove such declaratives.

While global using directives apply to the entire project, C# also supports using directives that apply only to the current file. The global using declarative, however, must appear before any other (non-global) using directives, which we cover after the `.csproj` using element.

csproj Using Element

It is also possible to specify global using declaratives within your project (`.csproj`) file with a `Using` element, as shown in Listing 5.11

LISTING 5.11: Sample .NET Console Project File with Using Element

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <Using Include="System.Net" />
    <Using Static="true" Include="System.Console"/>
  </ItemGroup>
</Project>
```

The important thing to note is that unlike the `ImplicitUsings` element, the `Using` element is a subelement to the `ItemGroup` element rather than the `PropertyGroup` element. And, given the `Using` element for `System.Net` specified, it is no longer necessary to fully qualify `HttpClient`, which is part of the `System.Net` namespace. Instead, the compiler will generate a global using `System.Net` directive when building the project. There is also another `Using` statement that includes a `Static` attribute on it, which we will explain further in the next section.

ADVANCED TOPIC

Nested Using Directives

Not only can you have using directives at the top of a file, but you can also include them at the top of a namespace declaration. For example, if a new namespace, `EssentialCSharp`, were declared, it would be possible to add a using declarative at the top of the namespace declaration (see Listing 5.12).

Listing 5.12: Specifying the `using` Directive inside a Namespace Declaration

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

public class Program
{
    public static void Main()
    {
        // ...
        // No need to qualify RegEx type with
        // System.Text.RegularExpressions because
        // of the using directive above
        Match match = Regex.Match(name, pattern);
        // ...
    }
}
```

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

End 10.0

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 static 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 5.13.

Listing 5.13: Using Static Directive

```
using static System.Console;

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

        WriteLine("Hey you!");

        Write("Enter your first name: ");
        firstName = ReadLine() ?? string.Empty;

        Write("Enter your last name: ");
        lastName = ReadLine() ?? string.Empty;

        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 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 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.

Note that C# 10 or later also support global static using directives such as:

Begin 10.0

```
global using static System.Console;
```

Similarly, a global using static directive can be configured in the csproj file:

```
<Using Static="true" Include="System.Console"/>
```

This is also shown in Listing 5.11.

End 10.0

Aliasing

Begin 12.0

Starting in C# 12.0, the using directive also allows **aliasing** a namespace or any type including tuples, pointers (Chapter 23), array types, and generic types (Chapter 12). 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 5.14, 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 5.14: Declaring a Type Alias

```
using CountDownTimer = System.Timers.Timer;
using StartStop = (DateTime Start, DateTime Stop);

public class HelloWorld
{
    public static void Main()
    {
        CountDownTimer timer;
        StartStop startStop;
```

```
        // ...
    }
}
```

Listing 5.14 uses an entirely new name, `CountDownTimer`, as the alias. It is possible, however, to specify the alias as `Timer`, as shown in Listing 5.15.

Listing 5.15: Declaring a Type Alias with the Same Name

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

public class HelloWorld
{
    public 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.

End 12.0

Begin 10.0

Of course, C# 10 and later also support global aliasing using directives such as:

```
global using Timer = Timers.Timer;
```

And like with all other global using directives, a global using alias directive can be configured in the csproj file:

```
<Using Include=" System.Timers.Timer" Alias="Timer"/>
```

End 10.0

Returns and Parameters on Main Method

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—named `args` by convention. All you need to do to retrieve the parameters is to access the array, as demonstrated in Listing 5.16 with Output 5.3. The output shows a run with no parameters and a second run with two parameters. The purpose of this program is to download a file whose location is specified by a URL. The first command-line argument identifies the URL, and the second argument is the filename to which to save the file. The listing begins with a `switch` statement that evaluates the number of parameters (`args.Length`) as follows:

1. If there are not two parameters, display an error indicating that it is necessary to provide the URL and filename.

2. The presence of two arguments indicates the user has provided both the URL of the resource and the download target filename.

Listing 5.16: Passing Command-Line Arguments to a Main Method

```
public class Program
{
    public static int Main(string[] args)
    {
        int result;
        if(args?.Length != 2 )
        {
            // Exactly two arguments must be specified; give an error
            Console.WriteLine(
                "ERROR:  You must specify the "
                + "URL and the file name");
            Console.WriteLine(
                "Usage: Downloader.exe <URL> <TargetFileName>");
            result = 1;
        }
        else
        {
            string urlString = args[0];
            string fileName = args[1];
```

```
            HttpClient client = new();
            byte[] response =
                client.GetByteArrayAsync(urlString).Result;
            client.Dispose();
            File.WriteAllBytes(fileName, response);
            Console.WriteLine($"Downloaded '{ fileName }' from '{
↪urlString }'.");
            result = 0;
        }
        return result;
    }
}
```

Output 5.3

```
>Downloader
ERROR:  You must specify the URL and the file name
Usage: Downloader.exe <URL> <TargetFileName>
>Downloader https://IntelliTect.com Index.html
Downloaded 'Index.html' from 'https://IntelliTect.com'.
```

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 method declarations, but if it is used, the program can return an exit 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 the Main method except that the first item in the array is the executable filename.

To download the file, Listing 5.16 uses a `System.Net.Http.HttpClient` object but only specified `HttpClient`—its namespace is imported by the `ImplicitUsings` element in the `.csproj` file.

ADVANCED TOPIC

Disambiguate Multiple Main Methods

If a program includes two classes with Main methods, it is possible to specify which one to use as the entry point. In Visual Studio, right-clicking on the project from Solution Explorer and selecting **Properties** provides a user interface on top of the project file. By selecting the **Application** tab on the left, you can edit the Startup Object and select which type's Main method will start the program. The result will be an additional element in the `PropertyGroup`:

```
<StartupObject>AddisonWesley.Michaelis.EssentialCSharp.
Shared.Program
</StartupObject>
```

On the command line, you can specify the same value, setting the `StartupObject` property when running a build. For example:

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

where `AddisonWesley.Program2` is the namespace and class that contains the selected Main method. (In fact, any item in the `PropertyGroup` section of a csproj file can be specified on the command line this way.)

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**.[5] As

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

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.

Begin 9.0

Top-Level Statements

Until C# 9.0, all executable code in C# was required to appear within a type definition and a member such as a method within a type. As we saw in Listing 1.1, this is no longer required. Instead, it is possible for a single file to have **top-level statements**, statements that appear independent of any type definition and even without a Main method. Such statements are allowed in only one file, and they will be the first statements to execute within the program—the equivalent of the many statements that appear in the Main method. In fact, given top-level statements, the compiler generates a class called `Program` that wraps the top-level statements and places them into a Main method. Furthermore, the method has a contextual keyword —`args`—that is the equivalent of the `string[] args` parameter of a Main method.

With top-level statements, C# syntax allows the statements outside a method as a simplification—especially for new C# developers familiar with other languages that are less structured—but then moves these statements into the Main method at compiler time. The end result, therefore, is that there are never any statements in the underlying CIL that are not placed within a type definition and within a type member.

Since the C# compiler moves top-level methods into its own generated main method that is defined within a class named `Program`, the compiler issues an error if you try to define an additional class called `Program`.[6] One additional restriction

6. As the compiler warning indicates, you could define a `Program` class as partial, indicating that the two definitions will be merged into the same `Program` class. For more information, see “Partial Classes” in Chapter 6.

on top-level statements is that any type definition within the same file as the top-level statements must appear after such statements.

The file with top-level statements can also contain methods, which we will call **top-level methods**, and such methods can optionally appear independent of a type definition as well.

When running some of the New Project wizards in Visual Studio, there is frequently an option called "Do not use top-level statements" that allows you to choose whether to go with the simpler version or to generate the structure explicitly, resulting in code that looks more like Listing 1.1 (without a class or Main method). Similarly, on the dotnet command line, some of the project templates (i.e., the Console argument when running the `dotnet new Console` command) frequently have a `--use-program-main` option. However, top-level statements are available to only those projects that have entry points, in other words, Main methods. The compiler doesn't allow top-level statements on programs that do not have Main methods (such as class libraries).

Top-level statements were mainly introduced to remove unnecessary ceremony when writing simple programs, reducing the complexity for beginners especially. Prior to top-level statements, a program with a single statement would require both a type definition and a Main method just to code the single statement. With top-level statements, this structure is optionally eliminated. In addition, top-level statements allow for easier embedding of seemingly complete C# snippets within text like Polyglot notebooks (https://github.com/dotnet/interactive) or the online version of this book (https://essentialcsharp.com).

End 9.0

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

5.17, the value of each variable that `Main()` uses when calling `Combine()` will be copied into the parameters of the `Combine()` method. Output 5.4 shows the results of this listing.

Listing 5.17: Passing Variables by Value

```
public class Program
{
    public 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}",
            Path.DirectorySeparatorChar,
            driveLetter, folderPath, fileName);
        return path;
    }
}
```

Output 5.4

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

Even if the `Combine()` method assigned `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 5.17, the variable names in the caller exactly matched the parameter names in the called method. This matching is provided simply for readability purposes, whether names match is irrelevant to the behavior of the method call. The parameters of the called method and the local variables of the calling method are found in different declaration spaces and have nothing to do with each other.

ADVANCED TOPIC

Reference Types versus Value Types

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

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

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

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

Reference Parameters (`ref`)

Consider Listing 5.18, which calls a function to swap two values, and Output 5.5, which shows the results.

Listing 5.18: Passing Variables by Reference

```
public class Program
{
    public 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 5.5

```
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 5.17'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 5.18, 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.

> **NOTE**
>
> The `ref` modifier assigns a parameter to refer to an existing variable on the stack rather than creating a new variable and copying the argument value into the parameters.

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 `TryGetPhoneButton()` method in Listing 5.19, which returns the phone button corresponding to a character.

LISTING 5.19: Passing Variables Out Only

```
public 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 5.6 shows the results of Listing 5.19.

Output 5.6

```
>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 5.19 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 mistake 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 5.19 uses this feature with the statement `TryGetPhoneButton(character, out char button)` without ever declaring the `button` variable beforehand. Prior to C# 7.0, it would be necessary to first declare the `button` variable and then invoke the function with `TryGetPhoneButton(character, out button)`.

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

Prior to C# 7.0's tuple syntax, a developer of a method might declare one or more `out` parameters to get around the restriction that a method may have only one return

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

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

Read-Only Pass by Reference (`in`)

In C# 7.2, support was added for passing a value type by reference that was read only. Rather than passing the value type to a function so that it could be changed, read-only pass by reference was added: It allows the value type to be passed by reference so that not only copy of the value type occurs but, in addition, the invoked method cannot change the value. 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) results in a compile error indicating that `number` is read only.

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 5.20.

Listing 5.20: `ref` **Return and** `ref` **Local Declaration**

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

By returning a reference to the variable, the caller is then able to update the pixel to a different color, as shown in the highlighted lines of Listing 5.20. Checking for the update via the array shows that the value is now black.

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

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

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

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

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

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

- If you're returning a reference, you obviously must return it. This means, therefore, that in the example in Listing 5.20, 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 (see "Instance Fields" (https://essentialcsharp.com/instance-fields) in Chapter 6 for more information on declaring fields):

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

- You can't declare a by-reference type for an auto-implemented property (see "Properties" (https://essentialcsharp.com/properties) in Chapter 6 for more information on declaring auto-properties):

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

- Properties that return a reference are allowed (see "Properties" (https://essentialcsharp.com/properties) in Chapter 6 for more information on declaring properties):

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

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

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

Parameter Arrays (`params`)

In the examples so far, the number of arguments that must be passed has been fixed by the number of parameters declared in the target method declaration. However, sometimes it is convenient if the number of arguments may vary. Consider the `Combine()` method from Listing 5.17. 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 5.21 with Output 5.7.

LISTING 5.21: Passing a Variable Parameter List

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        string fullName;

        // ...

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

        // ...

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

        // ...

        // Call Combine() with an array
        fullName = Combine(
            new string[] {
                $"{Environment.GetFolderPath(Environment.SpecialFolder.
↪UserProfile)}", "Documents",
                "Web", "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 5.7

```
C:\Users\Inigo\src\Chapter05.Tests\bin\Debug\net7.0\index.html
C:\WINDOWS\system32\Temp\index.html
C:\Users\Inigo\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 for demonstration only as it provides a rough implementation of what `System.IO.Path.Combine()` does already.

There are a few notable characteristics of the parameter array:

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

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

Guidelines

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

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

Recursion

Calling a method **recursively** or implementing the method using **recursion** refers to use of a method that calls itself. Recursion is sometimes the simplest way to implement a particular algorithm. Listing 5.22 counts the lines of all the C# source files (`*.cs`) in a directory and its subdirectory.

LISTING 5.22: 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);
        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;
        // This can be improved with a using statement
        // which is not yet described.
        FileStream stream = new(file, FileMode.Open);
        StreamReader reader = new(stream);
        line = reader.ReadLine();

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

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

Output 5.8 shows the results of Listing 5.22.

Output 5.8

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

All methods within a class must have a unique signature, and C# defines uniqueness by variation in the method name, parameter data types, or number of parameters. This does not include method return data types; defining two methods that differ only in their return data types will cause a compile error. This is true even if the return type is two different tuples. **Method overloading** occurs when a class has two or more methods with the same name and the parameter count and/or data types vary between the overloaded methods.

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

Method overloading is a type of **operational polymorphism**. Polymorphism occurs when the same logical operation takes on many (“poly”) forms (“morphs”) because the data varies. For example, calling `WriteLine()` and passing a format string along with some parameters is implemented differently than calling `WriteLine()` and specifying an integer. However, logically, to the caller, the method takes care of writing the data, and it is somewhat irrelevant how the internal implementation occurs. Listing 5.23 provides an example, and Output 5.9 shows the results.

Listing 5.23: Counting the Lines within `*.cs` Files Using Overloading

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

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

        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;
        // This can be improved with a using statement
        // which is not yet described.
        FileStream stream = new(file, FileMode.Open);
        StreamReader reader = new(stream);
        line = reader.ReadLine();
        while(line is not null)
        {
            if(line.Trim() != "")
            {
                lineCount++;
            }
            line = reader.ReadLine();
        }

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

Output 5.9

```
>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

The C# language designers also added support for **optional parameters**.[7] 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 5.24).

Listing 5.24: Methods with Optional Parameters

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

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

        Console.WriteLine(totalLineCount);
    }
```

7. Introduced in C# 4.0.

```
    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 5.24, the `DirectoryCountLines()` method declaration with a single parameter has been removed (commented out), but the call from `Main()` (specifying one parameter) remains. When no `extension` parameter is specified in the call, the value assigned to `extension` within the declaration (`*.cs` in this case) is used. This allows the calling code to not specify a value if desired, and it eliminates the additional overload that would otherwise be required. Note that optional parameters must appear after all required parameters (those that don't have

default values). Also, the fact that the default value needs to be a constant, compile-time–resolved value is fairly restrictive. You cannot, for example, declare a method like

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

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

Guidelines

DO provide good defaults for all parameters where possible.

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

CONSIDER organizing overloads from the simplest to the most complex.

A second method call feature is the use of **named arguments**.[8] 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 5.25).

Listing 5.25: Specifying Parameters by Name

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

public static void DisplayGreeting(
    string firstName,
    string? middleName = null,
    string? lastName = null
    )
```

8. Introduced in C# 4.0.

```
{
    // ...
}
```

In Listing 5.25, 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,[9] 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.

Guidelines

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

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

Given the combination of method overloading, optional parameters, and named parameters, resolving which method to call becomes less obvious. A call is **applicable** (compatible) with a method if all parameters have exactly one

9. A common occurrence when accessing Microsoft COM libraries such as Microsoft Word or Microsoft Excel.

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.

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 is 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 issues 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)` resolves 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 can pass information about an error to a calling method without using a return value or explicitly providing any parameters to do so. Listing 5.26 with Output 5.10 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 5.26: Converting a `string` to an `int`

```
public static void Main()
{
    string? firstName;
    string ageText;
    int age;

    Console.WriteLine("Hey you!");

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

    Console.Write("Enter your age: ");
    // Assume not null for clarity
    ageText = Console.ReadLine()!;
    age = int.Parse(ageText);

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

Output 5.10

```
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 numerical 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, console applications will display the error on the console with a message such as that shown in Output 5.11.

Output 5.11

```
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.ThrowOverflowOrFormatException(...)
    at System.Number.ParseInt32(String s)
    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 5.27, and the output appears in Output 5.12.

Listing 5.27: Catching an Exception

```
public class ExceptionHandling
{
    public static int Main(string[] args)
    {
        string? firstName;
        string ageText;
        int age;
        int result = 0;

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

        Console.Write("Enter your age: ");
        // Assume not null for clarity
        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 5.12

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

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

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

unhandled as though there were no exception handling. The resultant control flow appears in Figure 5.1.

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

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

The order in which you handle exceptions is significant. Catch blocks must appear from most specific to least specific. The `System.Exception` data type is least specific, so it appears last. `System.FormatException` appears first because it is the most specific exception that Listing 5.27 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 executes 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 5.28 demonstrates the try/finally block, and Output 5.13 shows the results.

LISTING 5.28: Finally Block without a Catch Block

```
using System;

public class ExceptionHandling
```

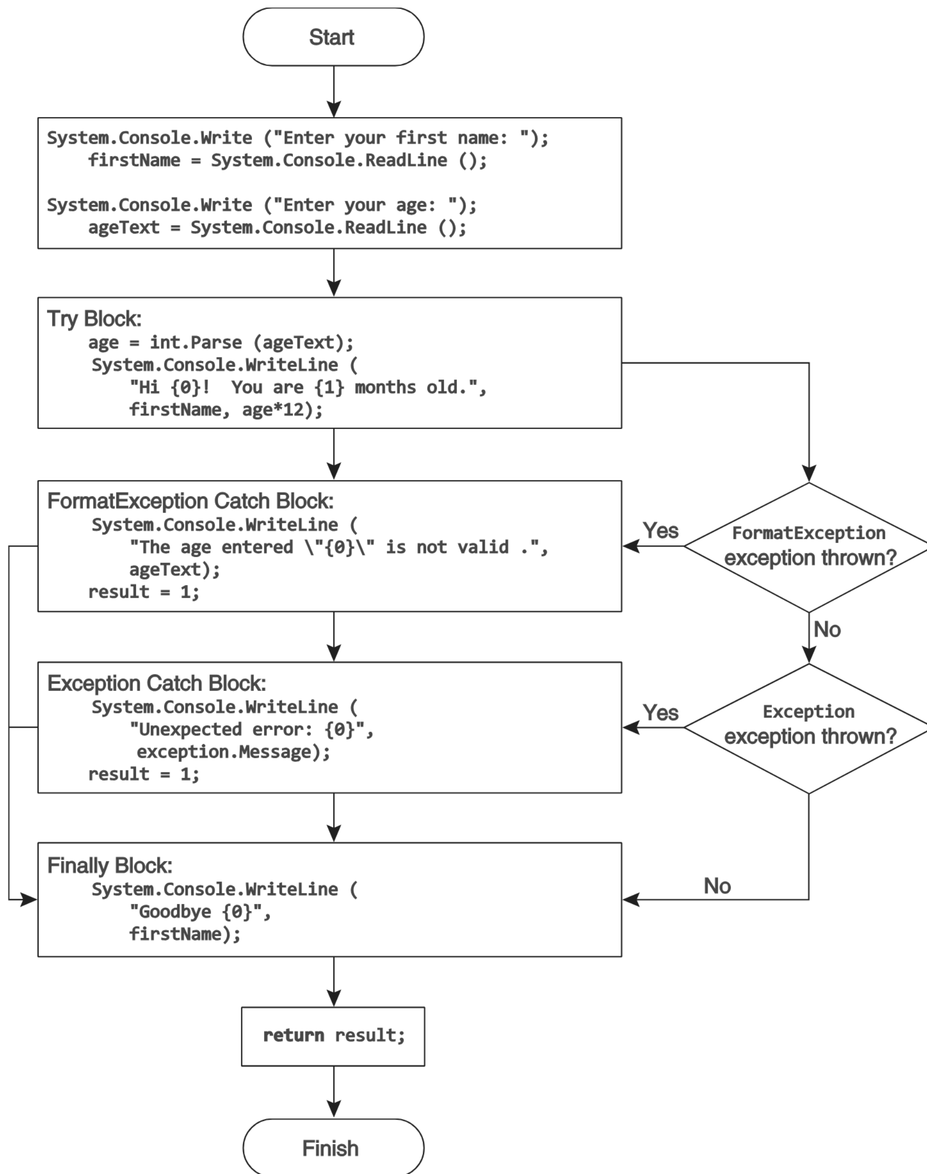


FIGURE 5.1: Exception-handling control flow

```
{
    public static int Main()
    {
```

```
        string? firstName;
        string ageText;
        int age;
        int result = 0;

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

        Console.Write("Enter your age: ");
        // Assume not null for clarity
        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 5.13

```
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.ThrowOverflowOrFormatException(...)
    at System.Number.ParseInt32(String s)
    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.

NOTE

If an exception goes unhandled before the process exits, the order of execution of the finally block, and whether it even executes, is implementation defined.

Guidelines

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

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

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

ADVANCED TOPIC

Exception Class Inheritance

All objects thrown as exceptions derive from `System.Exception`. (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 can handle the exception type specifically, accessing data related to the exception thrown and avoiding conditional logic to determine what type of exception occurred.

Therefore, 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
`System.Exception`	The "base" exception from which all other exceptions derive.
`System.ArgumentException`	Indicates that one of the arguments passed into the method is invalid.
`System.ArgumentNullException`	Indicates that a particular argument is `null` and that this is not a valid value for that parameter.
`System.ApplicationException`	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.

TABLE 5.2: Common Exception Types (continued)

Exception Type	Description
`System.FormatException`	Indicates that the string format is not valid for conversion.
`System.IndexOutOfRangeException`	Indicates that an attempt was made to access an array or other collection element that does not exist.
`System.InvalidCastException`	Indicates that an attempt to convert from one data type to another was not a valid conversion.
`System.InvalidOperationException`	Indicates that an unexpected scenario has occurred such that the application is no longer in a valid state of operation.
`System.NotImplementedException`	Indicates that although the method signature exists, it has not been fully implemented.
`System.NullReferenceException`	Thrown when code tries to find the object referred to by a reference that is `null`.
`System.ArithmeticException`	Indicates an invalid math operation, not including divide by zero.
`System.ArrayTypeMismatchException`	Occurs when attempting to store an element of the wrong type into an array.
`System.StackOverflowException`	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 5.29.

LISTING 5.29: General Catch Blocks

```
// A previous catch clause already catches all exceptions
#pragma warning disable CS1058
// ...
        try
```

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

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

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

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

Guidelines

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

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

Reporting Errors Using a `throw` Statement

C# allows developers to throw exceptions from their code, as demonstrated in Listing 5.30 and Output 5.14.

Listing 5.30: Throwing an Exception

```
public static void Main()
{
    try
    {
        Console.WriteLine("Begin executing");
        Console.WriteLine("Throw exception");
        throw new Exception("Arbitrary exception");
        // Catch 1
        Console.WriteLine("End executing");
    }
    catch(FormatException exception)
    {
        Console.WriteLine(
            "A FormatException was thrown");
    }
    // Catch 1
    catch(Exception exception)
    {
        Console.WriteLine(
            $"Unexpected error: { exception.Message }");
        // Jump to Post Catch
    }

    // Post Catch
    Console.WriteLine(
        "Shutting down...");
}
```

Output 5.14

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

As the comments in Listing 5.30 depict, throwing an exception causes execution to jump from where the exception is thrown into the first compatible catch block (`Catch 1`).[10] In this case, the second catch block handles the exception and writes out an error message. In Listing 5.30, there is no finally block, so following the `WriteLine()` method of `Catch 1`, execution falls through to the `Console.WriteLine()` statement following the try/catch block (after the `Post Catch` comment).

To throw an exception, it is necessary to have an instance of an exception. Listing 5.30 creates an instance using the keyword `new` followed by the type of the exception. Most exception types allow a message to be specified for the exception's construction parameter, so that when the exception occurs, the message can be retrieved.

Sometimes a catch block will trap an exception but be unable to handle it appropriately or fully. In these circumstances, a catch block can rethrow the exception using the `throw` statement without specifying any exception, as shown in Listing 5.31.

Listing 5.31: Rethrowing an Exception

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

    throw;
}
// ...
```

10. Technically, it could be caught by a compatible exception filter as well.

In Listing 5.31, 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. Of course, this is allowed only in a catch statement where the caught exception can be determined.

Guidelines

DO prefer using an empty throw when catching and rethrowing an exception, 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 catching and logging an exception before rethrowing it. Instead, allow the exception to escape until it can be handled appropriately.

Reporting Null Argument Exceptions

When nullability reference types are enabled, the compiler will make a best effort to identify when there is the possibility of a nullable argument passed as a non-nullable argument. If you assign a possible null value to a non-nullable argument, the compiler will issue a warning stating that you are attempting to convert a possible null value to a non-nullable argument. Assuming you address all such warnings appropriately, the compiler will catch most of the cases. However, if the caller is outside of your control (such as from a library you didn't write), doesn't turn on nullable reference types, ignores the warnings, or invokes the method from C# 7.0 or earlier, there is nothing preventing a null argument even though the parameter is declared as non-nullable. For this reason, the best practice is to check that any public non-nullable reference types are not null. With .NET 6.0 you can use a conditional `if (is null)` check:

```
if (is null) throw new ArgumentNullException(...)
```

You can accomplish this in a single statement using the null coalescing assignment operator with a `throw ArgumentNullException` expression if the parameter value is null (see Listing 5.32).

Listing 5.32: Parameter Validation by Throwing `ArgumentNullException`

```
httpsUrl = httpsUrl ??
    throw new ArgumentNullException(nameof(httpsUrl));
fileName = fileName??
    throw new ArgumentNullException(nameof(fileName));


// ...
```

With .NET 7.0, you can use the `ArgumentNullException.ThrowIfNull()` method (see Listing 5.33).

Listing 5.33: Parameter Validation with `ArgumentException.ThrowIfNull()`

```
ArgumentNullException.ThrowIfNull(httpsUrl);
ArgumentNullException.ThrowIfNull(fileName);

// ...
```

Internally, the `ArgumentNullException.ThrowIfNull()` method also throws the `ArgumentNullException`.

Guidelines

DO verify that non-null reference types parameters are not null and throw an `ArgumentNullException` when they are.

DO use `ArgumentException.ThrowIfNull()` to verify values are null in .NET 7.0 or later.

Additional Parameter Validation

There are obviously a myriad of other type constraints that a method may have on its parameters. Perhaps a string argument should not be an empty string, should not be comprised only of whitespace, or must have "HTTPS" as a prefix. Listing 5.34 displays the full `DownloadSSSL()` method, demonstrating this validation.

LISTING 5.34: Custom Parameter Validation

```
public class Program
{
    public static int Main(string[] args)
    {
        int result = 0;
        if(args.Length != 2 )
        {
            // Exactly two arguments must be specified; give an error
            Console.WriteLine(
                "ERROR:  You must specify the "
                + "URL and the file name");
            Console.WriteLine(
                "Usage: Downloader.exe <URL> <TargetFileName>");
            result = 1;
        }
        else
        {
            DownloadSSL(args[0], args[1]);
        }
        return result;
    }

private static void DownloadSSL(string httpsUrl, string fileName)
{
#if !NET7_0_OR_GREATER
    httpsUrl = httpsUrl?.Trim() ??
        throw new ArgumentNullException(nameof(httpsUrl));
    fileName = fileName ??
        throw new ArgumentNullException(nameof(fileName));
    if (fileName.Trim().Length == 0)
    {
        throw new ArgumentException(
            $"{nameof(fileName)} cannot be empty or only whitespace");
    }
#else
    ArgumentException.ThrowIfNullOrEmpty(httpsUrl = httpsUrl?.Trim()!);
    ArgumentException.ThrowIfNullOrEmpty(fileName = fileName?.Trim()!);
```

```
#endif

    if (!httpsUrl.ToUpper().StartsWith("HTTPS"))
    {
        throw new ArgumentException("URL must start with 'HTTPS'.");
    }

    HttpClient client = new();
    byte[] response =
        client.GetByteArrayAsync(httpsUrl).Result;
    client.Dispose();
    File.WriteAllBytes(fileName!, response);
    Console.WriteLine($"Downloaded '{fileName}' from '{httpsUrl}'.");
}
}
```

When using .NET 7.0 or higher, you can rely on the `ArgumentException.ThrowIfNullOrEmpty()` method to check for both null and an empty string. And, if you invoke the `string.Trim()` method when invoking `ThrowIfNullOrEmpty()`, you can throw an exception if the argument content is only whitespace. (Admittedly, the exception message will not indicate whitespace only is invalid.) The equivalent code for .NET 6.0 or earlier is shown in the else directive.

If the null, empty, and whitespace validation pass, Listing 5.34 has an if statement that checks for the “HTTPS” prefix. If the validation fails, the resulting code throws an `ArgumentException`, with a custom message describing the problem.

Introducing the `nameof` Operator

When the parameter fails validation, it is necessary to throw an exception—generally of type `ArgumentException()` or `ArgumentNullException()`. Both exceptions take an argument of type `string` called `paramName` that identifies the name of the parameter that is invalid. In Listing 5.33, we use the `nameof` operator[11] for this argument. The `nameof` operator takes an identifier, like the `httpsUrl` variable, and returns a string representation of that name—in this case, `"httpsUrl"`.

11. Introduced in C# 6.0.

The advantage of using the `nameof` operator is that if the identifier name changes, then refactoring tools will automatically change the argument to `nameof` as well. If no refactoring tool is used, the code will no longer compile, forcing the developer to change the argument manually, which is preferable because the former value would presumably be invalid. The result is that `nameof` will even check for spelling errors. The resulting guideline is: DO use `nameof` for the `paramName` argument passed into exceptions such as `ArgumentException` and `ArgumentNullException` that take such a parameter. For more information, see Chapter 18.

Guidelines

DO use `nameof(value)` (which resolves to `"value"`) for the `paramName` argument when creating `ArgumentException()` or `ArgumentNullException()` type exceptions. (`value` is the implicit name of the parameter on property setters.)

Avoid Using Exception Handling to Deal with Expected Situations

Developers should avoid throwing exceptions for expected conditions or normal control flow. For example, developers should not expect users to enter valid text when specifying their age.[12] Therefore, instead of relying on an exception to validate data entered by the user, developers should provide a means of checking the data before attempting the conversion. (Better yet, they should prevent the user from entering invalid data in the first place.) Exceptions are designed specifically for tracking exceptional, unexpected, and potentially fatal situations. Using them for an unintended purpose such as expected situations will cause your code to be hard to read, understand, and maintain.

Consider, for example, the `int.Parse()` method we used in Chapter 2 to convert a string to an integer. In this scenario, the code converted user input that was expected to not always be a number. One of the problems with the `Parse()` method

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

is that the only way to determine whether the conversion will be successful is to attempt the cast and then catch the exception if it doesn't work. Because throwing an exception is a relatively expensive operation, it is better to attempt the conversion without exception handling. Toward this effort, it is preferable to use one of the `TryParse()` methods, such as `int.TryParse()`. It requires the use of the `out` keyword because the return from the `TryParse()` function is a `bool` rather than the converted value. Listing 5.35 is a code snippet that demonstrates the conversion using `int.TryParse()`.

LISTING 5.35: 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 `TryParse()` method, it is no longer necessary to include a try/catch block simply for the purpose of handling the string-to-numeric conversion.

Another factor in favor of avoiding exceptions for expected scenarios is performance. Like most languages, C# incurs a slight performance hit when throwing an exception—taking microseconds compared to the nanoseconds most operations take. This delay is generally not noticeable in human time—except when the exception goes unhandled. For example, when Listing 5.26 is executed and the user enters an invalid age, the exception is unhandled and there is a noticeable delay while the runtime searches the environment to see whether there is a debugger to load. Fortunately, slow performance when a program is shutting down isn't generally a factor to be concerned with.

Guidelines

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

Summary

This chapter discussed the details of declaring and calling methods, including the use of the keywords `out` and `ref` to pass and return variables rather than via return 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

. (periods)
  fully qualified method names, 17-18, 50-53, 228
  nested namespaces, 221, 234-235, 567-570
  null-conditional operator, 98-99, 173-178, 434
! (exclamation points), 65-67, 143, 809-811
" (double quotes)
  escape sequence, 65-68, 71-74
  strings, 65-73, 143
(hash symbols)
  preprocessor directives, 207-214
#define directive, 207-214, 237-240
#elif directive, 207-214, 237-239
#else directive, 34, 207-214, 239
#endif directive, 207-214, 239, 573
#endregion directive, 207, 211-214, 236-239
#error directive, 34, 207-214, 239
#if directive, 207-214, 239, 898
#line directive, 207-214, 239, 573
#nullable directive, 83, 101-102, 214
#pragma directive, 207-214, 237-239
#region directive, 207-214, 237-239
#undef directive, 207-214, 237-239
#warning directive, 207-214, 239, 304
$ (dollar signs)
  string interpolation, 67, 70-73, 77
% (percent signs)
  compound assignment, 27, 139-142, 149
  overriding, 59-61, 77, 139
  precedence, 139-142, 216, 427
  remainder operation, 55, 138-140, 183-185
& (ampersands), 65-67, 143, 184-185
' (single quotes)
  characters, 64-73
  escape sequence, 65, 66-73
() (parentheses), 16, 141-142, 427
* (asterisks), 59, 65-66, 955
+ (plus signs), 59, 138-139, 142-143
, (commas), 103-106, 115-116, 193
- (minus signs), 138-139, 148-150, 182-183
/ (forward slashes), 59, 65-73, 573
: (colons), 106, 193, 344
; (semicolons)
  ending statements, 13, 23, 192-193
  for loops, 13, 23, 192-193
  preprocessor directives, 13, 23, 207-210
< (less than signs), 138-139, 167-169, 426
<> (angle brackets)
  generics, 623, 630, 635
  XML, 35-37, 168-170, 572-573
= (equals signs), 63, 167-170, 548
> (greater than signs), 138-139, 167-170, 698
? (question marks)
  command-line option, 269, 437, 896

conditional operators, 98, 166-171, 175
null-coalescing operator, 82, 97-98, 173-178
null-conditional operators, 82, 97-98, 173-178

@ (at signs)
identifiers, 16-19, 53, 65-67
verbatim strings, 65-73

[] (square brackets)
arrays, 115-116, 120-123, 132
attributes, 115-116, 881, 896-897
indexers, 122, 126-127, 857-861
null-conditional operators, 82, 98, 173-178

\ (backslashes)
escape sequences, 65, 66-74
as literals, 65-66, 67-73

\a escape sequence, 65-68, 71-73, 160-161

\b escape sequence, 65-73, 161

\f escape sequence, 65-73, 77

\n
escape sequence, 65-73
new lines, 70-72, 78, 160-161

\t escape sequence, 65-73, 160-161

\u escape sequence, 64, 65-73

\x escape sequence, 64, 65-73

^ (carets), 16, 65-67, 168-170

_ (underscores), 17-19, 59, 323

{} (curly braces), 24, 70-71, 162-163

~ (tildes)
complement operator, 143, 168-170, 187
finalizers, 581-584, 589-590, 596
list searches, 815, 820-822, 843
overriding, 65-67, 237, 241

A

abstract modifier
interface refactoring, 444-446, 453-455, 484
limitations, 405, 411, 414

Access modifiers
base class overriding, 393-394, 398, 402
classes, 315-316, 330, 566-567
derivation, 330, 393-394, 566-567
finalizers, 341, 581-582, 583-584
getters and setters, 316-317, 326, 330-333
type declarations, 330, 547, 565-567

Accessing
arrays, 120, 123-124, 129-130
base members, 306, 391-394, 407
instance fields, 303-306, 317, 330
properties, 319-320, 326, 330-333
referent type members, 901, 921-922, 1097
static fields, 360-363, 367, 373

Accuracy of floating-point types, 53-58, 92, 145-148

Acronyms as identifiers, 17-19, 53, 59

Action delegate, 727, 741-747, 753

Activation frames, 66, 246, 935

Add() method
collection initializers, 339, 754-759, 845
dictionaries, 758, 832-834, 845-849
interlocks, 528, 758, 1057
lists, 417, 758, 835-839

Addition
binary operator, 138-143, 149, 550-552
compound assignment, 139-143, 149, 736
increment operator, 138-139, 143, 149-153
strings, 67-69, 143, 514
unary operator, 138-139, 141-143, 149-150

Address operator (&), 183-185, 1098, 1102-1103

Addresses and pointers, 491-492, 1097-1099, 1102-1103

AggregateException type
asynchronous requests, 958-960, 1028, 1035
parallel loops, 959, 1028, 1035-1039
PLINQ queries, 823, 1034-1035, 1038
publish-subscribe pattern, 740-743, 753, 959-960
tasks, 958-960, 975, 994
wrapped exceptions, 622, 959-960, 1035

Aggregation for inheritance, 297, 385-386, 463

Ahead of time (AOT) compilation, 38-39, 1121, 1127-1128

Alerts escape sequence, 65-73, 143, 160

Aliases
- command-line option, 241, 269, 896
- namespaces, 221, 235, 567-570

Allow Unsafe Code option, 621, 1077, 1093-1095
AllowMultiple parameter, 259-260, 269-270, 906-909
AllowNull attribute, 82-83, 348-350, 354
AllowUnsafeBlocks option, 620-621, 1093, 1094-1095
Alternative statements for if statements, 158-163, 197, 205
Ampersands (&)
- address operator, 143, 149, 184-185
- bitwise operators, 169-170, 184-187, 552
- compound assignment, 140-141, 149, 184-185
- logical operators, 143, 168-170, 184-185
- overriding, 17, 143, 407

AND operations
- bitwise operators, 169, 184-187, 540
- enums, 534, 540-542, 550
- logical operators, 143, 167-170, 184-185

Angle brackets (<>)
- generics, 623, 630-635, 677
- XML, 35-37, 571-574, 897

Anonymous functions, 704-706, 797-800, 803
Anonymous methods
- internals, 705, 797-798, 801-803
- lambda expressions as replacement for, 697-706, 710
- overview, 705, 797-800, 803
- parameterless, 704-706, 797, 800-805

Anonymous types
- collection initializers, 339, 754-757, 805
- generating, 705, 797-805
- local variables, 705, 710, 797-799
- overview, 705, 754-756, 797-803
- selecting into, 754-756, 797-805
- type safety and immutability, 505, 797-805

Antecedent tasks, 935, 952-957, 975
Any() operator, 143, 167-169, 800
APIs (application programming interfaces)
- documentation, 45, 444-445, 463-464
- as frameworks, 45, 1110-1113, 1120
- wrapper calls, 45, 1078, 1086

AppDomain
- exceptions, 612-615, 618, 962-964
- process exits, 590-591, 594-595, 962-964

AppDomains for unhandled exceptions, 274, 604, 962-964
Append() method, 260, 462, 758
AppendFormat() method, 32-33, 75-77, 514
Applicable calls, 270-271, 417-418, 1078
ApplicationException type
- __arglist keyword, 19, 258, 601-603

args parameter for Main(), 21-22, 243-245, 258
ArgumentException type
- description, 651, 656, 664
- nameof operator, 289, 603, 894
- properties, 333, 802, 901-903
- throwing exceptions, 600-603, 612-614, 622

ArgumentNullException type
- description, 603, 654-656, 664
- properties, 347-350, 603, 802

ArgumentOutOfRangeException type, 600-603, 612-614, 916
Arguments
- generics, 623, 664, 677
- methods, 220, 224, 269-270
- named, 224, 269-270, 289
- passed by value, 224, 269, 488-489
- passing methods as, 220, 224, 260

Arithmetic operations, 138-143, 148-151, 184
Arithmetic operators, 138-143, 149, 167
ArithmeticException type, 600-605, 622, 916
Arity
- generic methods, 642-645, 664, 676-677

tuples, 108, 642-645, 664
type parameters, 642-645, 664, 677-678
Array accessors, 112, 123-130, 320
Arrays
anonymous types, 705, 797-805
assigning, 112, 115-120, 123-124
collection conversion to, 339, 754-759, 835-837
common errors, 114, 124, 132
declaring, 112, 115-120, 193
foreach loops, 194-195, 760, 1023-1025
forward accessing, 112, 120-126, 130
instance members, 129-130, 302, 306
instantiating and assigning, 116-120, 299, 338-339
length, 115-116, 122-125, 129-130
methods, 115, 124, 128-130
null-conditional operators, 82, 98, 173-178
overview, 112-116, 123-126, 856
parameters, 115-118, 224, 259-260
ranges, 112, 115-116, 124-128
redimensioning, 115-120, 123-124, 130-132
reverse accessing, 112, 121-127, 130-131
strings as, 115-116, 143, 1033
unsafe covariance, 669-676, 708
working with, 112, 115, 123-124
ArrayTypeMismatchException type, 124, 675, 1098
as operator
async, 985, 989-991, 1004-1008
Main() method, 21-22, 243-246, 258
return types, 335, 439, 554
ASP.NET, 11, 44-47, 1128
AsParallel() operator, 771-772, 1023-1024, 1031-1039
AspNetSynchronizationContext type, 944, 1013-1015, 1053-1054
Assemblies
attributes, 881, 897-899, 905-907
APIs, 43-45, 898-899, 1122-1123
boundaries, 557-559, 898-899, 1122-1123
building, 38, 898-899, 1122-1123
CIL, 38-44, 1114, 1124
CLI, 557-560, 1109, 1121-1123
constants, 373, 898-899, 1122-1123
extensions, 371, 898-899, 1122-1123
references, 557-560, 898-899, 1122-1123
AssemblyCopyrightAttribute, 897-899, 903-908, 914
AssemblyFileVersionAttribute, 897-900, 905-908, 914
AssemblyVersionAttribute, 897-899, 905-908, 912-915
Assert() method, 147, 462, 916-918
Assigning
arrays, 112, 115-120, 123-124
nulls to references, 82-83, 96, 100-101
nulls to strings, 82-83, 173, 176
pointers, 95-96, 1097-1099, 1102-1103
tuples, 106-108, 359, 807
variables, 26-27, 106, 116
Assignment operators
associativity, 140-142, 149, 552
compound, 139-143, 149, 552
delegates, 149, 548, 736
null-coalescing, 98, 173-178, 286
overriding, 149-151, 548-549, 552
Associations
data with static members, 302, 363, 367
with instance fields, 302-307, 363, 372
Associativity of operators, 138-143, 149, 552
AsSpan, 463, 856-859, 1084
Asterisks (*)
comments, 34-36, 571-573, 955
compound assignment, 140-142, 149, 955
indirection operators, 153, 955, 1102-1103
multiplication, 138-142, 149, 955
overriding, 142, 216, 955
pointers, 955, 1096-1103
precedence, 140-142, 216, 955
async
Main() method, 21-22, 991, 1003-1005
return types, 990-994, 1004-1006, 1019
async / await support, 985, 989, 1004-1008
async operator, 989, 1000, 1004

async return types, 989-995, 1004-1006, 1019
async void methods, 975, 1002-1004, 1008
Async() method, 994-997, 1000-1005, 1074
async/await support, 985, 989, 1004-1008
AsyncEnumberable class, 766, 994-1001, 1004-1006
Asynchronous invocation, 984-985, 1003-1005, 1012
Asynchronous lambdas, 701-706, 1008, 1012
Asynchronous methods return types, 990-994, 1002-1006, 1012
Asynchronous pattern, 935, 975, 1074
Asynchronous tasks
  complexities from, 944-945, 975, 1012
  introduction, 935, 944-945, 975
  task continuation, 951-952, 975, 1074
  task exceptions, 958-959, 975, 1003-1005
  thread exceptions, 958, 964, 973-975
  TPL, 944-945, 973-979, 1025
AsyncState property, 994-997, 1000, 1074
At signs (@)
  identifiers, 17-19, 65-68, 139
  verbatim strings, 65-70, 73, 1103
Atomic operations in multithreading, 154, 940-942, 1061
AttachedToParent task continuation option, 951-957, 1026, 1031
Attributes
  custom, 881, 897, 900
  description, 881, 896-900, 906-908
  enum, 533-534, 542-544, 906-908
  FlagsAttribute, 543, 905-908, 911
  initializing, 118, 338-340, 346
  vs. interfaces, 443-445, 482, 897
  named parameters, 269-270, 895-897, 909
  nullable, 82-83, 97, 101-102
  predefined, 881, 896-897, 900
  properties, 333, 881, 896-897
  restricting, 394, 808-811, 897
  retrieving, 333, 881-883, 897
  System.ConditionalAttribute, 354, 900, 912-914
  System.ObsoleteAttribute, 900, 905-908, 914-915
AttributeTargets flag, 896-900, 906, 908
AttributeUsageAttribute class, 900, 905-908, 914
Automatically implemented properties
  read-only, 322, 374-375, 507
  reference type, 96, 333, 347
  structs, 320-322, 333, 510-511
  working with, 318-322, 333, 474
AutoResetEvent event, 746-747, 1003-1005, 1064-1068
Average() function, 248, 948, 1001
await operator
  iterating, 760, 806, 863-868
  task-based asynchronous pattern, 989, 1004, 1074
Await() method, 989-993, 1003-1006, 1074
AwaitAsync() method, 994-997, 1000-1006, 1074
AwaitWithCancellationAsync() method, 966-969, 1003-1005, 1038

B

Backslashes (\)
  escape sequence, 64, 65-68, 71-74
  as literals, 57, 65-66, 67-73
Backspace escape sequence, 64, 65-72, 160
Base Class Library (BCL)
  description, 1109, 1113-1114, 1125-1126
  purpose, 557, 1113-1114, 1124-1126
  type names, 50, 51-53, 1125-1126
Base class overriding
  constructor invocation, 344-345, 402, 407-410

members, 398, 407-410, 417-418
new modifier, 397-398, 402-406, 657
overview, 402, 407-410, 417
sealed modifier, 397-398, 402, 405-406
virtual modifier, 398-402, 405-408, 455

Base classes
derived object conversion to, 386-391, 410, 418
inheritance, 295-297, 385-386, 410
protected member access in, 393-394, 476, 566

Base types
casting between derived types, 297, 389-391, 402
classes, 134, 391, 410
description, 134, 391, 410

BCL (Base Class Library)
description, 1109, 1113-1114, 1124-1126
purpose, 557-558, 1113-1114, 1124-1126
type names, 50, 53, 567-568

Binary digits (bits), 54-55, 60-61, 182-185

Binary expression trees, 141, 716-719, 723-725

Binary operators
associativity, 139-142, 149, 552
compound assignment, 139-141, 149, 552
non-numeric types, 63, 143, 552
null-coalescing, 98, 173-178, 286
overriding, 167-169, 548, 552
overview, 137-138, 167, 184
relational and equality, 63, 167-169, 548

Binary values
conversions to, 55, 60-63, 182-185
floating-point, 54-56, 145-148, 182-184
literals, 57, 61-63, 182-187
shift operators, 170, 182-187, 202

BinaryExpression expression trees, 139-141, 716-720, 723-725

BinarySearch() method
arrays, 128-130, 685-687, 842
list searches, 687, 839, 842-848

BinaryTree<T> class
constraints, 646-651, 662, 667
index operators, 648-650, 667, 865-866
iterators, 865-866, 869, 872-874
parameters, 646-651, 667, 872

Binding
dynamic, 920-923, 926-931
extension methods, 371-372, 396, 461-463
late, 725-727, 920, 931-933
metadata, 881-883, 931, 1127

Bits (binary digits), 54-55, 60-61, 182-187

Bitwise operators
bits and bytes, 61, 170, 182-187
complement, 169-170, 183-187, 552
compound assignment, 149, 183-187, 552
enums, 169, 184-187, 540-542
for positions, 119, 183-187, 202
shift, 182, 183-187, 202
working with, 139, 170, 183-187

Block statements, 13, 162-164, 702

BlockingCollection<T> class, 832-834, 1068-1069, 1076

Boolean (bool) type
array initial values, 63, 118-120, 554
conversions, 63, 89, 554-555
flow control, 137, 166, 169-170
lambdas, 166, 554, 703
logical operators, 63, 166-170, 184
overview, 63, 166, 554
relational and equality operators, 63, 167-170, 554

Boolean expressions for if statements, 158, 166-171, 184

Boolean operators
logical, 63, 167-170, 184-185
overriding, 167-170, 184, 552-554

Bounds
arrays, 115-120, 124, 135
floating-point types, 53-56, 145-148

Boxing
avoiding, 281, 523-527, 545
generics for, 524-527, 677, 681
idiosyncrasies, 17, 523-527, 545
in loops, 188, 192-194, 524-527
overview, 517, 523-527, 545

Brand casing for namespaces, 50-51, 221, 568-570
Break() method
 break statements, 13, 200-201, 875
 lambda expressions, 697-702, 716, 875
 overview, 200-201, 875, 1031-1032
 switch statements, 197-201, 205, 230
 working with, 200-201, 875, 1031-1032
 yield break, 870, 875, 1032
Breaking parallel loops, 1020-1023, 1031-1032, 1035-1039
Brittle base class problem, 391-394, 402, 410
Bubble sorts, 687, 698-700, 710-711
byte type, 53, 64, 96
Bytes, 488-489, 579, 1098-1099

C

C language and C# syntax similarities, 14-15, 48-49, 150
C++ language vs
 arrays, 112, 115-117, 135
 Boolean conversions, 63, 166-169, 554
 buffer overflows, 84-86, 1077, 1120
 delete operator, 139, 150-151, 168
 deterministic destruction, 581, 1077, 1118
 equality checks, 63, 168, 548
 global methods, 218, 232-233, 684
 global variables and functions, 2, 164, 232-233
 header files, 2-3, 14-15, 1114
 implicit overriding, 398, 402-403, 455-457
 increment and decrement operators, 139, 149, 150-154
 local variable scope, 164-165, 233, 714
 Main(), 14, 20-22, 243-245
 methods called during construction, 333, 336, 341
 multiple inheritance, 294-297, 443, 599
 operand evaluation, 63, 137, 141-143
 operator-only statements, 137, 150, 168
 pointer declarations, 1077, 1098-1099, 1102-1103
 preprocessor directives, 207-214
 pure virtual functions, 401-403, 441-443, 599
 short type, 53, 84-85, 1125
 string concatenation, 32, 67-69, 143
 structs, 217, 487, 1083
 switch statements, 197-200, 422-423, 441
 syntax similarities, 14-15, 48, 1114
 Variant and var, 94, 102-103, 708
 void type, 63, 84, 94-96
C++ similarity
 importance, 17, 63, 557
 literal suffixes, 53, 59, 67-69
 suffixes, 53, 59, 264
Calculated properties, 79, 318-322, 333
Call sites, 246, 261, 923-925
Call stacks for methods, 245-246, 261, 624-626
Callback functions, 219, 737, 1090-1092
CallerMemberName parameter attribute, 270, 895, 918-919
Calling
 constructors, 334-338, 341, 344-345
 collection initializers, 339, 754-759, 805
 external functions, 219, 232-233, 1078-1080
 object initializers, 334, 338-341, 346
CallSite<T> type
 camelCase, 223, 678, 923-925
 description, 223, 678, 923-925
 parameter names, 635, 678, 923-925
 property names, 801, 804, 923-925
 tuples, 111, 642-645, 923-925
 variable names, 223, 798-799, 923-925
 "can do" relationships, 651, 662-663, 923-925
Cancel() method
 parallel iterations, 1027-1032, 1035, 1038
 PLINQ queries, 963-968, 1034-1035, 1038
 tasks, 963-969, 975, 1038
Canceling
 iterations, 863, 875, 1030-1032
 parallel loops, 1023, 1027-1032, 1035
 PLINQ queries, 963-965, 1035, 1038
 tasks, 963-969, 975, 1036

CancellationToken class
- asynchronous methods, 966-969, 997, 1038
- asynchronous streams, 966-969, 997, 1038
- overview, 963-969, 1027-1029, 1038
- PLINQ queries, 963-969, 1035, 1038

CancellationToken property, 966-967, 968-969, 1038
CancellationTokenSource class, 963-966, 968-969, 1027-1029
Capacity of lists, 790, 835-837, 843-845
Capture() method, 228, 605, 1074
Captured outer variables, 248, 710, 711-715
Carets (^)
- arrays, 115, 121-122, 170
- bitwise operators, 168, 170, 183-187
- compound assignment, 27, 149-151, 170
- index from end operator, 121-122, 125-127, 170
- logical operators, 168-169, 170, 184
- overriding, 121, 127, 170

Carriage returns
- escape sequence, 65-72, 78
- newlines, 23, 65-72, 78

Cartesian products
- inner joins, 780-784, 788-789, 828
- query expressions, 808-812, 828, 831

Case labels
- goto statements, 156-158, 198-200, 205-206
- switch statements, 197-200, 205-206, 422-423

Case sensitivity, 17-18, 59, 323
Casing
- identifier formats, 17-19, 53, 59-61
- local variables, 164-165, 710, 714

cast operator and casting
- base and derived types, 390-391, 439-440, 555
- defining, 85, 439-440, 555
- enums, 85, 421, 537-540
- generic methods, 85, 439-440, 555
- implicit conversions, 85, 390-392, 555
- in inheritance chains, 390-391, 439-440, 555
- overriding, 85, 439-440, 552-555
- overview, 85, 439-440, 555

Catch blocks
- exceptions, 275, 280-284, 605
- general, 280-282, 605, 609-610
- rethrowing exceptions, 275, 601, 605-610
- working with, 281-282, 601, 605

Catching exceptions, 280, 601, 605
cells member in arrays, 114-115, 129-130, 195
Central processing units (CPUs)
- description, 933-938, 1021, 1107
- multithreading, 934-938, 941, 973
- PLINQ queries, 933-935, 1021, 1114

Centralizing constructor initializers, 336-346, 367
Chains
- constructors, 338, 344, 735
- circular, 162, 735, 780
- delegates, 693-694, 727, 735
- expressions, 716-718, 724-725, 809
- inheritance, 385-388, 396, 418
- iterators, 863-866, 871-873, 878
- null-coalescing operator, 98, 173-178, 179
- queries, 777, 809, 823-828
- tasks, 935, 952, 975

Characters (char) types
- arithmetic operations, 64-66, 144, 149-151
- array initial values, 64-66, 118, 144
- description, 49, 64-66, 144

Checked conversions, 88-90, 290, 538
checked keyword, 15, 63, 166-169
CheckForOverflowUnderflow property, 86-87, 620, 621
Child types, 134, 293-299, 386
Church, 294-295, 385, 568-570
CIL, 38-41, 1114, 1124
Circular wait conditions in deadlocks, 940-943, 974, 1060-1062
Class definitions
- description, 295, 298-299, 385
- guidelines, 293-295, 298, 568-570

class keyword, 15, 19, 298-299
Classes
  abstract, 295, 410-411, 444-445
  access modifiers, 314-316, 330, 566-567
  associated data, 302, 363, 782
  concrete, 410-411, 482, 568
  constraints, 652, 655, 660-662
  constructors. See Constructors
  declaring, 19, 298, 568-570
  deconstructors, 295, 341, 410
  defined, 295, 298-299, 568
  delegates, 693, 726-728, 743
  encapsulation, 301, 314-317, 394
  extension methods, 371-372, 396, 461-463
  files for storing and loading, 298, 311, 557-558
  generics, 623, 631-635, 677
  hierarchy, 221, 295-296, 385-386
  inextensible, 376, 396, 461-463
  inheritance. See Inheritance
  instance fields, 302-306, 360, 363
  instance methods, 306, 363, 370-371
  instantiating, 299, 363, 568
  interfaces. See Interfaces
  libraries, 557-559, 568, 1126
  members, 302, 394, 475
  methods, 218-220, 291, 363
  multiple interfaces, 444-445, 461-463, 482
  multiple iterators in, 568, 863-865, 873
  nested, 221, 376-378, 568-570
  nullable attributes, 97, 354, 897
  object-oriented programming, 293-295, 298-299, 302
  overview, 134, 295, 385
  partial, 298, 379-384, 568
  properties. See Properties
  refactoring, 227, 291, 568
  sealed, 394, 397, 406
  static, 363, 367-370, 373
  static members. See Static members
  vs. structs, 487, 499-500, 545
  this keyword, 15, 298, 307-309
Clear() method, 761-763, 863, 1115-1118
CLI, 4-7, 887-889, 1106-1109
Clock speeds, 154, 933, 938-942
Clone() method, 338, 365, 508-509
Close() method, 312, 581-589, 763
Closed over outer variables, 248, 710, 711-713
Closures, 581-584, 589-591, 605
CLR (Common Language Runtime)
  description, 38-39, 1106-1109, 1114
  enums, 533, 537-538, 1126
  namespaces, 567-568, 1114, 1126
CLS (Common Language Specification)
  description, 39, 1114, 1125-1126
  libraries, 39, 1113-1114, 1124-1126
  metadata, 1114, 1125-1126, 1127
CLSCompliant attribute, 903, 914, 1126
CLU language, 39, 1114, 1124-1126
Clusters, 786, 809, 863
Code access security, 315-316, 1093-1095, 1119-1120
Code blocks
  error trapping, 274-275, 281-282, 619
  if statements, 156-163, 197, 205
  loops, 156, 162-163, 188-193
  scope, 162-163, 164-165, 702
Code editors for preprocessor directives, 4, 207-214, 572
Code readability, 17, 161-163, 227
CodeAnalysis class, 558, 709, 712
Cold tasks, 934-937, 947, 975
Collect() method, 754-757, 761, 832-834
Collections
  anonymous types, 754-756, 797, 800-807
  class categories, 754-756, 832-837, 845
  dictionaries, 832-836, 845-849
  foreach loops, 194, 759-760, 863-864
  IEnumerable<T>, 800, 832-834, 995-997
  indexers, 832-834, 854, 860-863
  initializers, 338-340, 754-759
  interfaces, 463, 754-756, 832-834
  iterators. See Iterators
  join operations. See Join operations
  linked lists, 463, 832-837, 863
  null values, 82, 173-175, 863
  overview, 113, 754-755, 832-837

queues, 832-837, 853-855
searching, 808-810, 832-837, 843
shared states, 858, 974, 1069
sorted, 779, 821, 832-841
sorting, 696-698, 821, 839-841
stacks, 624, 631-634, 853-855
standard query operators. See Standard query operators
thread synchronization, 1041, 1051, 1069
total ordering, 777-779, 821, 839-842
Collisions, 780, 852, 941
Colons (:)
conditional operators, 163, 166-172, 175
constraints, 106, 344, 656
constructors, 336-339, 344, 757
derived classes, 344, 387, 463
hexadecimal format, 59-61, 64-66, 344
labels, 106, 206, 344
COM threading model, 935-938, 973, 1075
Combine() method
delegates, 259-260, 653, 736
hash codes, 248, 259, 851-852
parameters, 248, 259-260, 653
COMException type, 284, 609-614, 617
Command-line arguments, 224, 243, 269
CommandLineInfo class, 886-890, 896, 1109
CommandLineRequiredAttribute, 269, 896, 900-908
CommandLineSwitchAliasAttribute, 889, 895-896, 900-908
CommandLineUtils package, 4-5, 886-889, 1109
Commas (,)
arrays, 112, 115-120, 135
attributes, 881, 897, 898-900
constraints, 115-116, 193, 656
enums, 533-537, 540-542, 1091
for loops, 115-116, 188, 192-194
interfaces, 443-445, 461-463, 482
methods, 228, 291, 897
parameters, 116, 228, 701
variable declarations, 106, 115-116, 193
Comments
guidelines, 17, 34-36, 571-574
overview, 34-36, 291, 571-573
XML, 34-37, 571-574
Common Intermediate Language (CIL)
anonymous types, 610, 797-798, 1124-1126
assemblies, 38-40, 1114, 1124-1126
boxing code, 524-526, 610, 1124
compilation to, 38-40, 1114, 1124
description, 38, 1114, 1124
events, 752, 1114, 1124
general catch blocks, 524, 610, 1124
generics representation, 677-678, 1114, 1124
ILDASM, 38, 1114, 1124-1126
indexers, 38, 1114, 1124
metadata, 1114, 1124, 1125-1126
outer variable implementation, 38, 713, 1124-1126
output, 38, 1114, 1124
properties, 332-333, 1114, 1124
purpose, 38, 1114, 1124
Common Language Infrastructure (CLI)
assemblies, 1106-1109, 1114, 1121-1126
Base Class Library, 1109, 1113-1114, 1124-1126
CIL, 1106-1109, 1114, 1124
CLS, 1106-1109, 1113-1114, 1124-1126
compilation to, 1114, 1121, 1124-1126
CTS, 1106-1109, 1114, 1124-1126
description, 39, 1106-1109, 1124-1126
encapsulation, 39, 1106-1109, 1124-1126
garbage collection, 39, 577, 1115-1118
implementations, 39, 1106-1109, 1124-1125
manifests, 39, 1109, 1124-1126
metadata, 39, 1106-1109, 1124-1126
Microsoft .NET Framework, 5, 1109-1114, 1124
modules, 39, 1109, 1124-1126
multicast delegates, 739, 1106-1109, 1124
namespaces, 1078, 1109, 1124-1126
.NET Core, 5, 1109-1114, 1124
.NET Native feature, 1109, 1124, 1128

.NET Standard version, 1106-1109, 1113-1114, 1124
overview, 39, 1106-1109, 1124-1125
P/Invoke, 1078, 1106-1109, 1124-1126
performance, 39, 1106-1109, 1124
platform portability, 1109, 1113-1114, 1124-1125
runtime, 39, 1106-1109, 1124
type safety, 39, 1106-1109, 1124-1126
Xamarin compiler, 1106-1109, 1113-1114, 1124
Common Language Runtime (CLR)
description, 38-39, 1106-1108, 1114-1116
enums, 537, 1114, 1126
namespaces, 44, 1106-1108, 1114
Common Language Specification (CLS)
description, 1106-1109, 1114, 1124-1126
libraries, 39, 1114, 1124-1126
metadata, 39, 1114, 1124-1126
Common Type System (CTS), 1107, 1125, 1126
Compacting objects in garbage collection, 576-578, 1100, 1115-1118
Compare() method, 515, 687, 850
CompareExchange() method, 515-518, 850-851, 1056
CompareTo() method
collection sorts, 647, 838-842, 850
interfaces, 455-456, 647, 850
tree sorts, 647-650, 698, 838-840
Comparing
binary tree nodes, 648-650, 724, 872-873
collections items, 808-810, 832-837, 850
compare/exchange pattern, 423-425, 433, 1056
floating-point types, 53-56, 63, 145-147
strings, 63, 168, 514
Comparison operators
overriding, 167-169, 548-549, 552
pointers, 167-168, 548-549, 1102
Compatibility between enum types, 487, 533-538, 542-545
Compilation
ahead of time, 207-208, 1114-1116, 1121
disassembling, 34, 38-44, 1121
to machine code, 38-40, 1114-1116, 1121
sample output, 6, 12, 562
source code, 3-4, 10-12, 1114
static vs. dynamic, 367, 926, 1114
CompilerGeneratedAttribute, 333, 900, 906-908
Complement operator (~), 138-139, 170, 183-187
Complexity of asynchronous requests, 975, 981-985, 1012
ComponentModel class, 295, 394, 497
Components in compilation, 207, 1114-1116, 1121-1123
Composite formatting
description, 32-33, 36, 75-77
patterns, 77, 424, 433-435
strings, 32-33, 75-77, 143
Compound assignment
bitwise, 149, 183-187, 552
mathematical operators, 139-142, 149, 552
overriding, 149, 552, 736
Compress() method, 445, 687, 1118
Concat() method for strings, 75, 143, 259-260
Concat() operator for queries, 143, 808-812, 825-828
Concatenating strings, 75-77, 104, 143
Concrete classes, 293-295, 298-299, 482
Concurrent collection classes, 832-837, 1069, 1076
Concurrent operations, 154, 937-942, 1061
ConcurrentBag<T> class, 634, 1068-1069, 1076
ConcurrentDictionary<TKey, TValue> class, 659, 832-836, 845-849
ConcurrentQueue<T> class, 631-634, 853-855, 1068-1069

ConcurrentStack<T> class, 624, 631-634, 853-855
Conditional clauses in catch blocks, 275, 280-282, 605-606
Conditional operators
  null. See Null-conditional operator
  overriding, 169-172, 548, 552-554
  working with, 137, 143, 166-171
Conditional types, 166, 169-172, 205
ConditionalAttribute, 354, 900-903, 912-916
ConditionalExpression expression trees, 166, 428, 716-725
Conditions
  for loops, 156, 166, 188-194
  if statements, 156-161, 166, 205
Consequence statements in if statements, 156-163, 166, 204-205
Console
  comments, 28, 34-36, 571-574
  input, 7, 28-31, 225
  newlines, 67, 70-72, 78
  output, 28, 31, 723
  round-trip formatting, 62, 75-77, 514
  string methods, 28-31, 75, 225
const keyword
  constants, 15, 19, 373
  encapsulation, 15, 19, 394
ConstantExpression expression trees, 154, 716-720, 723-725
Constants
  expressions and locals, 193, 373, 714-716
  pattern matching, 423-425, 433-438, 441
  switch statements, 197-200, 205-206, 422-424
Constructed struct types, 293, 487, 545-547
Constructor implementation, 407, 463, 832-834
Constructors
  attributes, 338, 897, 903
  base classes, 402, 410, 660
  chaining, 336-338, 341-345, 512
  collection initializers, 338-341, 754-759
  constraints, 337-338, 657, 660-662
  declaring, 298, 334-338, 344-345
  default, 337-338, 341-345, 510
  finalizers, 341, 581-584, 596
  generics, 623, 638, 660
  initializers, 334, 338-341, 344-346
  methods called during, 334-338, 341, 344-346
  new operator, 335, 511, 548
  non-nullable reference type properties, 346-350, 505, 510
  object initializers, 334, 338-341, 344-346
  overloading, 336-338, 341-345
  overview, 294-295, 336-338, 341
  propagating exceptions from, 280, 609, 622
  static, 359-360, 363, 366-369
  structs, 338, 495, 510-511
Contains() method
  list searches, 833, 842-848, 853-855
  stack searches, 624, 846-848, 853-855
ContainsGenericParameters property, 660, 677-678, 891-893
ContainsKey() method, 845-849, 852-855, 929
ContainsValue() method, 515-516, 842-846, 848-850
Context switches, 200, 428, 937-938
Contextual keywords, 15-16, 19, 811-813
Continuation of tasks, 935, 950-956, 975
continue statements
  lambda expressions, 697-702, 716, 725
  overview, 156, 193, 203-206
  switch statements, 197, 200, 204-206
  working with, 188, 193, 203-206
ContinueWith() method
  asynchronous requests, 951-953, 980-981, 1074
  synchronization context, 951-953, 1003-1005, 1015-1018
  tasks, 951-953, 980-982, 1003-1005
Contravariance
  delegates, 671-676, 707-708

generics, 660, 672-676, 708
type parameters, 660, 672-676, 708
Control flow, 156-158, 203-205, 950
Conversions
as operator, 392, 439-440, 555-556
to binary representation, 60-61, 182, 185
Boolean type, 63, 88-89, 554-555
boxing, 88, 523-527, 530-532
casts, 88-89, 391-392, 555
checked and unchecked, 86-89, 391-392, 620-621
collections to arrays, 682, 774-776, 790
covariant, 391-392, 672-674, 708
dynamic objects, 88-89, 555, 920
enums, 89, 536-539, 1091
explicit, 88, 391-392, 555-556
false return values, 88-90, 147, 554
implementing classes and interfaces, 391-392, 444-445, 457
implicit, 88, 390-392, 555-556
overriding operators, 392, 548, 552-555
overview, 88-89, 391-392, 523
type safety, 88-89, 391-392, 555
without casts, 84-85, 88, 555
Cooperative cancellation, 963-968, 1030, 1036-1038
Copying
arrays, 130-132, 508, 676
directories, 261-264, 365, 371
null values, 82, 96, 508-509
reference types, 93-96, 134, 488-492
variables, 492, 508, 530
CopyTo() method
collections, 365, 371, 833-837
directories, 261-266, 365, 371
Core .NET frameworks, 44-47, 483, 1109-1113
CoreCLR compiler, 44, 47, 1109-1116
Cores, 47, 933-938, 1110-1113
Count property for collections, 682, 773-776, 835-837
Count() method
description, 773-776, 987, 1043-1045
elements, 760, 773-776, 1043-1045
index-based collection, 760, 773-776, 839
CountdownEvent event, 1003-1005, 1068, 1074
CountOccurrencesAsync() method, 976-978, 981, 986-988
Covariance
in arrays, 120, 673-676, 708
delegates, 669-676, 707-708
generics, 669-677, 708
out modifier, 358, 671-674, 707-708
CPUs (central processing units)
description, 933-938, 1021, 1107
multithreading, 934-938, 941, 973
PLINQ queries, 933-934, 1021, 1034-1039
Create() method, 227-228, 310-311, 402
Cryptographer class, 316, 369-370, 998
.cs file extension, 3-4, 10, 264
CTS (Common Type System), 1107, 1125, 1126
Curly braces ({})
arrays, 115-116, 162-163, 757
classes, 24, 162-163, 570
code blocks, 24, 70-72, 162-163
code formatting, 24, 70-72, 162-163
collection initializers, 162-163, 339, 757
lambda expressions, 162-163, 697-699, 702-704
methods, 24, 162-163, 702
namespaces, 24, 162-163, 567-570
object initializers, 162-163, 338-339, 757
pattern matching, 162-163, 427-429, 433-435
properties, 24, 162-163, 757
string interpolation, 32, 70-71, 162-163
switch statements, 162-163, 197-200, 428
type definitions, 24, 162-163, 757
use of, 24, 70-71, 162-163
Current property for iterators, 760, 863-868, 878
CurrentDomain class
exceptions, 612-615, 618, 622
process exits, 590-591, 594-595, 962-964

CurrentId property for asynchronous tasks, 366-368, 991, 1013-1016
Custom attributes, 881, 896-907, 914
Custom conversions, 88-89, 391-392, 523
Custom exceptions, 280, 599-601, 612-615
Custom objects in dynamic programming, 293-295, 920, 926
Custom value types for structs, 487, 544-547, 1097

D

Data namespace, 221, 567-570, 782
Data persistence, 310-311, 314, 782
Data types, 25, 92-93, 134
DataBinder class, 410, 683, 922-923
Deadlocks
  avoiding, 941-943, 974, 1060-1062
  causes, 940-943, 974, 1060-1062
DEBUG preprocessor identifier, 17-19, 207-214
Debug() method, 514, 564, 723
Debugging
  preprocessor directives for, 207-214
  source code, 3-4, 12, 572
Decimal type, 53-58, 84, 144-145
Declaration spaces in code blocks, 24, 162-165, 569
Declaring
  arrays, 112, 115-120, 123
  await using statement, 989-993, 999, 1003-1005
  BinaryTree<T> class, 648-651, 667, 865-866
  class type constraints, 651-656, 660-663
  classes, 291-293, 298, 568-570
  constants, 269, 298, 373
  constructors, 298, 334-338, 344-345
  deconstructors, 341, 358-359, 512
  delegate types, 689-693, 707, 716
  enums, 533-542, 1091
  events, 728, 743-747, 750-753
  external methods, 228, 232-233, 291
  generic interfaces, 444-445, 635-638, 663
  generic methods, 651, 663-664, 892
  instance fields, 303-307, 360, 363
  interface constraints, 444-445, 651-652, 656
  local variables, 25-27, 164-165, 193
  Main() method, 20-22, 227, 243-246
  methods, 228, 291, 298
  namespaces, 221-222, 235-239, 568-570
  out variables, 26-27, 193, 291
  parameters, 228, 267-270, 701
  pointers, 95, 1096-1103
  properties, 318-322, 333, 367
  read-only automatically implemented properties, 326, 374-375, 506-507
  read-only fields, 326, 374-376, 506-507
  return types, 84, 228-231, 700
  static constructors, 359-360, 367-370, 373
  static fields, 360-363, 367-369, 373-374
  structs, 506, 547, 1083
  tuples, 106-108, 359, 803
  types from unmanaged structs, 654, 1083-1084, 1097
  variables, 25-27, 193, 298
Deconstructors, 341, 357-358, 512
Decrement operators
  description, 138-139, 150-154, 183
  loop example, 138-139, 150-154, 1044
  thread-safe, 150-154, 1044, 1071-1072
Decrement() method, 150-151, 1044, 1071-1072
Default constructors, 337-338, 341-346, 510
Default interface members
  vs. extension methods, 461-463, 481, 484-485
  versioning, 446, 464-467, 481-484
default operator for generics, 630, 651, 662-663
default sections in switch statements, 197-200, 205-206, 422-424
Default types, 134, 269, 640
Default values
  generics, 269, 630, 638-641
  structs, 269, 510, 640

DefaultIfEmpty() method, 269, 288, 640
Deferred execution
 LINQ, 775-777, 793-796, 806
 query expressions, 718-720, 777, 806
Defining
 abstract classes, 410-411, 414, 453
 attributes, 881, 896-900, 914
 calculated properties, 318, 322, 333
 cast operators, 85, 439-440, 555
 custom conversions, 88-89, 391-392, 555
 custom exceptions, 599-601, 612-615, 618
 enums, 533, 540-542, 1091
 finalizers, 341, 581-584, 596
 generic classes, 631-634, 677, 832-835
 generic constructors, 635-638, 660, 677
 generic methods, 623, 663-664, 892
 indexers, 832-836, 860-864
 iterators, 760, 863-868, 873
 namespaces, 221, 235, 567-570
 nested classes, 298, 376-380, 568-570
 partial classes, 379-381, 383-384, 568
 preprocessor symbols, 15, 207-210, 214
 publishers, 725-728, 743, 753
 stacks, 624-628, 631-634, 854
 static methods, 360, 363, 367-370
 subscriber methods, 727, 730-731, 753
Delay() method, 1003-1005, 1066-1068, 1074-1076
Delegate class, 683, 694, 753
Delegate constraints, 651-657, 660-663
delegate keyword, 716, 747, 750
Delegate type, 693, 726-728, 738
Delegates
 constraints, 693, 716, 743
 deferred execution, 693, 716, 727
 event notifications, 727, 737-739, 753
 events, 726-728, 743, 753
 vs. expression trees, 684, 693, 716-725
 function pointers to, 693, 727, 1090-1092
 instantiating, 693-694, 726-728, 738
 internals, 693-694, 716, 727
 introduction, 693-694, 716, 727
 invoking, 693-694, 727, 739
 method returns and pass-by-reference, 693-694, 727, 735
 nesting, 691-694, 716, 727
 null-conditional operators, 173-175, 178-180, 733
 overview, 693-694, 727, 743
 predicates, 690, 716, 767
 process exits, 590-591, 594-595, 735
 publish-subscribe pattern operators, 727, 743, 753
 statement lambdas for, 697-699, 705, 716
 structural equality, 691-694, 707, 716
 tasks, 693-694, 727, 945-946
 types, 690-694, 726-728, 738
 unsafe code, 693, 1093-1095, 1104
delete operator, 138-139, 143, 169-170
Deleting files, 583-584, 770, 808-810
Delimited comments, 34-36, 70, 571-574
Delimiters, 23, 573, 716
DenyChildAttach task continuation option, 951-958, 1026, 1031
Dependencies in libraries, 11, 557-559, 562-564
Dequeue() method, 624, 760-762, 853-855
Dereferencing
 description, 95, 491-492, 1098
 null-conditional operator for, 98-99, 173-178, 733
 null references, 83, 98-101, 173-175
 null values, 82-83, 98-100, 173-175
 pointers, 95, 492, 1097-1103
Derivation
 casting between base and derived types, 297, 389-391, 402
 custom conversions, 89, 389-392, 555
 extension methods, 371-372, 396-397, 461-463
 interfaces, 444, 456, 458-463
 overview, 389, 397, 418
 private access modifier, 315-316, 393-394, 566

protected access modifier, 315-316, 393-397, 566
sealed classes, 397, 406, 410
single inheritance, 296-297, 386, 396-397
from System.Object, 389-391, 418, 463

Derived members, 386, 418, 660

Derived objects converted to base class, 386-391, 410, 418

Derived types
casting between base types, 297, 389-391, 402
classes, 296-297, 386, 410
description, 296-297, 386, 410

DescriptionAttribute, 881, 895-897, 900

Deserialize() method, 463, 538, 668

Deterministic destruction, 581-584, 596, 1118

Deterministic finalization, 581-584, 588-591, 596

Deterministic resource cleanup, 583-584, 1087, 1115-1118

Diagnostics class
debugging with, 496, 514, 564
notifications, 496, 564, 737
nullable attributes, 101-102, 350, 354
processes, 496, 564, 1010

Diagramming interfaces, 444-445, 456-458, 461-464

Dictionaries
collection initializers, 339, 754-758, 845
interfaces, 463, 832-836, 845
key equality, 845-852
sorted, 832-836, 845-849, 852
thread synchronization, 941-942, 1041, 1061
working with, 680, 832-836, 845-848

Dictionary<TKey, TValue> class
collections, 832-836, 845, 854
working with, 659, 845-849, 852-854

Digit separators, 23, 59-61, 143

Dimensions for arrays, 115-120, 123-125, 129-130

Directives
dynamic, 207, 237-241, 920
preprocessor. See Preprocessor directives
strings, 67-69, 207, 235-240

Directories, 261-262, 770, 782

DirectoryInfo class
extension methods, 371-372, 461-463, 886-890
inner joins, 463, 780-784, 789

DirectoryInfoExtension class, 371-372, 396, 461-463

DisallowNull attribute, 286, 348-350, 354

Disambiguating multiple Main() methods, 218-219, 227, 243-247

Disassembling programs, 38-44, 557, 1122

Discards
out parameters, 251-253, 358, 701
tuples, 106-108, 807, 815

DispatcherTimer class, 242, 1011-1013, 1074-1076

Disposability of tasks, 933-937, 971-975, 1013

Dispose() method
collections, 583-584, 588-589, 763
finalization, 583-589, 595-596, 1087
resource cleanup, 583-589, 595, 1087

DisposeAsync() method, 584, 1003-1005, 1074

Distinct members in query expressions, 808-812, 823-824, 828-831

Distinct() operator
description, 548, 809, 828-829
query expressions, 808-812, 828-831

Division
binary operator, 138-141, 167, 184-185
compound assignment, 27, 139-141, 149
shift operators, 138-140, 149-151, 183-184
by zeros, 138-139, 145-147, 148

DllImport attribute, 881, 897-900, 914

DLLs (dynamic link libraries), 920, 1078-1080, 1114

do/while loops
  overview, 156, 188-194, 204
  working with, 188-194, 203-204
Documentation
  reflection, 35, 44, 881-882
  XML files, 10, 35-37, 571-574
Dollar signs ($) for string interpolation, 66-67, 70-74, 77
DotGNU Portable.NET compiler, 12, 44, 1128
dotnet CLI, 4-5, 1109-1112, 1128
Dotnet CLI
  code building and executing, 4-5, 10-12, 1109
  NuGet references, 5, 558-561, 1109-1112
  project and library references, 5, 557-562, 1109
  working with, 4-5, 47, 1109-1112
Double quotes (")
  escape sequence, 65-67, 70-74
  strings, 67, 70-74, 143
double type
  overview, 92-93, 134, 1125
  special characteristics, 53, 145, 1125
DownloadData() method, 580, 976-978, 981
DownloadDataTaskAsync() method, 975-978, 981, 984-988
DownloadString() method, 243-244, 976-978, 981
Drawing class, 295, 298-299, 832-834
Duck typing, 50, 799, 926
Duplicating interfaces, 443-445, 461-465, 637
dynamic data type principles and behaviors, 920-922, 926, 1125
Dynamic invocation
  delegates, 693, 716, 727
  members, 920-925, 930-931
  nameof operator, 289, 894, 930-931
Dynamic link libraries (DLLs), 557-559, 1078-1080, 1114
Dynamic programming
  binding, 725, 920-926, 930-931
  custom objects, 293-295, 920, 926-928
  description, 725, 920, 926
  overview, 725, 920, 926
  principles and behaviors, 294-295, 920, 926
  reflection, 920-922, 931, 1119
  vs. static compilation, 920-922, 926, 1114
DynamicInvoke() method, 733, 923-931, 1078
DynamicObject class, 295, 680, 920-931

E

Editors
  dotnet CLI, 4-5, 12, 1109-1112
  preprocessor directives, 207-214, 237
  source code, 3-4, 12, 562
  Visual Studio 2019. See Visual Studio 2019
Elements, counting, 760, 773, 1043-1045
else clauses, 160-163, 169, 193
Empty collections, 113, 754-756, 863
Empty property for events, 324, 431, 743-752
Empty strings, 81-82, 288, 514
Encapsulation
  classes, 294-295, 314-317, 394
  CLI, 5, 394, 1106-1109
  const modifier, 314-316, 372-375, 394
  information hiding, 301, 314-316, 394
  object-oriented programming, 293-295, 314, 394
  with protected interface members, 394, 476, 484
  publications, 294-295, 314-316, 394
  readonly modifier, 331, 374-376, 506-507
  subscriptions, 394, 743, 753
  well-formed types, 314, 394, 545-547
EncryptAsync() method, 994-1001, 1004, 1033
EndsWith() method, 122, 462, 842-844
Enqueue() method, 258, 314, 853-855

Enter() method with monitors, 528, 1046-1049, 1053
Enum class, 487, 533-542, 1091
enum values
  conversions, 532-534, 538, 1091
  flags, 533-534, 540-544, 910
  FlagsAttribute, 541-544, 906-911
  metadata, 533-535, 541-544, 1091
  overview, 487, 533-535, 1091
  parsing, 533-534, 538-539, 1091
  type compatibility, 487, 533-538, 1091
Enumerable class
  delegates, 463, 694, 742
  element counting, 760-761, 765-766, 773-776
  filters, 463, 769, 833-834
  inner joins, 463, 780-784, 788-789
  methods, 463, 766, 832-834
  parallel queries, 772, 833, 1033-1039
  query expressions, 463, 809-812, 831-833
  sorts, 778-779, 839, 1034
  standard query operators, 777, 806-810, 833
Enumerator patterns for iterators, 760, 863-868, 878
Environment class
  command-line arguments, 269, 886-889, 896
  newlines, 67, 70-72, 78
  thread synchronization, 941-942, 1046, 1076
Equality
  C# vs. C++, 63, 168, 515
  collections, 463, 755, 832-834
  delegates, 691-694, 707, 716
  dictionary keys, 832-834, 845-852
  floating-point types, 53-56, 145-148
  with null, 82, 173-175, 493
  objects, 167-168, 493-494, 515
  operators, 63, 167-170, 548
EqualityComparer<T> class, 515-516, 838-842, 850-852
Equals signs (=)
  compound assignment, 63, 149, 168
  delegates, 167-168, 548, 736
  equality operator, 63, 167-168, 548
  null checks, 98, 168, 173-175
  null-coalescing operator, 98, 168, 173-178
  overriding, 63, 167-168, 548
  pointers, 63, 168, 1102-1103
  precedence, 63, 140-142, 167-168
  properties, 168, 333, 515
  relational and equality operators, 63, 167-168, 548
  shift operators, 149, 167-170, 183-184
  variable assignment, 63, 149, 168
Equals() method
  collections, 515-518, 832-835, 850
  description, 168, 515-518, 548
  with GetHashCode(), 515-520, 850-852
  implementing, 515-518, 548, 850
  null values, 98, 173-175, 515-518
  overriding, 515-518, 548, 850
  reflection, 493-494, 515-518, 548
Equi-joins, 780-784, 788-789, 809
Error handling
  catch blocks, 275-276, 605, 610
  exception class inheritance, 280, 599-601, 605
  with exceptions, 280, 599-601, 622
  P/Invoke, 1077-1078, 1085-1092
  publish-subscribe pattern, 727, 737-743, 753
  throw statements, 284-285, 601, 607-608
  trapping, 274, 599-601, 604-605
Errors
  array coding, 114-115, 124, 132
  preprocessor directives for, 207-214
Escape sequences, 65-67, 160-163, 204
Essential source code, 3-4, 12, 562
EssentialCSharp.sln file, 4, 10-14, 245
Etch A Sketch game, 119, 158-160, 196
Evaluation of operators, 141-143, 167-169, 548
event keyword, 743-750, 751-753
Event notifications, 737, 743-747, 750-753
EventArgs class, 181, 463, 743-752

EventHandler<T> class, 726-728, 743, 746-753
Events
- coding conventions, 743-744, 747, 750-753
- declaring, 743-747, 750-753
- generics and delegates, 726-728, 743, 747
- implementation, 444, 743-744, 753
- internals, 737, 743-747, 750-753
- listeners, 743-744, 747, 753
- overview, 743-744, 750, 753
- publication encapsulation, 743, 746-747, 753
- publish-subscribe pattern. See Publish-subscribe pattern
- purpose, 727, 743-744, 747
- subscription encapsulation, 727, 743, 753
- thread synchronization, 932, 973, 1041

Exception conditions, 601-606, 612-614, 622
Exceptional continuation, 204, 825-827, 950-952
ExceptionDispatchInfo class
- asynchronous requests, 608, 618, 1074
- purpose, 599, 608, 618

Exceptions
- catch blocks, 275, 601, 605
- class inheritance, 280, 599-601, 605
- custom, 280, 599-601, 612-615
- error trapping, 274-275, 280, 601
- expected situations, 280, 599-601, 605
- guidelines, 599-601, 612-614, 622
- overview, 280, 599-601, 622
- parallel loops, 1021-1023, 1028, 1035-1039
- PLINQ queries, 796, 825-827, 1034-1035
- propagating from constructors, 336-338, 341, 658
- publish-subscribe pattern, 727, 743, 753
- rethrowing, 601, 607-609, 622
- serializable, 461-463, 599, 617
- tasks, 599-601, 958, 975
- threads, 937, 962-964, 972-973
- throw statements, 283-284, 601, 607-609
- types, 280, 601, 622

ExceptionServices class, 461-463, 599-601, 617
Exchange() method, 463, 870-871, 1056
Exclamation points (!)
- equality operators, 167-170, 175, 548
- generics, 66-67, 638, 678
- inequality operators, 143, 167-170, 548
- logical operators, 143, 166-171, 184
- null-forgiving operators, 98, 168-170, 173-175
- overriding, 66-67, 143, 168-170
- pointers, 66-67, 168-170, 1102-1103

Excluding code, 161-163, 208, 213
Exclusive OR (XOR) operators
- bitwise, 168-169, 170, 184-187
- logical, 167-169, 170, 184

Executables, 12, 557, 1121-1122
ExecuteSynchronously task continuation option, 951-953, 957, 1003-1005
Execution time, 225, 933, 939-941
ExecutionEngineException type, 604-605, 613-618, 622
Exit() method with monitors, 528, 1047-1049, 1053
Expanded form for parameters, 228, 259-260, 701
Explicit casts
- base and derived types, 389-391, 457, 652
- enums, 536-539, 542, 885
- guidelines, 85, 390-392, 457
- overview, 85, 390-391, 555

Explicit deterministic resource cleanup, 583-584, 595-596, 1087
Explicit member implementation, 454-457, 476, 485
Exponential notation, 58-61, 150, 182
Expression bodied members
- declaring, 231, 697-699, 716-718
- properties, 231, 333, 522

Expression bodied methods, 231, 716-719, 725

Expression trees
- deferred execution, 716-720, 723-725, 775-777
- vs. delegates, 684, 693, 716-725
- description, 716-720, 723-725, 831
- examining, 716-720, 723-725, 831
- object graphs, 716-720, 723-725, 831
- SQL queries, 718, 724-725, 831

Expressions
- constant, 154-155, 373, 425
- lambda. See Lambda expressions
- query. See Query expressions

Extended Application Markup Language (XAML), 35-37, 572, 1109

Extensible Markup Language, 36, 37, 572-574

Extension methods
- classes, 371-372, 396, 461-463
- vs. default interface members, 396, 461-463, 481
- derivation, 371-372, 396, 461-463
- interfaces, 396, 444-445, 461-463

Extensions, 371-372, 396, 461-463

External methods and functions
- calling, 253, 1078-1080, 1086
- declaring, 232-233, 291, 1078

F

f-reachable (finalization) queues, 582-584, 588, 595-596

Factory interfaces for constructor constraints , 651-652, 656-662, 673-674

FailFast() method, 604, 622, 1074

Failure type, 604-605, 619, 802

false operator, 63, 166-171, 554

Fat arrow notation (=>) for lambda expressions, 697-706, 716-718, 1008

FCL (Framework Class Library), 557-558, 568, 1109-1114

Fibonacci numbers, 188, 189, 1020-1022

Fields
- guidelines, 302-303, 375, 521
- instance. See Instance fields
- static members, 360-363, 367-370, 373
- strings, 80, 115-116, 143
- thread synchronization, 1041, 1046, 1061

FIFO (first in, 142, 189, 624

FileAttributes class, 896-900, 906-908, 911

FileInfo class
- inner joins, 770, 782-784, 814-815
- projections, 770, 808-810, 814-815
- query expressions, 808-810, 814-815, 821-822

Files
- deleting, 583-584, 770, 808-810
- extensions, 10, 264, 770
- Java language vs. C#, 1-3, 14, 1114
- project, 10, 558-562, 808-810
- storing and loading with, 311-314, 557, 782

FileStream class
- finalizers, 312, 583-589, 596
- IAsyncDisposable, 583, 586-589, 997-999
- storing and loading files, 311-312, 597-598, 770
- ToString(), 311, 513-514, 532

Filters
- Browse, 770, 808-812, 843
- collections, 113, 682, 808-810
- exceptions, 280, 605, 808-810
- query expressions, 718, 808-812, 820

Finalization, 581-584, 589-591, 595-596

finalization (f-reachable) queues, 581-584, 588-590, 595-596

finalize() method, 341, 581-584, 595-596

Finalizers
- constructors, 341, 581-584, 589-590
- generics, 341, 581-584, 595-596
- resource cleanup, 582-584, 589-590, 595-596
- structs, 341, 511, 581-584

finally blocks in error trapping, 279-282, 604-605, 1047

FindAll() method, 770, 794, 844

First in, 6, 639, 812

fixed statement for pointers, 96, 491-492, 1096-1102
Flags, 463, 540-544, 906-910
FlagsAttribute
- enums, 540-543, 906-911
- overview, 897, 905-908, 911

Flatten() method, 769-770, 782, 800
Flattening sequences in query expressions, 808-810, 823-828, 831
float type
- overview, 53, 56, 145
- special characteristics, 53-56, 92, 145-148

Floating-point types
- equality issues, 53-56, 145-148
- overview, 53-56, 92, 145-148
- special characteristics, 53-56, 92, 145-148

Flow control
- Boolean expressions, 137, 166, 169
- code blocks, 156, 204, 950
- description, 156, 204, 950
- for loops, 188, 192-193, 204
- foreach loops, 194-195, 204, 1023-1025
- if statements, 156-163, 166, 204-205
- introduction, 137, 156, 950
- jump statements, 156, 200, 204-206
- preprocessor directives, 137, 207-214
- scope, 156, 164, 950
- switch statements, 156, 197-200, 204-206
- task continuation, 204, 950-955, 975
- threads, 936-938, 964, 973
- while and do/while loops, 156, 188-193, 203-204

for loops
- overview, 188-194, 760, 1021
- parallel iterations, 193, 1020-1023, 1035-1039

For() method, 190-194, 760, 1023-1024
Forcing resource cleanup, 576-577, 583-584, 1115-1118
foreach loops
- arrays, 192-195, 760, 1023
- async streams, 994-999, 1023-1025, 1035
- collections, 194, 759-761, 863-864
- enumerator patterns, 200, 760, 863-864
- IEnumerable<T>, 760, 765-766, 869-871
- overview, 192-194, 760, 863-864
- parallel iterations, 1020-1025, 1035-1039

Form feeds escape sequence, 65-73, 160, 204
Formal parameter lists, 228, 664, 701
Format items, 75-77, 843, 897
Format strings, 31-33, 75-77, 143
Format() method, 32-33, 75-77, 513-514
FormatException type, 84, 280-282, 619
FormatMessage() method, 32, 75-77, 514
Formatting
- hexadecimal format, 60-61, 66, 77
- round-trip, 62, 65-67, 77
- with string interpolation, 32-33, 67, 74-77
- strings, 32-33, 75-77, 143

Forward accessing arrays, 112, 120-126, 130-132
Forward slashes (/)
- command-line option, 269, 437, 896
- compound assignment, 67, 106, 149
- overriding, 59, 65-68, 573
- precedence, 65-67, 142, 216
- XML comments, 35-37, 571-572, 573-574
- XML elements, 36-37, 571-572, 573-574

Fragile base class problem, 397-398, 402, 410
Framework Class Library (FCL), 557-558, 1109-1114, 1126
Framework Design Guidelines, 17-18, 521, 1110-1113
Frameworks
- APIs, 45, 1078, 1110-1113
- .NET. See .NET frameworks

from keyword in query expressions, 808-813, 823, 831
FromCurrentSynchronizationContext() method, 1011-1015, 1051-1054, 1076
Full outer joins, 780-784, 789, 809
Fully qualified method names, 50, 228, 895
Func delegate, 688-693, 716, 727

Functions
- external, 1007, 1078-1080, 1086
- pointers to delegates, 689-693, 727, 1090-1092

G

Garbage collection
- CLI, 576-578, 584, 1115-1118
- managed execution, 39, 576-578, 1115-1118
- Microsoft .NET framework, 44, 576-578, 1115-1118
- objects, 576-578, 596, 1115-1118
- overview, 576-578, 595, 1115-1118
- resource cleanup. See Resource cleanup
- weak references, 576-577, 578, 1115-1118

GC class
- finalization, 582-584, 596, 1118
- finalizers, 341, 582-584, 595-596
- garbage collection timing, 576-578, 1100, 1115-1118
- resource cleanup, 576-578, 595, 1115-1118

General catch blocks, 280-282, 605-606, 609-610

Generics
- benefits, 623, 631-635, 677
- CIL representation, 630, 677-678, 709
- classes, 623, 631-634, 681
- constraints. See Constraints
- constructors, 623, 638, 677
- covariance and contravariance, 669-676, 708
- default values, 630, 638-640, 677
- defining, 623, 631-635, 677
- delegates, 690, 693-694, 726-728
- events, 623, 677, 726-728
- examples without, 638, 677, 681
- finalizers, 581-584, 590, 595-596
- instantiating, 623, 638, 677-681
- interfaces and structs, 444-445, 636-638, 651
- internals, 631-633, 646, 677-681
- introduction, 623, 677, 681
- Java, 623, 630-633, 638
- lazy loading, 638, 677, 680-681
- linked lists, 624, 631-638, 835-837
- methods, 623, 664, 681
- nested, 631-633, 638, 643-646
- overview, 623, 677, 681
- reflection, 677, 882-883, 892
- tuples, 108, 642-645, 803
- type parameters, 631-635, 664, 677
- working with, 623, 630-634, 677

get keyword for properties, 309, 318-320, 333

GetAsyncEnumerator() method, 766, 994-997, 1000

GetAwaiter() method, 989-997, 1000, 1003-1006

GetCommandLineArgs() method, 243-244, 269, 886-890

GetCustomAttributes() method, 901-903, 914, 931

GetEnumerator() method
- duck typing, 765-766, 864, 868-871
- iterators, 760, 864-871, 878
- purpose, 760, 863-865, 868-870
- queryable extensions, 766, 833, 868
- standard query operators, 766, 833, 868

GetFiles() method
- directories, 261-264, 770, 821
- inner joins, 770, 782-784, 814

GetGenericArguments() method, 664-665, 677, 891-893

GetHashCode() method
- collections, 519-520, 832-834, 851-852
- description, 418, 516-520, 851-852
- overriding, 408, 516-520, 851-852
- value types, 494, 516-520, 850-852

GetLastError() method, 383, 1079-1080, 1085-1087

GetLength() method, 79, 125, 129

GetMethods() method, 332-333, 829, 884

GetProperties() method, 79, 333, 883-884

GetResult() method
- getters, 253, 332, 1006
- access modifiers, 315-316, 330, 566
- properties, 79, 884, 949

GetType() method

description, 883-885, 888-893
null values, 99-100, 356, 884-885
objects, 421, 883-885, 891-893
reflection, 883-885, 888-893
GetValue() method, 329, 532, 888-890
GhostDoc tool, 4-7, 576, 1118
Global methods, 113, 237-238, 664
Global variables and functions, 232-233, 237, 710
goes to operator, 137-139, 143, 439
goto statements
lambda expressions, 697-705, 716, 725
overview, 156, 205-206, 246-247
switch statements, 197-200, 205-206, 422
Graphs for expression trees, 716-720, 723-725, 831
Greater than signs (>)
generics, 663-664, 677-678, 698
overriding, 139, 167-169, 698
pointers, 139, 167-170, 1102-1103
referent member access, 139, 167-169, 1102-1103
relational operators, 138-139, 167, 168-170
shift operators, 139, 167-170, 183
XML, 35-37, 167-169, 698
group by clause, 786, 809-812, 823-828
group clauses in query expressions, 809-812, 823-828, 831
GroupBy() method, 786-788, 809, 823-826
Grouping query expressions, 808-811, 823-828, 831
GroupJoin() method
inner joins, 782-789, 809, 826
outer joins, 780-789, 809

H

Handle() method
asynchronous requests, 975, 1003-1005, 1074
task exceptions, 274, 604, 958-961
Handlers for process exits, 589-591, 594-595, 1010
Hardcoding values, 57-60, 94-96, 373
HasFlags() method, 540-544, 888, 906-911
Hash codes and tables
collection equality, 167, 515-519, 850-852
dictionaries, 521, 845-847, 850-852
GetHashCode(), 418, 516-520, 851-852
Hash symbols (#) for preprocessor directives , 19, 66, 207-214
Hat operator (^) for arrays, 121-122, 125-127, 170
Header files, 37, 207-210, 239
Heap
boxing, 523-528, 1100, 1118
new operator, 300, 335, 511
reference types, 95-96, 488-492, 1097
HelloWorld program
assemblies, 12, 557-561, 1122
CIL output, 38-40, 1114, 1124
compiling, 2, 12-14, 557-559
debugging, 2, 12-14, 557-561
description, 2, 20, 557-561
dotnet CLI for, 4-5, 10-12, 557-561
preparing, 2, 20, 557
project files, 10-14, 20, 557-561
stand-alone, 2, 12-14, 557-559
Visual Studio 2019 for, 2-4, 7, 12
HelloWorld.dll file, 10-12, 557-561, 1122
Hexadecimal numbers
formatting numbers as, 59-61, 66, 182
notation, 60-61, 66, 182-184
Unicode characters, 60-61, 66, 144
HideScheduler task continuation option, 970-971, 1011-1013, 1031
Hierarchy of classes, 295-298, 385-386, 568-570
High-latency operations
async/await syntax, 985, 991, 1004
asynchronous example, 933-935, 975, 1074
synchronous example, 933, 939-940, 975

WPF example, 933-934, 1016, 1074
Hill climbing, 193, 1025, 1119
Hold and wait conditions in deadlocks, 941-943, 974, 1060-1062
Hooking up publishers and subscribers, 726-728, 731-732, 743
Horizontal tabs escape sequence, 65, 66-67, 160
Hot tasks, 944, 947, 973-975
Hungarian notation, 53, 59-61, 66
Hyper-Threading, 932-938, 941, 972-973
Hyphens (-), 17-19, 59, 138-139

I

I/O-bound operations
latency, 933, 934, 937-941
performance considerations, 524, 933-934, 1120
IAsyncDisposable interface, 584, 763-764, 995-1000
IAsyncEnumerable<T> interface
async returns, 994-997, 1000, 1004-1006
asynchronous streams, 793-795, 994-997, 998-1000
LINQ with, 793-796, 995-997, 1000
IAsyncEnumerator<string> interface, 763, 994-997, 1000
ICollection<T> interface
collection initializers, 339, 754-759, 832-834
element counting, 759-760, 765, 773
members, 463, 759, 832-836
IComparable<string> interface, 463, 838-840, 850
IComparable<T> interface, 647-650, 667, 838-842
IComparer<T> interface, 647-650, 838-842, 850
Id property for asynchronous tasks, 333, 367-368, 1014
IDE (integrated development environment)
debugging support, 7, 209, 572
Visual Studio 2019, 1-4, 7, 572
Identifiers
guidelines, 17-19, 298, 895
keywords as, 15-19, 298, 811-813
overview, 17-19, 113, 567
Identity of objects, 295, 299-302, 782
IDictionary<TKey, TValue> interface, 656, 832-836, 845-849
IDisposable interface
resource cleanup, 583-584, 763-764, 1087
tasks, 584, 763-764, 971
IDynamicMetaObjectProvider interface, 458-459, 920, 928-931
IEnumerable<T> interface
collections, 463, 759-760, 832-834
delegates, 463, 832-834, 865
element counting, 760-762, 765, 773
filters, 463, 769, 832-834
foreach loops, 760, 763-765, 868
iterators, 760, 863-868, 871
PLINQ queries, 463, 793-796, 1034-1035
projections, 463, 800, 832-834
query expressions, 463, 793-796, 833
standard query operators, 463, 769, 833
task-based asynchronous pattern, 975, 994-997, 1074
yield statement, 866-871, 875-879
IEnumerator<T> interface
foreach loops, 760, 763-765, 868
iterators, 863-866, 871, 995-997
yield statement, 866-870, 875-879, 994
IEqualityComparer<T> interface, 515-516, 838-840, 850-851
IEquatable<T> interface, 515-516, 793-795, 850
if statements
Boolean expressions, 158, 166, 168-171
code blocks, 156-163, 166, 205
continue statement replacement, 156-161, 203-206

- nested, 158-163, 166, 205
- overview, 156-161, 197, 205
- switch statements for, 197-200, 205-206, 428
- working with, 158-162, 166, 205

IFileCompression interface, 444, 445, 456-463

IGrouping type, 463, 786-788, 823-826

ILDASM (IL Disassembler), 38-40, 41, 43-44

IList<T> interface, 463, 832-837, 854

IList<TKey> interface, 832-837, 849, 854

IList<TValue> interface, 463, 832-837, 854

Immutability
- anonymous types, 505-507, 797-798, 801-805
- delegates, 505-507, 707, 716
- strings, 28, 67-69, 80-81
- value types, 96, 488-489, 505-507

Immutable library, 375, 524, 832-834

Implementing class conversions with interfaces, 444-445, 451-453, 456-457

Implicit conversions
- base and derived types, 389-391, 402, 652
- guidelines, 88-89, 391-392, 457
- types, 88-89, 390-392, 555

Implicit deterministic resource cleanup, 583-584, 595-596, 1087

Implicit member implementation, 455-458, 476, 485

implicit operator for types, 391, 439, 554-555

Implicitly typed local variables, 102, 714, 798-799

import directive, 207-209, 213-214, 235-239

in modifier, 315-316, 351, 405

in parameters, 224, 228, 701

Including code, 4, 298, 572

Increment operators
- description, 139, 150-154, 1057
- prefix and postfix, 139-142, 149-153, 154
- thread-safe, 154, 1043-1045, 1057

Increment() method, 150-154, 1044, 1057

Indenting code
- if statements, 24, 158-163, 166
- readability, 24, 36, 160-163

Index operator
- lists, 122, 126, 860-861
- variable parameters, 112, 153, 860-861

Index type, 112, 126, 860-861

IndexerNameAttribute, 860, 861, 895-899

Indexers, 808-810, 833, 860-864

Indexes
- arrays, 115-116, 120, 123-128
- dictionaries, 832-836, 845-849
- list searches, 833, 843, 860
- sorted collections, 113, 682, 832-837

IndexOf() method, 121-122, 760, 842

IndexOutOfRangeException type, 126-127, 600-603, 760

Indirection operators for pointers, 511, 1097-1102, 1103

Inequality
- floating-point types, 53-56, 145-148
- with null, 82, 98-99, 173-175
- operators, 138-139, 167-170, 548

Inextensible classes, 396-397, 461-463, 482

Inferences for generic methods, 651, 660-666, 892

Infinite recursion errors, 246, 261-263, 802

Infinity value, 54-57, 139, 148

Infix notation for binary operators, 138-142, 149, 153

Information hiding in encapsulation, 314-316, 394, 476

Inheritance
- abstract classes, 385-386, 410-411, 414
- base class overriding. See Base class overriding

- casting in, 85, 386, 390-391
- constraints, 386, 656, 660-662
- definitions, 294-297, 385-386, 599
- derivation. See Derivation
- exception classes, 280, 599-601, 605
- interfaces, 443-445, 458, 461-463
- is operator, 386, 548, 599
- overview, 294-297, 385-386, 599
- System.Object class in, 294-295, 299, 418
- value types, 96, 134, 487-489

Initial section in for loops, 188-195, 863, 1021-1024

Initialization, 338-341, 346, 367

Initializers
- collections, 339, 754-759, 832-834
- constructors, 334, 338-341, 344-346
- object, 338-341, 346, 757

Initializing
- anonymous type arrays, 118, 754, 797-805
- arrays, 112, 115-120, 135
- attributes, 881, 896-900, 903
- dictionaries, 680, 832-834, 845-849
- fields, 334, 338-340, 346
- loop counters values, 150, 193, 1043-1045
- static members, 363, 367-370, 373
- structs, 338, 510-511, 1083-1084

Inner classes, 376-378, 394, 475

Inner joins
- description, 780-784, 788-789, 809
- one-to-many relationships, 780-784, 788-789, 809
- performing, 780-784, 788-789, 809

InnerException property, 616, 617-618, 960

InnerExceptions property
- aggregate exceptions, 616-618, 959-960, 1028
- asynchronous requests, 616-618, 958-960, 1028
- error handling, 601, 616-618, 960
- parallel loops, 960, 1028-1032, 1035-1039
- tasks, 616-618, 958-961, 1028

INotifyCompletion interface, 461-463, 1003, 1006

INotifyPropertyChanged interface, 181, 463, 732

Input, 28-31, 199, 227-228

Insert() method, 305, 463, 758

Inserting dictionary items, 758, 832-836, 845-849

Instance fields
- accessing, 306, 330, 338
- declaring, 302-304, 307, 360
- description, 302-303, 338, 363
- initializing, 338-340, 346, 367

Instance members
- arrays, 129-130, 302, 790
- strings, 69, 129, 514
- working with, 302, 305-307, 475

Instantiation
- arrays, 115-120, 129, 790
- classes, 298-299, 410, 568
- defined, 299, 338, 360
- delegates, 683, 693-694, 727
- generics, 623, 677-679, 892

Integers and int type
- characteristics, 56, 96, 144-145
- characters, 64-66, 87-88, 144
- conversions, 56, 84-88, 144
- default, 56-57, 87, 640
- description, 56-57, 84, 144
- enums, 487, 533-535, 1091
- overflow, 55-56, 86-88, 620
- vs. strings, 84, 88, 144

Integrated development environment (IDE)
- debugging support, 4, 7, 572
- Visual Studio 2019, 1-4, 7, 572

IntelliSense feature, 44, 572, 812

Interlocked class
- increment and decrement operators, 150-151, 154, 1056-1057
- working with, 1056, 1061, 1076

internal access modifier
- description, 315-316, 384, 565-566
- interface refactoring, 445-446, 455, 465
- on type declarations, 316, 565-567, 1119

Interoperability, 441-445, 463, 1119-1120

InteropServices class
SafeHandle, 1077-1081, 1087-1088, 1090-1092
unmanaged exceptions, 610, 1077-1078, 1085-1087
Interpolation of strings, 31-32, 67-71, 143
Intersect() operator, 167-170, 771, 780
into clause in query expressions, 809-812, 823-827, 831
IntPtr class, 96, 523-524, 1081
InvalidAddressException type, 612-614, 618-619, 1098
InvalidCastException type
description, 84-85, 390-391, 647-651
generics, 630, 638, 649-651
using, 85, 391, 648-651
InvalidOperationException type
catch blocks, 280-282, 605-606, 609-610
collections, 754-756, 759, 832-835
description, 649-651, 656, 662
null values, 99-100, 628-630, 656
tasks, 802, 975, 994
wrapped exceptions, 612-614, 618, 622
Invocation
base class constructors, 402, 410, 660
delegates, 693, 716, 727
dynamic, 716, 920-926, 931
referenced packages and projects, 557-562, 568, 1078
using statement, 702, 705, 716
Invoke() method
delegates, 733, 1078, 1090-1092
dynamic invocation, 733, 930-931, 1078
events, 727, 733, 753
IOException type, 466, 482, 617-619
IOrderedEnumerable<T> interface, 463, 833, 995-997
IProducerConsumerCollection<T> interface, 832-834, 995-997, 1069
IQueryable<T> interface
delegates, 463, 793-796, 833
query expressions, 793-796, 830, 833
queryable extensions, 463, 793-796, 833
is a kind of relationships, 295-297, 385-386, 781-782
is a relationship
abstract members, 302, 410-414, 456
classes, 295-296, 302, 385-386
derivation forms in, 296, 385-386, 389
IsCancellationRequested property, 963-969, 1030, 1038
IsCompleted property
multithreading, 940, 1043-1046, 1051
parallel loops, 1024, 1027-1032, 1035
IsDefined() method, 269, 463, 515
ISerializable interface, 456-458, 461-463, 617
IsGenericType property, 891, 892, 893
IsInvalid member, 324-325, 353, 440
Iterations
loops, 188, 192-194, 1021-1023
parallel. See Parallel iterations
Iterators
compiling, 765, 863-868, 877-878
defining, 863-868, 871-873, 878
escaping, 863-868, 871-873, 878
multiple, 760, 863-866, 871-873
origin, 863-868, 871-873, 878
recursive, 863-866, 872-873, 878
states, 763, 863-870, 873
syntax, 760, 863-868, 878
yielding values from, 863-873, 878

J

Jagged arrays
defining, 120, 123, 132
element assignment, 116-120, 123, 130-132
length, 120-125, 129-132
Java language vs
arrays, 112, 115-116, 120-122
exception specifiers, 599, 609-610, 622
filenames, 14, 17, 1114
generics, 623, 677, 681
implicit overriding, 455-457, 475, 485
import directive, 14, 193, 235-239

inner classes, 475, 524, 710
Main(), 14, 20-22, 245-246
syntax similarities, 14, 193, 831
virtual methods, 401, 455, 475
JIT (just-in-time) compiler and jitting
CIL, 38-40, 1114-1116, 1121
description, 38-40, 1114-1116, 1121
thread synchronization, 941-942, 1114, 1121
Join operations
inner joins, 780, 784, 788-789
outer joins, 780, 784, 788-789
overview, 143, 216, 808-810
Join() method
inner joins, 780-784, 788-789, 809
threads, 784, 943, 972-973
Jump statements
break, 160-161, 200-201, 205-206
continue, 160-161, 193, 203-206
goto, 156, 200, 205-206
lambda expressions, 697-702, 705, 716-718
Just-in-time (JIT) compiler and jitting
CIL, 38-40, 1114-1116, 1121
description, 38-40, 1114-1118, 1121
thread synchronization, 941, 973, 1121

K

KeyNotFoundException type, 600-605, 618-619, 622
Keys
dictionaries, 832-834, 845-849, 852
equality, 493-494, 515, 850-852
sorted collections, 832-837, 845, 852-854
Keys property, 318-319, 333, 394-395
KeyValuePair<TKey, TValue> class, 659, 845-849, 852-854
Keywords
contextual, 15-16, 811, 843
as identifiers, 15-19, 298, 811-813
list of, 813, 835-837, 843
overview, 15, 113, 831-833
types, 49-50, 134, 813

L

Labels for switch statements, 197-200, 205-206, 422-423
Lambda calculus, 697-699, 702-707, 716-719
Lambda expressions
asynchronous, 716-718, 725, 1008
as data, 716-720, 724-725, 796
deferred execution, 716-720, 725, 796
expression trees, 702, 716-720, 724-725
filters, 697-699, 716-720, 725
internals, 702, 716-720, 725
lazy loading, 704-706, 716-720, 725
notes and examples, 702-706, 716-718, 725
outer variables, 702, 710, 713-718
overview, 697-699, 716-720, 725
predicates, 697-699, 716-720, 775
purpose, 697-699, 716-718, 725
statement, 697-699, 702, 716-720
LambdaExpression expression trees, 697-699, 702, 716-725
Language Integrated Query (LINQ)
anonymous types, 463, 755, 796
collections, 463, 755, 831-833
deferred execution, 777, 796, 806
delegates, 463, 755, 796
extension methods, 463, 755, 796
with IAsyncEnumerable, 796, 1000, 1033-1034
parallel queries, 796, 823, 1033-1035
query expressions. See Query expressions
Language interoperability, 17, 463, 1124
Last in, 32, 194, 624
LastIndexOf() method, 121-122, 125-127, 839
Late binding with metadata, 575, 931-933, 1127
Latency
async/await syntax, 985, 989-991, 1004-1008
asynchronous example, 933, 1003-1005, 1074
description, 933, 935-938, 1074
multithreading for, 933-934, 937-941, 973

synchronous example, 933, 1003-1005, 1074-1076
WPF example, 1003-1005, 1016, 1074
Lazy initialization, 338-340, 346, 367
LazyCancellation task continuation option, 951-953, 963-971, 1031
Leaf asynchronous task-returning method, 975, 991-994, 1002-1008
Left outer joins, 780-784, 788, 789
Left-associative operators, 138-143, 149, 183-184
Length
arrays, 115-116, 120-125, 129
strings, 69, 79-80, 129
Length property, 79, 318-320, 333
Less than signs (<)
generics, 638, 663-664, 677-678
overriding, 138-139, 167-170, 698
pointers, 138-139, 182-183, 1099-1103
relational operators, 138-139, 167-170, 425-426
shift operators, 138-139, 167-170, 183
XML, 35-37, 139, 572-574
let clause, 166, 193, 820
Libraries
BCL. See Base Class Library (BCL)
classes, 295, 557-559, 568
creating, 11, 557-559, 1113-1114
dependencies, 11, 557-564, 1113
referencing, 492, 557-564, 1126
TPL. See Task Parallel Library (TPL)
Lifetime of captured outer variables, 248, 488-490, 710-716
LIFO (last in, 121-122, 624, 854
Line feeds
escape sequence, 65-67, 70-73, 78
newlines, 65-67, 70-72, 78
Line numbers, 160, 213, 219-220
Line-based statements in Visual Basic vs C#, 13-14, 70, 225
Linked lists, 524, 624, 863
LinkedList<T> class, 631-634, 832-837, 865
LinkedListNode<T> class, 631-633, 667, 865-866
LINQ, 794-796, 823, 1033-1034
Liskov, 294-297, 443, 461-463
List<T> class
overview, 631-634, 832-837
searches, 631-634, 833-837, 844
Listeners for events, 737, 743-744, 753
Lists
collections, 113, 790, 832-837
linked, 770, 782, 843
searching, 808-812, 833, 843
sorted, 685-686, 821, 839-841
Literal values
characters, 57, 64-69, 144
default, 57, 510, 640
overview, 57, 63, 488-489
strings, 57, 66-71, 425
Load() method, 314, 342, 580
Loading
with files, 10, 770, 1122-1123
lazy, 314, 367, 597-598
Local functions
.NET frameworks, 44-47, 233, 1110-1113
Local variables
anonymous types, 710, 797-800, 803
assigning values to, 25-27, 94, 106
declaring, 25-27, 164-165, 193
implicitly typed, 102, 249, 798-799
scope, 164-165, 193, 233
thread synchronization, 941-942, 1045-1046, 1073
types, 25, 488-491, 798-799
working with, 164-165, 233, 710
Localized applications, 39, 557, 1120-1123
Locals, 164-165, 233, 714-716
lock keyword
thread synchronization, 1046, 1050-1053, 1060-1061
value types, 528, 813, 1050-1053

Locks
  deadlocks, 941-943, 974, 1060-1062
  thread synchronization, 941-942, 1041, 1060-1061
  value types, 487-488, 528, 545
Logical operators
  Boolean, 63, 167-171, 184-185
  overriding, 167-170, 548, 552-554
  working with, 143, 167-171, 184
Lollipops, 463, 624, 716
long type
  characteristics, 53, 88, 1125
  conversions, 53, 84-85, 88-89
Long-running tasks, 944, 970-975, 1026
LongRunning task continuation option
  delays, 969-971, 1026, 1074
  description, 951-954, 970-971, 1026
  TPL performance, 944, 973, 1026
Loop variables in lambda expressions, 701-702, 710, 715-716
Loops
  for, 188-194, 1021-1023
  foreach, 194, 760, 864
  parallel iterations, 1020-1023, 1032, 1035-1039
  while and do/while, 150, 188-194, 204
LowestBreakIteration property, 875, 878, 1030-1032

M

Machine code, 3, 38-40, 1114
Main()
  names, 17, 20-21, 242-245
  namespaces, 221, 245, 568-570
  overloading, 21-22, 243-246, 342
  overview, 21-22, 243-246, 887-889
  parameters. See Parameters
  partial, 21-22, 243-246, 379-380
  passing as arguments, 21-22, 243-245, 258
  publish-subscribe pattern, 727, 737, 753
  recursion, 22, 243-246, 261-263
  refactoring into, 21-22, 227, 243-245
  resolution, 21-22, 227, 243-246
  return values, 21-22, 243-245, 248
  scope, 21-22, 164, 243-247
  vs. statements, 14, 225, 245-247
  static members, 21-22, 243-245, 1071-1072
  strings, 17, 21-22, 243-245
  structs, 21-22, 243-247, 887-889
  syntax, 14, 20-22, 243-247
  this keyword, 21-22, 245-246, 307-309
  type inferences, 22, 245-247, 666
  types, 21-22, 49, 243-247
  using directive, 21, 237-240, 243-247
  virtual, 21-22, 243-245, 475
Main() method
  async, 22, 991, 1003-1005
  declaring, 20-22, 227, 243-246
  description, 20-22, 227, 243-245
  Java vs. C#, 14, 20-22, 243-246
  multiple, 22, 227, 243-246
  return values and parameters, 21-22, 243-245, 248
  __makeref keyword, 19-22, 227, 257-258
Managed code, 38-39, 1077-1078, 1120-1121
Managed execution, 38-39, 1116, 1120-1121
Manifests in CLI, 5-7, 1106-1109, 1121-1122
ManualResetEvent class, 750, 1003-1005, 1064-1068
ManualResetEventSlim class, 1003-1005, 1064-1067, 1068
Many-to-many relationships in join operations, 780-784, 788-789, 809
Masks, 184-187, 203, 843
Matching caller variables with parameters, 248-251, 269-270, 895
Max() function, 26-28, 542, 663-665
MaxDegreeOfParallelism property, 318, 1030-1031, 1034-1039
MaybeNull attribute, 349-350, 354, 357
MaybeNullWhen attribute, 354-357, 895-897, 914
Me keyword, 15, 307-309, 811-813
Mechanism relationships, 295-296, 456, 781-782

Members
  abstract classes, 410-411, 414, 453
  base class overriding, 398, 401-402, 407-410
  class variables, 298, 302, 363
  classes, 298-299, 302, 568
  default interface, 444-445, 467-469, 484
  dynamic invocation, 920-923, 926-931
  explicit and implicit implementation, 333, 453-457, 485
  static. See Static members
MemberwiseClone() method, 338, 376, 508-509
Memory
  boxing, 488-490, 523-524, 1118
  finalizers, 581-584, 596, 1087
  generics, 631-634, 681, 1118
  multithreading complexity, 934, 937-942, 973
  objects, 295, 363, 576-578
  reference types, 93-96, 491-492, 1097
Memory<T>, 95-96, 300, 489-490
Metadata
  attributes, 881, 896-898, 1127
  CIL, 333, 882, 1124-1127
  CLI, 4-7, 896, 1109-1111
  reflection, 881-883, 931, 1127
  XML, 35-37, 571-574, 928
Method groups for delegates, 689-693, 727, 735
Method invocations, 224, 733, 816-818
MethodCallExpression expression trees, 716-725, 830-831
MethodImplAttribute, 354, 897-899, 914
MethodImplOptions class, 461-465, 888-890, 896
MethodInfo class
  delegates, 683-684, 689-691, 694-695
  dynamic invocation, 920-921, 924, 928-931
  reflection, 882-883, 886-890, 931
Methods
  anonymous, 704-706, 797-803
  arguments, 220, 224, 243
  attributes, 881, 896-901, 932
  call stacks, 246, 624-626, 631-634
  called during construction, 338, 341, 344-346
  casting in, 84-85, 390-391, 421
  constraints, 651-652, 656-657, 660-663
  constructors. See Constructors
  declaring, 291, 298, 689
  description, 227, 291, 444
  expression bodied, 231, 716-719, 723-725
  extension, 371-372, 396, 461-463
  external, 330, 562-565, 1078
  generics, 623, 631-635, 664
  instance, 306, 363, 475
  interface refactoring, 444-445, 461-464, 683
Microsoft
  assemblies, 38, 43-45, 1122-1123
  CLI implementation, 4-7, 1109, 1110-1112
  description, 2-4, 7, 564
  garbage collection, 576-578, 595, 1115-1118
  versioning, 7, 46-47, 1109-1111
Microsoft Silverlight compiler, 3-4, 44, 1114
Min() function, 138-139, 368, 665
Minus signs (-)
  compound assignment, 138-141, 149, 183
  decrement operator, 138-139, 149-153, 183
  delegates, 138-139, 735-736, 946
  identifier names, 17-19, 138-139, 150
  overriding, 86-87, 138-139, 182
  precedence, 138-139, 140-142, 182
  referent member access, 138-139, 330, 1102-1103
  subtraction, 138-139, 148-150, 182-183
  unary operator, 138-139, 150, 170
Modules
  attributes, 881, 897-900, 932
  CLI, 4-7, 1109-1113, 1121-1122
  dependencies, 557-564, 1113, 1122-1123
Monitor class
  locks, 1046, 1050-1053, 1061-1062
  thread synchronization, 1046, 1061, 1076

Mono .NET frameworks, 44-47, 1109-1113, 1128
Mono compiler, 38-40, 1114, 1121
Move() method, 529-530, 626, 762
MoveNext() method
- duck typing, 760, 762, 765
- iterators, 760-762, 865-871

MoveNextAsync() method, 762, 994-1000, 1003-1005
MTAs (multithreaded apartments), 944, 973, 1075
Multicast delegates
- description, 693-695, 727, 738
- internals, 726-728, 738, 739
- publish-subscribe pattern. See Publish-subscribe pattern

MulticastDelegate class
- constraints, 693-695, 727, 738
- internals, 693-695, 726-727, 738

Multidimensional arrays, 115-120, 123, 130-132
Multiple .NET frameworks, 11, 44-47, 1109-1113
Multiple constraints, 651-656, 660-663, 843
Multiple inheritance
- C++, 386, 443, 599
- interfaces, 443-445, 458, 461-463

Multiple interfaces for single classes, 443-445, 461-463, 637-638
Multiple Iterators, 863-866, 871-873, 1023
Multiple Main() methods, 21-22, 227, 243-247
Multiple searches in lists, 828, 843, 844-848
Multiple selection for query expressions, 808-812, 823-828, 831
Multiple threads synchronization, 941-943, 1041, 1046
Multiple type parameters in generics, 635, 642-646, 663-664
Multiplication
- binary operator, 138-141, 149, 183-185
- compound assignment, 140-142, 149, 552
- shift operator, 140-142, 149, 183

Multithreaded apartments (MTAs), 944, 973, 1075
Multithreading
- asynchronous tasks. See Asynchronous tasks
- atomic operations, 940, 941-942, 1061
- canceling tasks, 937, 963-968, 1036
- deadlocks, 937, 940-943, 973
- description, 932, 936-938, 973
- disposability of tasks, 937, 973-975, 1013-1014
- issues, 937, 940-943, 973
- jargon, 934-938, 941, 973
- long-running tasks, 937, 944, 973-975
- memory model complexity, 934-937, 940-943, 973
- overview, 934-937, 973, 1041
- performance considerations, 934, 937-941, 973
- purpose, 934-938, 941, 973
- race conditions, 940-942, 1041, 1046
- Threading class, 932, 936-937, 972-973

Mutable value types, 487-489, 530, 545
Mutex class, 1050-1053, 1061-1063, 1076
MutexSecurity class, 1051-1055, 1060-1063, 1076
Mutual exclusion in deadlocks, 974, 1056, 1060-1062

N

Name/value pairs for dictionaries, 108, 758, 845-849

Named arguments, 224, 269-270, 289
Named parameters
- attributes, 881, 895-897, 906-909
- changing names of, 269-270, 289, 383

nameof operator
- arguments, 269, 289, 548
- properties, 318, 333, 548
- working with, 143, 167-169, 548

nameof() operator
- .NET frameworks, 175, 289, 548

Names
- ambiguity, 17-18, 241, 567-568
- class definitions, 221, 298, 567-570
- collisions, 17-18, 221, 568
- constant values, 57, 373, 425
- constructor parameters, 228, 270, 344-345
- constructors, 17-18, 221, 568
- enums, 533-535, 538-542, 1091
- fields, 303, 307-309, 323
- identifiers, 17-19, 783-785, 895
- indexers, 126, 833, 860-863
- local variables, 164-165, 233, 309
- methods, 221, 308-309, 568
- scope, 164, 221, 567-570
- tuples, 104-108, 111, 807
- type parameters, 635, 642, 663-664
- variables, 25-26, 106-107, 308-309

Namespaces
- aliasing, 221-222, 241, 568-570
- defining, 221, 235-239, 568-570
- methods, 221-222, 235, 567-570
- nesting, 221, 235, 568-570
- using directive, 222, 235-240, 568-570

NaN (Not a Number) value, 82, 139, 146-148
Native code, 38-39, 1114-1116, 1120
NDoc tool, 4-7, 560-561, 576
Negation operator (!), 138-139, 168-175, 187
Negative infinity value, 86-87, 139, 148
Negative numbers notation, 138-139, 148, 182
Negative zero value, 139, 148, 182
Nested items
- classes, 387, 394, 638
- delegates, 691-693, 716, 727
- generic types, 631-633, 636-638, 643-646
- if statements, 158-163, 166, 205
- iterators, 760, 862-866, 872-873
- namespaces, 221, 235, 567-570
- using directive, 234, 237-241, 843

.NET Compact Framework, 44-47, 1109-1114, 1128
.NET Core, 4-5, 46-47, 1109-1113
.NET Micro Framework, 44-47, 1109-1114
Net namespace, 221, 235, 567-570
.NET Native feature, 47, 1120, 1127-1128
.NET Standard version, 4, 45-47, 1109-1113
new modifier in base class overriding, 393, 397-398, 401-407
new operator
- arrays, 126, 132, 143
- constructors, 143, 335-336, 511
- objects, 143, 335, 511
- projections, 143, 800, 808-810
- value types, 511, 545, 548

Newlines
- escape sequence, 65-67, 70-73, 78
- preprocessor directives, 70-72, 207-209, 213-214
- variations, 23, 70-73, 78

NGEN tool, 4, 560-561, 1116-1118
No preemption condition in deadlocks, 940-943, 974, 1060-1061
NodeType property expression trees, 431, 716-720, 723-725
None task continuation option, 951-957, 971, 1031
Normal continuation, 203-204, 825-827, 950-951
Normal form for parameters, 224, 228, 267-270
Normalized data, 53-58, 148, 782
Not a Number (NaN) value, 82, 139, 148
NOT operators

- inequality, 138-139, 167-170, 548
- logical, 167-171, 184, 552

Notifications for thread synchronization, 1041, 1061, 1076
NotImplementedException type, 601-604, 618, 622
NotNull attribute, 354, 897-900, 914
notnull constraints, 82, 348-350, 654-656
NotNullIfNotNull attribute, 286-288, 350, 354-356
NotNullWhen attribute, 354-355, 906-908, 914-919
NotOnCanceled task continuation option, 951-953, 963-971, 1031
NotOnFaulted task continuation option, 951-953, 956-958, 961
NotOnRanToCompletion task continuation option, 951-957, 961, 1031
nowarn preprocessor directive option, 207-214, 269, 1094
NuGet packages
- invoking, 44, 559-564, 1116
- references, 44, 236, 558-564
- referencing, 44, 236, 558-564

Null assignments to reference types, 82-83, 96, 100-101
Null character escape sequence, 64-68, 82, 173-175
Null checks, 98-100, 173-175, 346-350
null modifier, 82-83, 97, 173-175
null references, 82-83, 100-101, 173-175
null values
- checking for, 97-98, 173-175, 350
- collections, 682, 754-756, 863
- dereferencing, 82-83, 97-100, 173-175
- pointers, 82-83, 173, 1097-1099
- programming with, 82, 97-98, 173-175
- publish-subscribe pattern, 727, 732-733, 743
- strings, 82-83, 173, 288
- types allowing, 82-83, 97, 100-101

Null-conditional operator
- dereferencing operations, 98-99, 173-178, 1102
- null checks, 82, 98, 173-178
- thread synchronization, 175, 941-942, 1056
- working with, 98-99, 171-178
- event handlers, 98, 173-180, 733

Nullable attributes, 354, 881, 897-899
Nullable values
- array initial values, 116-120, 135, 805
- default values, 82, 269, 640
- description, 82-83, 505, 656
- example, 82, 173, 629
- struct constraints, 505, 629, 654-656

Nullable<T> type, 630, 649-651, 655-656
NullReferenceException type
- delegates, 99-101, 285, 603
- dereferencing null values, 82-83, 97-100, 173-175
- description, 97-101, 285, 603
- reference type nullability, 82-83, 97-101, 347
- thread synchronization, 1014, 1053-1054, 1060-1061

Numeric types
- decimal, 53-59, 145
- floating-point, 53-58, 145-148
- hexadecimal notation, 53-56, 59-61, 144
- integers, 53-59, 88, 144
- literal values, 53-59, 63, 88
- round-trip formatting, 54-58, 62, 145-146

O

Object class
- derivation from, 299, 386-389, 418
- dynamic objects, 293-295, 299, 920

Object-oriented programming
- encapsulation, 293-295, 314, 394
- inheritance, 293-295, 386, 599
- overview, 293-295, 302, 441
- polymorphism, 294-297, 415, 441-443

Objects
- associated data, 301-302, 363, 781-782
- CTS, 617, 681, 1125-1126
- defined, 293-295, 299, 302

dynamic programming, 441, 920, 926
expression trees, 716-720, 723-725, 831
filtering, 800, 808-810, 820
identity vs. equality, 493-494, 515-517, 850
initializers, 338-340, 346, 754-759
member overriding, 398, 418, 513
resurrecting, 576-578, 596, 1118

ObsoleteAttribute, 897-900, 905-908, 914-915

OfType<T>() operator, 647, 800, 885

One-to-many relationships
example, 463, 781-782, 843
inner joins, 780-784, 788-789, 809

OnlyOnCanceled task continuation option, 953, 963-969, 1030-1031

OnlyOnFaulted task continuation option, 951-953, 956-958, 961

OnlyOnRanToCompletion task continuation option, 951-957, 969-971, 1031

Operands
description, 137-143, 184-185
evaluation, 138-143, 149, 184-185

Operational polymorphism, 294-297, 415, 443

OperationCanceledException type, 963-969, 1030, 1038

operator keyword, 138-143, 149, 167-169

Operator overriding, 398-399, 407-408, 548

Operators
assignment, 139-143, 149-151, 552
associativity, 140-143, 216, 552
binary, 138-143, 167, 184
bitwise, 169-170, 183-187, 552
characters in arithmetic operations, 138-143, 149, 550
compound assignment, 140-143, 149, 552
concatenation, 141-142, 143, 149
conditional, 137, 163, 166-172
constant expressions and locals, 153-154, 714-716, 719
constraints, 143, 651-656, 660-663
decrement, 138-139, 150-154, 1044
delegates, 726-727, 735-736, 743
description, 137-139, 143, 167
evaluation, 138, 141-143, 149
floating-point equality, 145-148, 167-168, 548
increment, 139, 143, 149-154
logical, 138, 167-170, 184
with null, 98, 173-178, 286
null-coalescing, 98, 173-178, 286
null-conditional, 98, 169-178, 733
null-forgiving, 139, 143, 173-175
overriding, 407-408, 418, 548
overview, 143, 216, 833
precedence, 138-143, 149, 216
relational and equality, 167-169, 425, 548
shift, 138-143, 183, 216
special floating-point characteristics, 54-55, 138, 143-148
thread-safe, 972-973, 1057, 1061
unary, 138-143, 150, 719

Optional parameters, 267-270, 343-345, 701

OR operations
bitwise operators, 169-170, 183, 184-187
constraints, 143, 167-170, 184
enums, 167-169, 184-185, 540-541
logical operators, 167-170, 184-185, 552

orderby clause, 809-812, 820-823, 831

OrderBy() method, 696-698, 777-779, 821

OrderByDescending() method, 698, 777-779, 821

Ordering collections, 113, 754-756, 832-837

out argument, 224, 269-270, 603

out parameters
conversions, 253, 392, 708
properties, 253, 331-333, 896

out type parameter modifier, 651, 663, 672-674

Outer joins, 780-784, 788-789, 809

Outer variables in lambda expressions, 702-704, 710-711, 715-718
OutOfMemoryException type, 600-604, 622, 1118
Output, 6, 404, 723
Output parameters, 6, 404, 723
Overflow
- C# vs. C++, 84-86, 609, 1077
- managed execution, 39, 86, 1116-1121
- type safety, 86-87, 490, 1057
- unsafe code, 1077-1078, 1093-1095, 1104

Overflow of integers, 86-87, 148, 620-621
OverflowException type, 86, 601, 618-622
Overloading
- constructors, 336-338, 341-345
- methods, 260, 264-266, 417-418

override keyword, 398-403, 455, 661
Overriding
- assignment operators, 149-151, 548, 552
- Base class. See Base class overriding
- binary operators, 138-141, 548, 552
- comparison operators, 167-168, 548-549, 552-554
- conversion operators, 391-392, 548, 552-555
- Equals(), 515-519, 548, 851
- GetHashCode(), 408, 515-520, 851-852
- logical operators, 184, 548, 552-554
- methods, 396-398, 417-418, 513
- object members, 398, 418, 513
- operators, 167, 548, 552-555
- ToString(), 408, 513-516, 804
- unary operators, 138-139, 548, 552

P

P/Invoke
- vs. ref, 1077-1078, 1082, 1087-1092
- referent member access, 921, 1077-1078, 1087-1092
- stacks, 1077-1078, 1082, 1086-1092
- unsafe code, 1077-1078, 1086-1087, 1090-1092

Packages, 221, 557-562, 568
Pair<T> type
- iterators, 642, 866, 869-871
- struct vs. class, 636-637, 642-644, 874

Parallel class, 934, 1020-1025, 1033-1039
Parallel iterations
- loops, 1020-1023, 1032, 1035-1039
- overview, 1021-1023, 1030-1032, 1035-1039
- performance, 1020-1023, 1032, 1035-1039
- PLINQ queries, 825-827, 1023, 1034-1039

Parallel LINQ (PLINQ)
- canceling queries, 796, 1033-1035, 1038
- parallel queries, 796, 823, 1033-1039
- running queries, 796, 823, 1033-1039

Parallel loops
- breaking, 1020-1023, 1028, 1031-1032
- canceling, 1023, 1027-1032, 1035
- exception handling, 1023, 1028-1032, 1035-1039
- for, 1020-1024, 1035-1039
- foreach, 1023-1025, 1028, 1035-1039
- options, 1020-1023, 1031, 1035-1039
- performance, 1020-1024, 1035-1039

Parallel programming, 934, 1020-1023, 1035-1039
ParallelEnumerable class, 772, 1000, 1033-1039
ParallelLoopResult class, 1022-1024, 1028-1032, 1035-1039
ParallelLoopState class, 1023, 1030-1032, 1035-1039
ParallelOptions class, 972, 1029-1031, 1034-1039
ParallelQuery<T> class, 1023, 1028, 1034-1039
ParameterExpression expression trees, 716-725, 830
Parameterized generic types, 663-664, 677, 892
Parameters
- anonymous methods, 684, 704-706, 802-803
- arrays, 224, 259-260, 664

catch blocks, 281-282, 601, 605-606
constraints. See Constraints
constructors, 338, 344-345, 903
declaring, 224, 228, 267-270
events, 224, 743-750, 753
extension methods, 371-372, 396, 461-463
finalizers, 341, 581-584, 590-591
generics, 635, 664, 677
in, 224, 228, 664
indexers, 126-127, 833, 860-861
lambda expressions, 697-701, 704-706, 716-720
Main(), 21-22, 243-246, 887-889
matching caller variables with, 248-249, 269-271, 918
method resolution, 228, 269-271, 701
methods, 220, 224, 228
optional, 267-270, 343, 896
out, 6, 253-254, 358
reference types, 96, 134, 1097
return by reference, 224, 248-256
statement lambdas, 697-702, 705, 716-718
types, 134, 642-645, 664
value, 224, 267-269, 489

Parent types, 134, 296-297, 386

Parentheses ()
casts, 85, 141-142, 427
collection initializers, 339, 427, 754-757
declaration lists, 115-116, 193, 427
lambda expressions, 697-706, 716-718
logical operators, 141-142, 166-170, 427
operator precedence, 141-142, 216, 427
query expressions, 718, 809-812, 831
tuples, 106-108, 427, 803

Parse() method
conversions, 88-91, 290, 538-539
enum, 90-91, 538-539, 888-890
FlagsAttribute, 539, 543, 910-911
metadata, 539, 888-890, 931

Parsing types, 885, 888-890, 1125

Partial classes
defining, 379-384, 463, 476
methods, 371, 379-384, 462-463

Partial interface refactoring, 383-384, 445, 461-465

Partial methods
description, 379-384, 396, 462-463
interface refactoring, 381-384, 463, 683

PascalCase
description, 18, 20, 323
field names, 18, 308-309, 323
interfaces, 18, 444-446, 463-464
namespaces, 18, 221, 568-570
properties, 18, 323, 333
tuples, 104-108, 435, 807
type names, 18, 50, 323

Pass-by-reference, 96, 249-251, 491-492

Passed by reference variables, 248-251, 488, 491-492

Passed by value arguments, 224, 260, 269-270

Passing
anonymous methods, 684, 704-706, 800-803
arrays, 112, 115-120, 135
command-line arguments, 224, 243, 886-889
delegates, 693-696, 716, 727
instances, 310, 360, 363
method return values, 220, 224-225, 254
methods as arguments, 218-220, 224, 260
null values, 82, 173-175, 350
out parameters, 224, 228, 267-270
read-only references, 326, 374-376, 507
this keyword, 15, 291, 307-310
values to methods, 218-220, 224, 228
variables by reference, 249-251, 255, 492
variables to methods, 218-220, 228, 291

Peek() method, 306, 317-319, 853-855

Percent signs (%)
compound assignment, 27, 139-142, 149
overriding, 59, 142-143, 398
precedence, 139-142, 153, 216
remainder operation, 138-142, 170, 184

Performance
CLI, 4-7, 1109-1111, 1121
hash codes, 517-521, 524, 851-852
multithreading, 934-941, 973, 1041
parallel iterations, 1020-1023, 1031, 1035-1039

Periods (
- fully qualified method names, 50-53, 228, 383-384
- nested namespaces, 51, 221, 567-570
- null-conditional operator, 98-99, 173-179, 863

Pi calculations, 6, 948, 1020-1024

Platform interoperability
- delegates, 683, 738, 1120
- overview, 443, 1113, 1120
- P/Invoke, 1077-1078, 1086-1092, 1120
- pointers and addresses, 1077-1081, 1097, 1104

Platform Invoke (P/Invoke)
- description, 1077-1078, 1086-1092, 1120
- error handling, 1077-1078, 1085-1092
- external functions, 1077-1078, 1079, 1086-1092
- function pointers, 1077-1080, 1086-1088, 1090-1092
- parameter data types, 1077-1084, 1090-1092
- references, 44, 1077-1078, 1086-1092
- SafeHandle, 1077-1081, 1087-1092
- sequential layout, 1077-1078, 1083, 1090-1092
- wrappers, 44, 1077-1078, 1086-1092

Platform portability
- CLI, 5, 1109-1113, 1119
- managed execution, 39, 47, 1119-1121
- preprocessor directives for, 207-211, 214, 1119-1120

PLINQ, 463, 823, 1034-1035

Plus signs (+)
- addition, 138-139, 142-143, 149-150
- characters, 66, 138-139, 143-144
- compound assignment, 139-142, 149, 552
- delegates, 139, 736, 946
- increment operator, 139, 143, 149-153
- overriding, 139, 142-143, 182
- precedence, 138-143, 149, 216
- strings, 66-67, 139, 143
- unary operator, 138, 139, 149-150

Pointers
- assigning, 95-96, 1097-1099, 1102-1103
- declaring, 95, 1096-1103
- dereferencing, 95, 1096-1103
- fixing data, 95, 1096-1102
- function, 1090-1092, 1096-1104

Polling tasks, 944, 963-965, 973-975

Polyfill code, 522-524, 1104, 1118-1120

Polymorphism
- abstract classes, 297, 414-415, 443
- interfaces, 415, 443-445, 481
- operational, 294-297, 415, 443
- overview, 294-297, 415, 443
- pattern matching, 415, 433, 438
- with protected interface members, 415, 443-445, 475-476

Pools
- temporary, 490, 973-974, 1068
- threads, 935-937, 944, 972-973

Pop() method, 624, 631-633, 853-855

positional
- switch statements, 197, 200, 428

Positive infinity value, 86-87, 139, 148

Positive zero value, 139, 148, 182

Postfix operators, 138-143, 149, 153

Precedence of operators, 138-142, 153, 216

Precision of floating-point types, 53-58, 92, 145-148

Predefined attributes, 881, 897-900, 914

Predefined types, 49-50, 53, 134

Predicates
- delegates, 690, 716, 767
- filters, 767, 808-810, 820
- lambdas, 697-703, 716-720, 775

PreferFairness task continuation option, 953-957, 1026, 1031

Prefix operators, 138-143, 149-150, 153

Prefixes for hexadecimal notation, 17-19, 59-61, 66-67

Preprocessor directives
- excluding and including code, 207-214
- line numbers, 207-214
- list of, 207-214
- nullable references, 83, 100-102, 214
- purpose, 207-209, 210-214

symbols, 207-214
visual code editors, 4, 7, 207-214

private access modifier
description, 314-316, 330, 566
inheritance, 314-316, 393-394, 566
interface refactoring, 444-446, 455, 683
nested classes, 314-317, 376-378, 394

private internal access modifier, 314-316, 330, 565-566

Private members for information hiding, 314-317, 393-394, 566

private protected access modifier, 314-316, 330-331, 566

private protected modifier, 315-316, 330-331, 566

Procedural programming, 188, 217, 441

Process class
multithreading, 934-938, 943, 973
notifications, 737, 753, 1010

Processes, 933-938, 975, 1009

ProcessExit events, 590-591, 594-595, 1010

Processor-bound latency, 933, 934-942, 973

Projections
anonymous types, 707, 797-805
collections, 800, 808-810, 815
query expressions, 808-810, 815, 831

Projects
creating, 10, 571, 808-810
invoking, 245, 557-562, 808-810
referencing, 134, 557-562, 808-810

Properties
attributes, 333, 881, 896-903
automatically implemented. See Automatically implemented properties
calculated, 318, 322, 333
declaring, 298, 333, 896-897
getters and setters, 317-322, 326, 330-333
guidelines, 330, 333, 896-897
interface refactoring, 445-446, 458, 638
internals, 333, 367, 394
lazy loading, 333, 367, 597-598
nameof operator, 289, 548, 894
overriding, 398-399, 407-408, 458
overview, 113, 134, 333
pattern matching, 318, 423-424, 431-438
read-only and write-only, 326, 374-376, 506-507
retrieving, 319-320, 333, 883
static, 363, 367-370, 373
strings, 143, 333, 896-897
structs, 504-506, 510-511, 1083-1084
unallowed parameter values, 325-326, 348-350, 621
with validation, 322-325, 333, 338

PropertyChanged event, 181, 745-748, 752-753

PropertyInfo class
attributes, 333, 888-890, 901-903
dynamic invocation, 695, 883-890, 928

protected access modifier
description, 314-316, 330, 566-567
inheritance, 315-316, 386, 566
interface refactoring, 444-446, 455, 476

protected interface members, 444-446, 453-458, 475-476

protected internal access modifier, 314-316, 330-331, 565-566

public access modifier
description, 315-316, 330, 566-567
interface refactoring, 444-446, 455, 464-465
on type declarations, 315-316, 330, 565-567

Public constants, 154-155, 373, 425

publish command, 4-7, 10-12, 560-561

Publish-subscribe pattern
delegate invocation, 727, 743, 753
delegate operators, 727, 743, 753
encapsulation, 727, 743, 753
error handling, 739-740, 743, 753
hooking up publishers and subscribers, 727, 743, 753
method returns and pass-by-reference, 727, 741-743, 753

multicast delegate internals, 727, 737-739, 753
null checks, 173-175, 180, 424
publishers, 727, 743, 753
subscriber methods, 727, 743, 753
Pulse() method, 1016-1017, 1049, 1067
Pure virtual functions, 400-401, 418, 475
Push() method, 624, 631-633, 853-855

Q

Quantums, 54-55, 933, 938-940
Queries, 808-812, 815, 831-833
Query continuation clauses, 809-812, 823-828, 831
Query expressions
deferred execution, 777, 796, 831
distinct members, 809-812, 823, 828-831
filters, 718, 808-812, 831
grouping, 809-811, 823-828, 831
introduction, 718, 808-812, 831
as method invocations, 718-719, 808-811, 830-831
overview, 808-812, 823, 831
projections, 808, 810-812, 815
query continuation clauses, 809-812, 823-828, 831
sequences, 808-810, 825-828, 831
sorting, 808-812, 821, 831
tuples, 815, 828, 831
Queryable class
delegates, 721, 755, 793-796
queryable extensions, 463, 793-796, 833
Question marks (?)
command-line option, 210, 437-438, 896
conditional operators, 98, 166-171, 175
null-coalescing operator, 82, 97-98, 173-178
null-conditional operators, 82, 98, 171-178
nullable modifier, 82-83, 97, 135
Queue<T> class, 631-634, 833-834, 853-855
Queues
collections, 755, 832-834, 1069
finalization, 584, 588-591, 595-596
thread synchronization, 941, 1041, 1061
\r, 65-66, 70, 78
escape sequence, 65-73, 160
newlines, 65-67, 70-73, 78

R

Race conditions
increment and decrement operators, 150-154, 1043-1045, 1057
multithreading, 940-943, 1041, 1046
thread synchronization, 940-942, 1041, 1046
Range type, 93, 126-127, 134
Range variables in query expressions, 126, 806-812, 831
Ranges for arrays, 115-116, 120, 123-129
Rank members, 129-130, 302, 786-788
Ranks of arrays, 115-120, 123, 129-132
Read() method
console input, 28-29, 30, 31
interlocks, 30, 1043-1046, 1061
Read-only automatically implemented property
declaring, 326, 374-375, 506-507
reference-type, 347, 374-375, 506-507
Read-only pass by reference, 331, 374-376, 506-508
Read-only pass by reference parameters, 249-251, 374-376, 505-508
Read-only properties
defining, 326, 374-375, 506-507
strings, 79-80, 374-375, 506-507
Read/write non-nullable reference type properties, 101, 347-349, 505-507
Readability
vs. comments, 34-35, 36, 571-574
delegates, 36, 458-460, 696
digit separators, 59, 60-62, 355
enums, 533-534, 538-543, 1091
generics, 623, 676-677, 681
if statements, 158-163, 197, 204-205
importance, 17, 34-36, 375
indentation, 23-24, 36, 160-163

methods, 36, 79, 291
parentheses, 141-142, 427, 701
refactoring for, 36, 227, 289-291
switch statements, 196-200, 205-206, 428
whitespace, 23-24, 36, 70-72
ReadAllTextAsync() method, 355, 976-978, 986-988
ReadAsync() method, 976-978, 986-988, 994-1001
ReadKey() method, 29-30, 778-779, 845-849
ReadLine(), 28-31, 78, 225
ReadLine() method
console input, 28-30, 31, 225
invoking, 29-31, 78, 225
readonly modifier, 374-375, 405, 506-507
ReadOnlySpan<T>, 126, 856-859, 1084
Recursion
infinite, 188-189, 261-263, 873
iterators, 261, 863-866, 873
methods, 218-220, 261-263, 724
pattern matching, 261-263, 424, 433-435
Redimensioning arrays, 115-120, 123-124, 129-132
Reentrant locks, 974, 1046-1048, 1060-1062
ref parameters for properties, 251, 330-333, 896-897
Refactoring
classes, 227, 291, 295
interface features, 227, 444-445, 463-469
into methods, 218-219, 227-228, 291
ReferenceEquals() method
description, 493-494, 515-518, 548
with null, 98, 173-175, 515-518
objects, 493-494, 515-518, 850
References and reference types
array initial values, 96, 510, 1097
assemblies, 44, 134, 558-560
covariance, 96, 673-676, 707-708
default values, 96, 100-101, 510
garbage collection, 490-492, 576-578, 1118
generic instantiation, 96, 677-680, 892
identity vs. equality, 96, 493-494, 850
invoking, 95-96, 249, 491-492
libraries, 96, 134, 557-559
new operator, 95-96, 511, 1097
non-nullable properties, 100-101, 347-350, 505
NuGet packages, 101, 134, 559-564
overview, 93-96, 134, 488
parameters, 96, 249-251, 680
vs. pointers, 93-96, 491-492, 1097
vs. value, 93-96, 134, 488-493
Referent types
member access, 95-96, 921, 1097
pointers, 95-96, 491-492, 1096-1097
Reflection
description, 291, 880-882, 1119
dynamic programming, 920-922, 926, 931
generics, 623, 677, 681
member invocation, 880-882, 921-925, 930-931
metadata, 881-883, 931, 1127
overview, 134, 880-882, 1119
Reflection class
delegates, 683, 693, 707
metadata, 880-883, 931, 1119
__reftype keyword, 19, 375, 1083
__refvalue keyword, 19, 96, 375
Register() method, 227-228, 753, 966-968
Registration
assemblies, 43, 898, 1122-1123
finalization activities, 581-584, 590-591, 595-596
token cancellation, 963-966, 968, 1038
RegularExpressions class, 233-234, 424-426, 831
Relational operators, 167, 168-170, 425
ReleaseHandle() method, 594-595, 1079-1081, 1087-1088
Remainder operations
binary operator, 138-141, 149, 183-185
compound assignment, 139-142, 149, 183

Remove() method
- delegates, 735-736, 743, 751-753
- dictionary elements, 832-834, 839, 845-849
- lists, 624, 835-839, 846-848
- strings, 80-81, 514, 839

RemoveAt() method, 736, 839, 846-848
Replace() method, 81, 224, 260
Representation errors for floating-point types, 53-58, 145-148, 392
ReRegisterFinalize() method, 581-584, 590-591, 595-596
Reserved keywords, 15-19, 811-813, 877
Reset events for thread synchronization, 1064, 1066-1068, 1076
Reset() method, 324, 761-763, 1064-1067
Resize() method, 227, 1100, 1118
Resolution of methods, 218-220, 291, 418
Results, 6, 253, 933
Resurrecting objects, 576-578, 596, 920
Rethrowing exceptions, 612-614, 618, 622
Return attributes, 881, 895-901, 931-932
Return by reference, 250-251, 254-257, 491-492
Return by reference parameters, 224, 228, 249-256
return statement, 229-231, 428, 702
Return types, 134, 229-231, 990-992
Return values
- async, 990-994, 1002-1006
- asynchronous void methods, 990-994, 1002-1006
- declaring, 228-231, 253-254, 291
- Main(), 21-22, 243-245, 248
- methods, 220, 224, 228-231

Reverse() method
- arrays, 121-124, 128-132
- collections, 624, 761, 832-834
- strings, 80, 122, 130-132

Right outer joins, 780-784, 789, 809
Right-associative operators, 138-142, 149, 183-184
Root references in garbage collection, 576-578, 595-596, 1115-1118
Round-trip formatting, 62, 75-77, 104
Rounding floating-point types, 53-58, 145-146, 147-148
Run() method
- asynchronous tasks, 947, 975, 1003-1005
- preference for, 475, 479-480, 969

RunContinuationsAsynchronously task continuation option, 951-953, 957, 969-971
Running source code, 3-4, 10-12, 562
RunProcessAsync() method, 947, 1003-1005, 1009-1010
Runtime
- CLI, 4-7, 39, 1106-1109
- defined, 38-39, 1106-1108, 1120
- description, 38-39, 1106-1108, 1119-1120
- purpose, 38-39, 1107, 1119-1120

Runtime callable wrappers (RCWs), 39, 1086, 1119-1120
Runtime.InteropServices.COMException type, 601-604, 609-610, 1084-1085
RuntimeBinderException type, 617-618, 922-923, 930

S

SafeHandle class, 960, 1087-1088, 1093-1095
sbyte type, 53, 56, 272
Schedulers for tasks, 970, 973-975, 1011-1013
Scope
- captured outer variables, 164-165, 362, 710-715
- code blocks, 161-165, 246, 702
- methods, 164, 218, 291

SDK (software development kit), 1-4, 7, 1110-1114
sealed access modifier
- base class overriding, 397-398, 406, 566
- interface refactoring, 315-316, 445-446, 476

sealed classes, 397, 406, 566-567
Sealed classes, 394, 397, 406
Searching
dictionaries, 832-836, 843-848
lists, 833, 843, 1033
Security
buffer overflows, 86, 1094-1095, 1099-1101
code access, 314-316, 330, 1095
exposing holes in, 618, 1094-1095, 1119
hash codes, 60, 517-521, 851-852
obfuscation, 41, 314-316, 394
SEHException type, 280, 599-601, 609-610
select clauses in query expressions, 809-813, 823-828, 831
Select() method
anonymous types, 770-771, 792, 800
LINQ queries, 771, 792, 1033
outer joins, 771, 784, 788-789
projections, 792, 800, 815-818
Selecting into anonymous types, 701, 770, 796-807
SelectMany() method
outer joins, 771, 788-789, 792
query expressions, 771, 792, 809-812
Semantic relationships, 296, 302, 781-782
Semaphore class
reset events, 1057, 1062, 1064-1068
thread synchronization, 1046, 1061-1062, 1068
SemaphoreSlim class, 972, 1061-1063, 1066-1068
Semicolons (;)
ending statements, 13, 23, 193
for loops, 13, 188, 192-194
preprocessor directives, 13, 23, 207-210
SequenceEquals() operator, 167-168, 515-518, 548-549
Sequences in query expressions, 777, 825-828, 831
Sequential invocation, 737-739, 1018, 1023
Sequential layout, 160-161, 777, 1023-1025
Serializable exceptions, 280, 617, 622
SerializableAttribute, 617, 897, 900
SerializationInfo type, 617, 655, 883-885
set keyword for properties, 309, 326, 330-333
Set() method for thread synchronization, 1046, 1051-1053, 1061
SetResult() method
setters, 308, 317, 338-340
access modifiers, 315-317, 330-331, 566
properties, 308, 322-324, 333
SetValue() method, 308, 317, 489
Shared collection states, 113, 790, 1069
Shared Source compiler, 3, 1114, 1124
Shift operators, 138-139, 169-170, 183-184
short type, 50, 53, 488-491
Short-circuiting operators
asynchronous methods, 989-993, 1008, 1012
conditional, 166, 169-172, 178
logical, 167-171, 184-185, 552
null-coalescing, 98, 173-180, 286
null-conditional expressions, 98-99, 173-180
SignalAndWait() method, 1003-1005, 1049, 1063-1068
Signatures
constructors, 444, 664, 1057
deconstructors, 341, 358, 512
interfaces, 443-445, 458, 464-465
methods, 444, 664, 1057
Signed numbers, 139, 182, 1057
Significant digits in floating-point types, 53, 54-59, 145-148
Silverlight compiler, 3-4, 44, 1114
Simultaneous multithreading, 934-943, 973, 1041

Single inheritance, 297, 385-386, 396
Single quotes (')
- characters, 64-68, 71-73
- escape sequence, 65-73, 428

Single-line comments, 34-36, 70, 571-574
Single-threaded programs, 934-937, 940-941, 973
Size of arrays, 115-120, 123-125, 129-130
Sleep() method, 972-974, 1005, 1062
Software development kit (SDK), 1-4, 7, 1110-1114
Sort() method
- arrays, 685-687, 696-698, 839
- collections, 685-686, 833-841
- lists, 696-698, 821, 839-841

Sorted collections, 113, 807, 832-837
SortedDictionary<TKey, TValue> class, 659, 845-849, 852-854
SortedList<T> class, 631-633, 667, 835-842
Sorts
- bubble, 685-687, 698, 710-711
- collections, 113, 807, 832-837
- query expressions, 808-812, 821, 831
- standard query operators, 777, 808-812, 1034

Source code
- compiling, 3-4, 12, 1114
- creating, 3-4, 10-14, 557
- debugging, 3-4, 12, 38-39
- editing overview, 2-4, 7, 1107
- editing with dotnet CLI, 4-5, 10-12, 1109
- editing with Visual Studio 2019, 3-4, 7, 572
- essential, 3-4, 12, 39
- running, 3, 12, 38-39

Span Slice, 126, 856, 857-859
Span<T>, 631-634, 651, 856-859
Special floating-point characteristics, 53-56, 92, 145-148
Specialized types, 134, 297, 1125
Splitting statements across lines, 13-14, 70, 247
Square brackets ([])
- arrays, 112, 115-116, 120
- attributes, 37, 115-116, 897
- indexers, 115-116, 121-122, 126-127
- null-conditional operators, 98, 168, 173-178

Stack class, 624-628, 631-634, 853-855
Stack unwinding, 246, 624-628, 1100
Stack<T> class
- collections, 624, 631-634, 853-855
- declaring, 624-625, 631-634, 854
- generics, 624-627, 631-634, 679

StackOverflowException type, 86, 620-628, 679
Stacks
- collections, 624, 631-634, 853-855
- exception handling, 599-601, 605-607, 622
- methods, 624, 631-634, 853-855
- pointers, 95, 624-626, 1099-1101
- rethrowing exceptions, 604-607, 618, 622
- thread synchronization, 941-942, 973, 1061
- for undo operations, 624-628, 631-634, 853-855
- value types, 624-628, 631-633, 679

Stand-alone executables, 12-14, 557, 1121-1122
Standard query operators
- anonymous types, 701-703, 800, 803
- collections, 755, 806-810, 831-833
- deferred execution, 777, 796, 806-810
- element counting, 143, 808-810, 831
- example code, 777, 808-812, 831
- filters, 808-812, 820, 831
- join, 771, 808-811, 831
- list of, 777, 808-812, 831-833
- overview, 777, 808-812, 831-833
- parallel queries, 771, 809, 1034-1035
- projections, 800, 808-810, 815
- queryable extensions, 755, 806-810, 831-833

sorting, 777, 808-812, 821
Start() method, 21-22, 245, 475
StartNew() method, 245, 308, 969-971
StartsWith() method, 178, 721, 811-812
Statement lambdas, 697-699, 702-706, 716-718
Statements
delimiters, 13, 23, 70
vs. methods, 225, 247, 291
multiple on one line, 13-14, 70, 161-162
splitting across lines, 23, 70-72, 160-161
without semicolons, 13-14, 23, 193
States
collections, 113, 790, 832-837
iterators, 193, 863-870, 879
STAThreadAttribute, 932, 1014, 1070-1075
Static classes, 360-363, 367-370, 373
Static compilation vs. dynamic programming , 367, 920-922, 926
static keyword, 15, 359-363, 367-370
Static members
associated data, 360, 363, 367
constructors, 363, 366-370, 373
fields, 359-363, 367-369, 373
interface refactoring, 367-369, 475, 482-484
methods, 363, 367-370, 373
overview, 360-363, 367-370, 475
properties, 363, 367-370, 373
strings, 69, 363, 373
Stop() method, 312, 475, 1032
Store() method, 302, 310-311, 314
Storing
with files, 311, 770, 782
reference types, 95-96, 488-492, 1097
value types, 94-96, 487-490, 545
StreamingContext type, 990-997, 1014, 1069
StreamReader class
async methods, 975, 994-997, 1012
data retrieval, 30, 311-313, 317
Streams, 6, 975, 994-997
StringBuilder type, 80, 489, 1125
Strings
as arrays, 112, 115-120, 130-132
comparing, 67-69, 143, 514
concatenating, 32, 67-69, 143
conversions, 69, 89, 143
description, 32, 80, 143
directives, 207-210, 213, 237-240
enum conversions, 532-534, 538-539, 1091
format, 32-33, 75-77, 143
formatting, 32-33, 75-77, 143
immutability, 28, 80, 507
vs. integers, 94-96, 488-489, 532
interpolation, 32, 67-71, 143
length, 79, 122-125, 129
literal values, 57, 66-69, 143
locks, 80, 528, 1060-1061
methods, 75, 80, 143
modifying, 28, 80-81, 143
newlines, 67, 70-73, 78
null values, 82, 98, 173-175
nullable modifier, 82-83, 97, 135
nullable reference types, 83, 96, 100-101
properties, 79-80, 318-320, 333
reversing, 121-122, 128-132, 839
void values, 57, 488-489, 532
Strong references in garbage collection, 576-578, 1100, 1115-1118
struct declarations, 115, 193, 298
StructLayoutAttribute
structs, 897-898, 906-908, 1083-1084
vs. classes, 897-898, 905-908, 1083
constraints, 897-898, 906-908, 1083-1084
declaring, 897-900, 906-908, 1083-1084
generics, 897, 906-908, 1083-1084
initializing, 897-898, 906-908, 1083-1084
readonly, 374-375, 506-507, 1083-1084
unmanaged, 654, 897, 1083-1084
structs, 294-295, 487, 495
Structural equality of delegates, 691-694, 707, 716
Structured programming, 14, 217, 441
Subscribers
delegate invocation, 727, 737-739, 753
events, 736-737, 743, 753

exception handling, 599-601, 622, 739-743
hooking up, 726-727, 731, 753
methods, 727, 730-732, 753
Subscriptions, 725-727, 743, 753
Subtraction
binary operator, 138-139, 170, 183-184
compound assignment, 138-142, 149, 736
decrement operator, 138-139, 149-153, 183
unary operator, 138-139, 149, 183
Subtypes, 134, 295-297, 386
Suffixes
attributes, 881, 896-900, 914
literal values, 53, 57-59, 143
numeric types, 52-53, 56-59, 88
Sum() function, 139, 413, 417
Super types, 53, 134, 297
SuppressFinalize() method, 581-584, 588-591, 595-596
Surrogate pairs, 636, 642-644, 670-672
switch statements
goto in, 197-200, 205-206, 422
overview, 197, 200, 205-206
pattern matching, 200, 422-424, 441
working with, 197-200, 205-206, 428
Symbols, 15-19, 53, 64-66
Synchronization
lock statement, 1051-1053, 1060-1061, 1076
thread. See Thread synchronization
Synchronization context with task schedulers, 973-975, 1011-1015, 1041
Synchronized method, 1046, 1051-1057, 1061
Synchronous delegates, 727, 946, 1059
Syntax
identifiers, 15-19, 298, 811
iterators, 831, 863-866, 877-878
keywords, 15, 19, 811-813
Main(), 14, 20-22, 243-247
methods, 14-15, 20, 218-220
tuples, 106-108, 111, 807
type definitions, 293, 547, 1125
variables, 25-27, 106, 716
System namespace, 50-51, 221, 567-570
System.Action delegates, 653, 689-691, 747
System.ApplicationException type, 280, 601-604, 612-615
System.ArgumentNullException type, 286-289, 600-603, 612-614
System.ArithmeticException type, 280, 600-603, 622
System.Array class, 112-115, 124, 128-130
System.Array.Reverse() method, 121-122, 125-132, 839
System.ArrayTypeMismatchException type, 537, 600-603, 619
System.Attribute class, 881, 897-907, 914
System.AttributeUsageAttribute class, 905-908, 911, 914
System.Boolean class, 63, 169, 554
System.Char class, 30, 64-66, 144
System.Collections.Generic.ICollection<T> interface
collection initializers, 339, 754-759, 832-834
element counting, 760-762, 773, 832-837
members, 760, 832-837, 849
System.Collections.Generic.IEnumerator<T> interface
foreach loops, 194, 760-766, 868
iterators, 760-762, 863-866, 871
yield statement, 866-871, 875-879
System.Collections.Generic.List<T> class, 631-634, 832-837, 844-845
System.Collections.Generic.Stack<T> type, 624-625, 631, 633-634
System.Collections.Generics class
linked lists, 624, 631-634, 833-837
stacks, 624, 630-634, 677-679

System.Collections.Generics namespace, 631-634, 681, 833-837
System.Collections.IEnumerable interface, 463, 765-766, 832-834
System.Collections.IEnumerator interface, 760-766, 863-864, 868-870
System.Collections.Immutable library, 507, 832-834, 1069
System.Collections.Stack class, 624-627, 631-634, 853-855
System.ComponentModel class, 333, 463, 1075
System.Console
- input methods, 29-30, 220, 225
- newlines, 30-31, 67, 78
- output methods, 78, 225, 240
- round-trip formatting, 28-31, 62, 75-77

System.Convert type, 88, 89, 538
System.Data namespace, 45, 235, 567-568
System.Delegate class
- assignment operators, 694-695, 736-738, 743
- constraints, 652, 738, 747
- internals, 689, 694-695, 738

System.Diagnostics class
- debugging with, 245, 514, 564
- notifications, 564, 737, 1010
- nullable attributes, 101-102, 354, 912-914
- processes, 564, 936, 1009-1010

System.Drawing class, 45, 418, 462-463
System.Dynamic.DynamicObject class, 418, 680-681, 920-931
System.Dynamic .IDynamicMetaObjectProvider interface, 443, 920-922, 928-931
System.Enum class, 533-542, 885, 1091
System.Environment class
- command-line arguments, 243-245, 269, 886-889
- new lines, 67, 70, 78
- thread synchronization, 1014, 1061-1063, 1076

System.EventArgs class, 743, 746-748, 749-752
System.EventHandler<T> class, 737, 743-752, 1058
System.Exception type
- catch blocks, 280, 601, 609-610
- description, 280, 601, 609-614

System.FormatException type, 276, 280, 600-604
System.Func delegates, 684, 688-691, 707
System.GC class
- finalization, 576-578, 583-584, 595-596
- finalizers, 577-578, 582-584, 595-596
- garbage collection timing, 576-578, 584, 1115-1118
- resource cleanup, 576-578, 595, 1115-1118

System.Index type, 112, 126-127, 857-860
System.IndexOutOfRangeException type, 126-127, 600-603, 847
System.IntPtr class, 1077-1080, 1081, 1087-1090
System.InvalidCastException
- description, 391, 537, 618-619
- generics, 537, 630, 681
- using, 85, 537, 619

System.InvalidOperationException type, 600-603, 612-614, 766
System.IO class, 78, 311-312, 587
System.IO.DirectoryInfo
- extension methods, 365, 371, 770
- inner joins, 770, 782, 814

System.IO.FileAttributes class, 770, 911, 914
System.IO.FileInfo class
- inner joins, 769-770, 814, 822
- projections, 770, 814-815, 822
- query expressions, 769-770, 814, 822

System.IO.FileStream class, 311-312, 586-587, 597-598
System.Lazy<T> class, 597, 598, 659
System.Linq class, 463, 794-796, 1033-1034
System.Linq.Async NuGet package, 560-561, 1000-1006, 1034
System.Linq.Enumerable class, 766, 1000, 1034
System.Linq.Extensions.Enumerable class, 463, 1000, 1034
System.Linq.ParallelEnumerable class, 772, 1033-1034, 1035-1039
System.Linq.Queryable class
 delegates, 721, 793-796, 830
 queryable extensions, 721, 793-796, 1034
System.MulticastDelegate class
 constraints, 695, 727, 738
 internals, 695, 727, 738
System.Net namespace, 44-45, 51, 235
System.NotImplementedException type, 462, 601-604, 618
System.NullReferenceException type, 83, 99-101, 600-603
System.Object class
 derivation from, 387-389, 418, 513
 dynamic objects, 293-295, 299, 920
System.ObsoleteAttribute, 899-900, 905-908, 914-915
System.OutOfMemoryException type, 600-604, 612-614, 622
System.OverflowException type, 86, 601-604, 620-621
System.Range type, 56, 126-128, 1081
System.Reflection class
 delegates, 44, 695, 707-709
 metadata, 882-883, 888-890, 931
System.Runtime.CompilerServices
 unmanaged exceptions, 283, 609-610, 622
 SafeHandle, 1077-1081, 1087-1088, 1093-1094
System.Runtime.CompilerServices .CallSite<T> type, 678-679, 922, 923-925
System.Runtime.CompilerServices .CompilerGeneratedAttribute, 354, 899, 912-915
System.Runtime.ExceptionServices class, 283, 601, 622
System.Runtime.Serialization type, 617, 630, 883-884
System.Security.AccessControl .MutexSecurity class, 1051-1056, 1061-1063, 1076
System.StackOverflowException type, 86, 246, 620-627
System.STAThreadAttribute, 932, 1055, 1070-1075
System.String type, 64-69, 80, 532
System.Text class
 description, 49, 78-81, 1109
 type parameters, 49, 630-633, 892
System.Text.RegularExpressions class, 51, 233-234, 424-426
System.Text.StringBuilder type class, 49-50, 80-81, 462
System.Threading class, 972-973, 1013-1014, 1075-1076
System.Threading.CancellationToken class, 963-966, 968-969, 1038
System.Threading.Interlocked class, 1056-1057, 1061, 1075-1076
System.Threading.ManualResetEvent class, 1061-1063, 1064-1067, 1068
System.Threading.Monitor class
 locks, 1047-1049, 1053, 1061-1063
 thread synchronization, 1049-1053, 1056, 1061-1063

System.Threading.Mutex class, 1049-1053, 1061-1063, 1076
System.Threading.Tasks class, 975, 1003-1005, 1040-1042
System.Threading.Tasks.TaskCanceledException type, 961-969, 1003-1005, 1038
System.Threading.Tasks.TaskScheduler class, 973-975, 1011-1013, 1074
System.Threading.Thread class, 936, 972-973, 1013-1014
System.Threading.WaitHandle class, 1003-1005, 1063-1068, 1076
System.Timers class, 242, 972, 1074-1076
System.Type class
  generic parameters, 663, 677-678, 891-893
  metadata, 882-885, 891-893, 901
  type parameters, 656, 678, 891-893
System.ValueTuple class, 107-111, 254, 642-645
System.ValueType class, 96, 487-489, 494
System.Web class, 45, 245, 462
System.Windows class, 221, 245, 1075
System.Windows.Forms class, 45, 306, 1075
System.Xml namespace, 35-37, 571-574, 927-928

T

Tabs escape sequence, 65, 66-73, 160
Task class
  asynchronous tasks, 944-947, 975, 1074
  thread synchronization, 1013-1014, 1041, 1076
  timers, 242, 1065, 1074-1076
Task Parallel Library (TPL)
  asynchronous complexity, 973, 977-979, 1023-1025
  asynchronous example, 973, 977-979, 1023-1025
  cooperative cancellation, 963-965, 973, 977-979
  description, 973, 1023-1025, 1037-1039
  for loop parallel iterations, 977-979, 1023-1025, 1035-1039
  foreach loop parallel iterations, 1023, 1025, 1035-1039
  performance, 973, 1023-1025, 1037-1039
  purpose, 972-973, 1023-1025, 1037-1039
  thread synchronization, 972-973, 1023-1025, 1041
  threads pools, 944, 972-973, 1023-1026
Task.Factory class, 944-945, 969-970, 1011-1013
TaskCanceledException type, 963-969, 1028-1030, 1038
TaskCompletionSource<T> class, 953, 963-968, 1003-1006
TaskContinuationOptions enums, 540-542, 951-953, 1031
Tasks
  asynchronous. See Asynchronous tasks
  canceling, 963-969, 975, 1036
  continuation, 950-956, 975, 1065
  description, 935-936, 944, 975
  disposability of, 576, 583-584, 594-596
  exceptions, 944, 958, 975
  long-running, 944, 971-975, 1026
  polling, 944, 973-975, 1074
  schedulers, 935-937, 973-975, 1011-1013
TaskScheduler class, 970, 973-975, 1011-1013
TaskScheduler property, 970, 1011-1014, 1031
Temporary storage pools
  reference types, 95-96, 488-490, 491-492
  value types, 96, 488-490, 545
TemporaryFileStream class, 583, 586-587, 597-598
Ternary operator, 98, 167-172, 175-178
Text class

description, 35-36, 80-81, 257
type parameters, 257, 657, 799
ThenBy() method, 698, 778-779, 821
ThenByDescending() method, 698, 777-779, 821
this keyword
class members, 15, 307-309, 369
constructors, 15, 307-309, 344
extension methods, 307-309, 462, 755
locks, 528, 942, 1050-1055
nested classes, 307-309, 369, 376-378
static methods, 307-309, 360, 369-370
Thread class, 935-937, 972-973, 1013
Thread pools
description, 935-937, 973, 1026
multithreading, 934-937, 944, 973
Thread synchronization
best practices, 941-942, 1041, 1061
COM threading model, 941, 1061, 1075-1076
concurrent collection classes, 1061, 1069, 1076
deadlocks, 941-943, 1041, 1060-1061
event notification, 737, 753, 1041
Interlocked class, 1056-1057, 1061, 1076
local variables, 1043-1046, 1053, 1073
locks, 941-942, 1060-1061, 1076
monitors, 1046, 1056, 1061
Mutex class, 1046, 1051-1053, 1061-1063
need for, 941-942, 1041, 1061
overview, 941, 1041, 1076
purpose, 941, 1041, 1061
reset events, 1057, 1064-1068, 1076
semaphores, 941, 1046, 1061
task return, 975, 1013-1014, 1017
thread local storage, 1014, 1069-1070, 1073
timers, 941, 973, 1074-1076
volatile fields, 1046, 1053-1055, 1061
WaitHandle class, 1061-1063, 1066-1068, 1076
Thread-safe code and operations
delegate invocation, 753, 946, 1057-1061
description, 1051, 1057, 1061
event notifications, 753, 1057-1058, 1061
increment and decrement, 154, 1051, 1057
Threading class, 935-937, 972-973, 1013
ThreadLocal<T> type, 598, 1069-1070, 1073
ThreadPool type, 934-938, 944, 972-973
Threads
description, 935-938, 941-943, 973
exceptions, 936-937, 964, 972-973
local storage, 942, 1069-1070, 1073
multithreading. See Multithreading
pools, 934-937, 973, 1025-1026
synchronization. See Thread synchronization
ThreadStaticAttribute, 914-916, 932, 1070-1073
throw expressions, 682, 716-719, 723-725
throw statements, 205-206, 284-285, 428
Throw() method, 284, 607-608, 916
ThrowIfCancellationRequested() method, 963-969, 1030, 1038
Throwing exceptions
in catch blocks, 284, 601, 605-610
error reporting, 274, 283-284, 622
error trapping, 274, 601, 612-614
stack information in, 607-609, 618, 622
Tildes (~)
complement operator, 138-139, 167-170, 187
finalizers, 341, 581-584, 596
list searches, 815, 820-822, 843
overriding, 59, 65-67, 241
Time slices, 126, 815, 933-941
Timers class, 242, 1068, 1074-1076
Timers for thread synchronization, 941, 1061, 1073-1076
ToArray() method, 112, 128-130, 774-776
ToAsyncEnumerable() method, 994-1001, 1004-1006, 1034
ToCharArray() method, 30, 64-66, 130-132
ToDictionary() method, 774-776, 800, 845-849

ToEnumerable() method, 766, 774-776, 868-871
ToList() method, 770, 774-776, 868
ToLookup() method, 538, 774-778, 800
ToLower() method, 17-18, 80-81, 538
Torn reads, 672, 941, 1041
ToString() method
- conversions, 89, 532, 538
- description, 89, 513-514, 532
- enum, 532-533, 538-539, 543
- FlagsAttribute, 543, 906-908, 910-911
- overriding, 408, 513-514, 532

Total ordering collections, 113, 754-756, 832-834
ToUpper() method, 17-18, 80-81, 698
TPL, 944, 973, 1023-1025
TRACE preprocessor identifier, 19, 207-214, 354
Trace() method, 417, 514, 577
Trapping errors, 114, 274-275, 604-605
Trim() method, 227-228, 288, 325
TrimEnd() method, 122, 325, 514
TrimStart() method, 29-32, 288, 325
TrimToSize() method
- lists, 129, 770, 835-839
- queues, 538, 1023, 1118

true operator, 63, 166-171, 554
True/false evaluations, 63, 147, 166-171
Try blocks, 162-164, 275, 605
TryParse() method
- attributes, 90-91, 539, 888-890
- conversions, 89-91, 290, 538-539
- dynamic invocation, 90-91, 538-539, 888-890
- enum, 90-91, 538-539, 888-890

Tuples
- declaring and assigning, 106-108, 111-112, 359
- generics, 111, 642-645, 803
- GetHashCode() and Equals() overriding, 515-520, 850-852
- groups, 106-108, 644, 807
- pattern matching, 106-108, 433-435, 438
- projections, 770, 800, 815
- query expressions, 106-108, 796, 814-815
- return values, 111, 254, 644
- System.ValueTuple type, 107-111, 643-645, 803
- anonymous type replacement, 111, 803, 807
- generics, 111, 642-645, 803

Two's complement notation, 138-139, 182, 183-187
Two-dimensional arrays, 115-120, 123-124, 130-132
Type checking, 419-421, 651, 1119
Type class
- generic parameters, 635, 663-664, 891-893
- metadata, 655, 882-885, 1125-1127
- type parameters, 655-656, 678, 892

Type parameters
- cascading, 635, 642-646, 656
- constraints. See Constraints
- contravariance, 651, 660, 672-675
- covariance, 664, 669, 672-675
- generics, 635, 663-664, 892
- lists, 631-635, 642, 664
- multiple, 635, 642-645, 663-664
- names, 228, 635, 664
- null modifier, 82-83, 97, 654-656
- obtaining, 646, 664-666, 891-892
- type of, 635, 663-666, 891-893

Type safety
- anonymous types, 703, 797-805
- CLI, 1109, 1119-1121, 1125
- generics, 638, 664, 681
- managed execution, 922, 1093-1095, 1120

typeof() method
- attributes, 885, 896-898, 901-903
- locks, 528, 885, 1053-1057
- reflection, 883-885, 888-890, 1119

Types
- anonymous. See Anonymous types
- arrays. See Arrays
- Boolean, 63, 166, 169-170
- character, 64-66, 134, 144
- conversion overview, 84, 88-89, 523
- CTS, 134, 1125, 1126

decimal, 54-60, 145
declarations, 547, 885, 1125
default, 52-53, 269, 640
definitions, 93, 134, 1125
delegates, 690, 693-694, 726-728
description, 93, 134, 1125
dynamic, 664, 920-922, 926
exceptions, 280, 599-601, 605
floating-point types, 53-56, 145-148
generics, 623, 631-635, 677
hardcoding values, 57, 94-96, 373
hexadecimal notation, 60-61, 64-66, 144
implicitly typed local variables, 249, 488, 798-799
integers, 56-57, 96, 144
lambda expressions, 697-699, 702-703, 716-720
local variables, 25-27, 164-165, 714
methods, 134, 295-297, 664
nested, 221, 570, 643-646
null allowed, 82-83, 173, 628
null checks, 82, 98-100, 173-175
nullable reference, 83, 96-97, 100-102
nullable values, 96-97, 505, 629
overview, 93, 134, 832-837
parameters, 635, 664, 891-893
parsing, 88-91, 433, 888-890
pattern matching, 423-424, 433-438, 635
reference, 93-96, 134, 491-492
retrieving, 134, 800, 883-885
return values, 84, 229-231, 254
strings. See Strings
tuples, 106-108, 111, 642-644
Unicode standard, 64-65, 66, 144
unmanaged, 654, 1078-1083, 1093
from unmanaged structs, 654, 1078-1084, 1097
value. See Values and value types
verifying, 421, 651, 1119
well-formed. See Well-formed types

U

uint type, 56, 96, 1081
ulong type, 53, 95-96, 1097
Unary expression trees, 139-141, 716-720, 723-725
Unary operators
overriding, 138-143, 548, 552
plus and minus, 138-139, 141-143, 149-150
UnauthorizedAccessException type, 315-316, 618, 1119
Unboxing operation, 6, 523-527, 531-532
Unchecked conversions, 88-89, 391-392, 523
unchecked keyword, 15-16, 19, 701
Uncompress() method, 246-248, 357-358, 445
Underscores (_)
digit separators, 59, 66, 138-139
field names, 17-19, 59, 323
identifier names, 17-19, 59, 323
variable names, 17-19, 59, 323
VB line continuation character, 59, 65-70, 213
Undo operations
generics for, 623-624, 631-634, 681
stacks for, 624-626, 631-634, 1100
Unhandled exceptions
description, 604, 618, 962
parallel loops, 962-964, 1028-1030, 1035
reporting, 274, 604, 962
tasks, 604, 622, 958-962
threads, 622, 958-959, 962-964
UnhandledException type, 604, 610-614, 958-962
Unicode standard
character representation, 64-65, 66, 144
escape sequence, 64-69, 143-144
localized applications, 64-66, 1113, 1120
Union() operator, 139-143, 169, 788
Unity .NET frameworks, 44-47, 1109-1113, 1128
Unmanaged code and types
buffer overflows, 1080, 1093-1095, 1120
calls into, 1077-1080, 1083, 1120
constraints, 652-655, 1083, 1120
description, 1077-1080, 1083, 1120
exceptions, 609-610, 622, 1120

guidelines, 39, 1077-1078, 1120
parameters, 1077-1080, 1093, 1120
performance, 39, 1077-1080, 1120
pointers, 1080, 1090-1093, 1097-1099
resource cleanup, 594-595, 1087, 1118-1120
stack allocation, 1077-1080, 1093, 1100
structs, 1078-1080, 1083, 1093-1097
UnobservedTaskException type, 958-961, 964, 969
Unreachable end points in methods, 230, 246, 462
Unsafe code
delegates, 684, 693, 1090-1095
platform interoperability. See Platform interoperability
pointers, 1077, 1092-1101, 1104
unsafe modifier, 330, 1055-1057, 1093-1095
Unwrap() method, 531, 980-982, 989
UserName class, 298-299, 568, 895
ushort type, 53, 64, 654
using directive
aliasing, 210, 235-241, 898
deterministic finalization, 581-584, 587-590, 596
namespaces, 222, 235-240, 568-570
nested, 207, 234, 237-239
strings, 207-209, 213, 235-240
using statement, 193, 239, 428
using static directive
strings, 237-241, 360, 373
working with, 237-241, 360, 369-370

V

Validation, 324-325, 393, 478
value keyword for properties, 309, 318-320, 333
Values and value types
boxing, 523-524, 530-532, 545
CTS, 92-93, 134, 1125
description, 93, 134, 494
enums. See enum values
generic instantiation, 494, 630, 677-679
hardcoding, 57, 94-96, 487-489
immutable, 96, 488-489, 505-507
inheritance and interfaces, 482, 487, 636
from iterators, 488, 545, 863-868
lock statement, 96, 487-489, 528
new operator, 494, 511, 548
overview, 93, 134, 487-489
parameters, 134, 487-489, 494
vs. reference, 93-96, 134, 488-493
structs, 487-488, 544-545, 1097
variables, 94-96, 134, 488-489
Values property for sorted collections, 835-837, 845-849, 852
ValueTask<T> class, 659, 949, 990-994
ValueTuple class, 107-111, 359, 644
ValueType class
var keyword, 102-103, 487-489, 799
pattern matching, 433-434, 532, 545
type declarations, 487, 533, 545-547
Variable parameters, 102-103, 260, 269
Variables
anonymous types, 707, 797-805
copying, 488-489, 492, 530
declaring, 25-27, 102-103, 193
description, 25-27, 116, 193
for loops, 150, 188-195, 715
foreach loops, 192-195, 715, 863-864
immutable, 373-375, 488, 505-507
implicitly typed local, 249, 488, 798-799
instance fields, 302-306, 360, 363
lambda expressions, 697-699, 702-706, 716-720
parameters, 224, 228, 248
query expressions, 716-718, 808-812, 831
reference types, 93-96, 488, 491-492
scope, 164-165, 362, 716
thread synchronization, 941-942, 1046, 1061
tuples, 104-108, 111, 644
value types, 93-96, 488-489, 545
working with, 25-27, 94, 302
Variadic generic types, 638, 642-645, 664
Variance, 102-103, 673-676, 708
Variants, 297, 488, 708
Verbatim strings

interpolation, 32, 67, 70-74
working with, 66-73, 143
Versioning
C# 8.0 and later, 101, 476, 483
encapsulation and polymorphism with protected interface members, 443-446, 475-476, 484
.NET frameworks, 44-47, 561, 1110-1113
overview, 113, 488, 1118
pre C# 8.0, 101, 462-467, 476
refactoring features, 227, 255-257, 488
Vertical bars (|)
bitwise operators, 168-170, 183, 184-187
compound assignment, 140-141, 149, 169-170
logical operators, 138, 167-170, 184-185
overriding, 65-66, 169-170, 216
Vertical tabs escape sequence, 65, 66-72, 160
Virtual abstract members, 401, 410-414, 475
Virtual Execution System (VES)
CLI, 38, 1106, 1108
description, 38, 1106-1108, 1125
managed execution, 38-39, 1106-1108, 1116
runtime, 38-39, 1106-1108, 1116
Virtual functions in C++, 401, 443, 1078-1080
Virtual methods
overridden, 398-401, 413, 455
overriding, 398-401, 475, 661
virtual modifier
base class overriding, 401-403, 406-408, 661
interface refactoring, 401, 443-446, 455
VirtualAllocEx() method, 1080, 1081-1082, 1086-1088
VirtualMemoryManager class, 1078-1081, 1087-1088, 1118
Visual Basic language vs
global methods, 228, 233, 237
global variables and functions, 2, 94, 233
Imports directive, 50, 234-241
line-based statements, 13-14, 217-219, 225
Me keyword, 14-15, 63, 319
named arguments, 224, 228, 269-270
redimensioning arrays, 112, 115-118, 130-132
variable declarations, 25-27, 94, 799
void type, 84, 94-96, 134
Visual code editors, 3-4, 7, 572
Visual Studio 2019
CLI, 4-7, 560-561, 1109
code building and executing, 2-4, 7, 10-12
debugging with, 4, 7, 1095
NuGet references, 4, 44, 559-564
preprocessor directives, 207-214, 237
project and library references, 11, 557-564
working with, 2-4, 7, 1095
void type
asynchronous method returns, 84, 990-993, 1002-1006
partial methods, 379-384, 664, 802-803
pointers, 84, 95-96, 1096-1099
return values, 84, 230, 254
strings, 53, 80-84, 489
Volatile fields, 373-375, 488-490, 1055
volatile modifier, 315-316, 710, 1053-1055

W

Wait() method
asynchronous requests, 1003-1005, 1066-1068, 1074
asynchronous tasks, 947, 1003-1005, 1074
reset events, 954-956, 1003-1005, 1064-1068
WaitAll() method
asynchronous tasks, 947, 1003-1005, 1018
thread synchronization, 1005, 1063-1068, 1076
WaitAny() method
asynchronous tasks, 947, 1003-1005, 1074
thread synchronization, 1018, 1063, 1066-1068
WaitForPendingFinalizers() method, 581-584, 588-591, 595-596

WaitHandle class, 1063, 1066-1068, 1087-1088
WaitOne() method, 1003-1005, 1063, 1066-1068
Warnings
  method overriding, 211-212, 402-403, 915
  preprocessor directives, 207-214
Weak references in garbage collection, 576-577, 578, 595-596
Web class, 245, 557, 1113-1114
Well-formed types
  assembly references, 559, 568, 1097
  encapsulation, 394, 545-547, 1119
  garbage collection, 576-578, 1100, 1115-1118
  lazy initialization, 510, 598, 802-805
  namespaces, 50-51, 221, 567-570
  overriding object members, 401, 414, 547
  overriding operators, 421, 545-548, 552-554
  overview, 134, 293, 545-547
  resource cleanup. See Resource cleanup
  XML comments, 35-37, 547, 571-574
Where() operator, 721, 768-771, 820
while loops, 188-194, 203-204, 1021
Whitespace
  overview, 15-17, 23, 72
  in strings, 23, 66-73, 288
  trimming, 23, 70-72, 288
Win32Exception() method, 601, 604, 1085
Windows class, 4, 7, 221
Windows Forms, 7, 45, 1075
Windows Presentation Foundation (WPF)
  description, 45, 463, 1109-1114
  high-latency example, 934, 1016, 1074-1076
Windows UI applications, 7, 245, 1109-1113
WithCancellation() method, 963-969, 1030, 1036-1038
Work stealing, 317, 937, 973-975
WPF (Windows Presentation Foundation)
  description, 245, 463, 1109-1114
  high-latency example, 934, 1016, 1074-1075
Wrapped exceptions, 280, 618, 622
Wrappers with API calls, 45, 1078, 1086
Write() method
  console output, 28-31, 225, 723
  invoking, 31, 225, 311-312
  newlines, 31, 78, 225
  strings, 31-32, 75, 514
Write-only properties, 326, 374-375, 506-507
WriteLine() method
  console output, 31, 78, 225
  invoking, 78, 225, 240
  newlines, 31, 78, 225
  round-trip formatting, 31-32, 62, 75-78
  strings, 31-32, 78, 225

X

Xamarin .NET frameworks, 44-47, 1109-1113, 1128
Xamarin compiler, 38-40, 44, 1114
XAML (Extended Application Markup Language), 35-37, 1109, 1114
xcopy deployment, 508-509, 557, 1123
XML (Extensible Markup Language)
  binding elements, 35-37, 571-574, 928
  comments, 35, 36, 571-574
  delimited comments, 35, 36, 571-574
  documentation files, 35-37, 571-574, 928
  namespace, 35-37, 221, 571-574
  overview, 35, 36-37, 571-574
  reflection, 35-37, 571-573, 882
  single-line comments, 35-36, 37, 571-574
XmlSerializer class, 35-37, 571-574, 617
XOR (exclusive OR) operators
  bitwise, 170, 183, 184-187
  logical, 170, 184, 185-187

Y

yield statements
  contextual keyword, 428, 811, 877-879
  requirements, 702, 870, 877-879
  yield break, 13, 870, 875-879

yield return, 230, 866-870, 875-879
Yielding values from iterators, 863-873, 877-879

Z

Zero-based arrays, 112, 118-127, 132
Zeros
division by, 54-55, 138-140, 148
floating-point types, 53-56, 145-148

Index of 8.0 Topics

A

Abstract classes
- defining, 476
- vs. interfaces, 482
- overview, 476
- polymorphism, 443

Aggregation, 476-478

async returns, 994, 1004

Asynchronous streams, 994, 1004

await using statement, 587, 764

B

Buffer overflows
- C# vs. C++, 86
- managed execution, 86
- type safety, 86
- unsafe code, 1094

C

Constraints
- constructor, 476
- classes, 655
- delegates, 476
- generic methods, 476
- inheritance, 443
- interfaces, 464
- limitations, 478
- multiple, 478
- notnull, 655
- operators, 143
- overview, 478
- structs, 506
- type parameters, 655
- unmanaged, 654

Constructor overloading, 476-478

D

Delegate invocation, 467, 476

I

implementation
- versioning, 476

Index from end operator, 121-122

Interfaces
- vs. abstract classes, 482
- vs. attributes, 482
- collections, 463
- constraints, 655
- converting between implementing classes, 476
- default members, 467
- deriving, 458
- diagramming, 463
- duplicating, 476
- extension methods, 461
- generics, 443
- implementation, 476
- inheritance, 461
- overview, 476
- polymorphism, 443
- purpose, 476
- refactoring features, 467
- value types, 482
- versioning. See Versioning

is operator

pattern matching example, 441
pattern matching overview, 441
positional pattern matching, 441
properties, 548
recursive pattern matching, 441
tuples, 815
type, 419
types with, 419
is { } property pattern
.NET frameworks, 318

N

.NET frameworks
description, 46
garbage collection, 577
multiple, 47
overview, 46
standard, 1113
versioning, 483
Non-nullable reference type properties, 83, 101
Non-nullable reference types, 83, 101
Null assignment for reference types, 83, 100
Null-coalescing assignment, 175-176
Null-coalescing operator, 175-176
Null-forgiving operator, 173-175
nullable modifier
reference types, 83
strings, 83
type declarations, 101
overview, 83
preprocessor directives, 214
Nullable reference types
default values, 101
nullable modifier, 83
introduced, 101

P

Pattern matching
is operator, 441
with null, 431
polymorphism, 441
positional, 435
properties, 431
recursive, 441
switch statements, 441
tuples, 433
types, 433
Pattern matching types, 433, 441
Positional pattern matching, 435, 441
Property matching, 476-478

R

Ranges, 476-478
Read-only struct members, 506-507
Read-only structs, 506-507
Resource cleanup
description, 576
deterministic finalization, 584
exception propagating from constructors, 584
finalizers, 584
forcing, 576
garbage collection, 576
guidelines, 478
resurrecting objects, 576
Reverse accessing arrays, 121, 125

S

String interpolation, 32, 77

T

Task-based Asynchronous Pattern, 935, 975
Task-based Asynchronous Pattern (TAP)
async/await syntax, 1019
async return types, 1019
asynchronous lambdas and local functions, 1019
asynchronous streams, 1019
await operator, 1019
await using statement, 1019
complexity, 1019
description, 1019
high-latency example, 1074
IAsyncDisposable, 1019
LINQ with IAsyncEnumerable, 1019
overview, 1019

return types, 1019
synchronization context, 1019
synchronous issue example, 1019
task schedulers, 1019
ValueTask<T> returns, 1019
Windows UI, 1074

U

Unmanaged constraints, 654-655

Index of 9.0 Topics

F

Fit and finish features
- Target-typed new expressions, 682
- static anonymous functions, 712
- Target-typed conditional expressions, 160
- Covariant return types, 401
- Extension GetEnumerator support for foreach loops, 760
- Lambda discard parameters, 701
- Attributes on local functions, 881

I

Init only setters, 338, 340

P

Pattern matching enhancements, 433, 441

Performance and interop
- Native sized integers, 56
- Function pointers, 1090
- Suppress emitting localsinit flag, 712

R

Records, 499-500

S

Support for code generators
- Module initializers, 512
- New features for partial methods, 384

T

Top-level statements, 160, 247

Index of 10.0 Topics

A

Allow const interpolated strings, 67, 70
Allow both assignment and declaration in the same deconstruction, 336, 359

C

CallerArgumentExpression attribute, 918-919

E

Enhanced #line pragma, 212-213
Extended property patterns, 424, 432

F

File-scoped namespace declaration, 568-569

G

global using directives, 236-237

I

Improved definite assignment, 27, 359
Improvements of structure types, 487, 545
Improvements on lambda expressions, 697-699
Interpolated string handlers, 32, 67

R

Record structs, 487, 499
Record types can seal ToString(), 499, 514

Index of 11.0 Topics

A

Auto-default structs, 510-511

E

Extended nameof scope, 569, 894

F

File-local types, 52-53

G

Generic attributes, 881, 897
Generic math support, 52, 56

I

Improved method group conversion to delegate, 684, 693

L

List patterns, 424, 436

N

Newlines in string interpolation expressions, 67, 70
Numeric IntPtr, 56, 1081

P

Pattern match Span<char> on a constant string, 69, 425

R

Raw string literals, 67, 70
ref fields and scoped ref, 375, 510
Required members, 350, 478

U

UTF-8 string literals, 67-69

Index of 12.0 Topics

A

Alias any type, 241, 651

D

Default lambda parameters, 269, 701

I

Inline arrays, 123, 1084

P

Primary constructors, 512, 639