Essential C# 6.0

"Welcome to one of the greatest collaborations you could dream of in the world of C# books—and probably far beyond!"

—From the Foreword by Mads Torgersen, C# Program Manager, Microsoft

Mark Michaelis with Eric Lippert

IntelliTect

Free Sample Chapter

Share with Others
Essential C# 6.0
Titles in The Addison-Wesley Microsoft Technology Series address the latest Microsoft technologies used by developers, IT professionals, managers, and architects. Titles in this series cover a broad range of topics, from programming languages to servers to advanced developer techniques. The books are written by thought leaders and experts in their respective communities, including many MVPs and RDs. The format of this series was created with ease-of-use in mind, incorporating features that make finding topics simple; visually friendly charts and fonts; and thorough and intuitive indexes.

With trusted authors, rigorous technical reviews, authoritative coverage, and independent viewpoints, the Microsoft Community can rely on Addison-Wesley to deliver the highest quality technical information.

Visit informit.com/mstechseries for a complete list of available products.

Make sure to connect with us! informit.com/socialconnect
To my family: Elisabeth, Benjamin, Hanna, and Abigail.
You have sacrificed a husband and daddy for countless hours of writing, frequently at times when he was needed most.
Thanks!
This page intentionally left blank
Contents at a Glance

Figures xv
Contents ix
Tables xvii
Foreword xix
Preface xxiii
Acknowledgments xxxv
About the Authors xxxvii

1 Introducing C#  1
2 Data Types  35
3 Operators and Control Flow  89
4 Methods and Parameters  161
5 Classes  217
6 Inheritance  289
7 Interfaces  325
8 Value Types  351
9 Well-Formed Types  383
10 Exception Handling  433
11 Generics  455
12 Delegates and Lambda Expressions  505
Contents at a Glance

13 Events 543
14 Collection Interfaces with Standard Query Operators 571
15 LINQ with Query Expressions 621
16 Building Custom Collections 643
17 Reflection, Attributes, and Dynamic Programming 683
18 Multithreading 731
19 Thread Synchronization 815
20 Platform Interoperability and Unsafe Code 849
21 The Common Language Infrastructure 877
A Downloading and Installing the C# Compiler and CLI Platform 897
B Tic-Tac-Toe Source Code Listing 903
C Interfacing with Multithreading Patterns prior to the TPL and C# 6.0 909
D Timers Prior to the Async/Await Pattern of C# 5.0 939

Index 945
Index of 6.0 Topics 983
Index of 5.0 Topics 991
Index of 4.0 Topics 995
Index of 3.0 Topics 1001
Contents

Figures xv
Tables xvii
Foreword xix
Preface xxiii
Acknowledgments xxxv
About the Authors xxxvii

1 Introducing C# 1
   Hello, World  2
   C# Syntax Fundamentals  4
   Console Input and Output  18

2 Data Types  35
   Fundamental Numeric Types  36
   More Fundamental Types  45
   null and void  58
   Categories of Types  61
   Nullable Modifier  64
   Conversions between Data Types  65
   Arrays  71

3 Operators and Control Flow  89
   Operators  90
   Introducing Flow Control  107
   Code Blocks ({} )  114
# Contents

- Code Blocks, Scopes, and Declaration Spaces 116
- Boolean Expressions 118
- Bitwise Operators ($\ll$, $\gg$, $|$, $\&, \^$, $\sim$) 128
- Control Flow Statements, Continued 134
- Jump Statements 146
- C# Preprocessor Directives 152

## 4 Methods and Parameters 161
- Calling a Method 162
- Declaring a Method 169
- The using Directive 175
- Returns and Parameters on `Main()` 180
- Advanced Method Parameters 183
- Recursion 192
- Method Overloading 194
- Optional Parameters 197
- Basic Error Handling with Exceptions 202

## 5 Classes 217
- Declaring and Instantiating a Class 221
- Instance Fields 225
- Instance Methods 227
- Using the `this` Keyword 228
- Access Modifiers 235
- Properties 237
- Constructors 254
- Static Members 265
- Extension Methods 275
- Encapsulating the Data 277
- Nested Classes 281
- Partial Classes 284

## 6 Inheritance 289
- Derivation 290
- Overriding the Base Class 302
Abstract Classes 314
All Classes Derive from System.Object 320
Verifying the Underlying Type with the is Operator 321
Conversion Using the as Operator 322

7 Interfaces 325
Introducing Interfaces 326
Polymorphism through Interfaces 327
Interface Implementation 332
Converting between the Implementing Class and Its Interfaces 338
Interface Inheritance 338
Multiple Interface Inheritance 341
Extension Methods on Interfaces 341
Implementing Multiple Inheritance via Interfaces 343
Versioning 346
Interfaces Compared with Classes 347
Interfaces Compared with Attributes 349

8 Value Types 351
Structs 355
Boxing 362
 Enums 371

9 Well-Formed Types 383
Overriding object Members 383
Operator Overloading 395
Referencing Other Assemblies 403
Defining Namespaces 409
XML Comments 413
Garbage Collection 418
Resource Cleanup 421
Lazy Initialization 429

10 Exception Handling 433
Multiple Exception Types 433
Catching Exceptions 436
General Catch Block 440
Guidelines for Exception Handling 443
Defining Custom Exceptions 446
Rethrowing a Wrapped Exception 449

11 Generics 455
C# without Generics 456
Introducing Generic Types 461
Constraints 473
Generic Methods 486
Covariance and Contravariance 491
Generic Internals 498

12 Delegates and Lambda Expressions 505
Introducing Delegates 506
Lambda Expressions 516
Anonymous Methods 522
General-Purpose Delegates: System.Func and System.Action 524

13 Events 543
Coding the Observer Pattern with Multicast Delegates 544
Events 558

14 Collection Interfaces with Standard Query Operators 571
Anonymous Types and Implicitly Typed Local Variables 572
Collection Initializers 578
What Makes a Class a Collection: IEnumerable<T> 582
Standard Query Operators 588

15 LINQ with Query Expressions 621
Introducing Query Expressions 622
Query Expressions Are Just Method Invocations 640

16 Building Custom Collections 643
More Collection Interfaces 644
Primary Collection Classes 646
Providing an Indexer  663
Returning "null" or an Empty Collection  666
Iterators  667

17 Reflection, Attributes, and Dynamic Programming  683
Reflection  684
nameof Operator  694
Attributes  696
Programming with Dynamic Objects  719

18 Multithreading  731
Multithreading Basics  734
Working with System.Threading  741
Asynchronous Tasks  749
Canceling a Task  768
The Task-Based Asynchronous Pattern  775
Executing Loop Iterations in Parallel  798
Running LINQ Queries in Parallel  809

19 Thread Synchronization  815
Why Synchronization?  817
Timers  845

20 Platform Interoperability and Unsafe Code  849
Platform Invoke  850
Pointers and Addresses  862
Executing Unsafe Code via a Delegate  872
Using the Windows Runtime Libraries from C#  873

21 The Common Language Infrastructure  877
Defining the Common Language Infrastructure  878
CLI Implementations  879
C# Compilation to Machine Code  879
Runtime  883
Application Domains  888
Assemblies, Manifests, and Modules  888
Contents

Common Intermediate Language  891
Common Type System  892
Common Language Specification  893
Base Class Library  893
Metadata  894

A  Downloading and Installing the C# Compiler and CLI Platform  897
    Microsoft's .NET  897

B  Tic-Tac-Toe Source Code Listing  903

C  Interfacing with Multithreading Patterns prior to the TPL and C# 6.0  909

D  Timers Prior to the Async/Await Pattern of C# 5.0  939

Index  945
Index of 6.0 Topics  983
Index of 5.0 Topics  991
Index of 4.0 Topics  995
Index of 3.0 Topics  1001
Figures

**Figure 2.1:** Value Types Contain the Data Directly  62
**Figure 2.2:** Reference Types Point to the Heap  63

**Figure 3.1:** Corresponding Placeholder Values  128
**Figure 3.2:** Calculating the Value of an Unsigned Byte  129
**Figure 3.3:** Calculating the Value of a Signed Byte  129
**Figure 3.4:** The Numbers 12 and 7 Represented in Binary  131
**Figure 3.5:** Collapsed Region in Microsoft Visual Studio .NET  159

**Figure 4.1:** Exception-Handling Program Flow  206

**Figure 5.1:** Class Hierarchy  220

**Figure 6.1:** Refactoring into a Base Class  291
**Figure 6.2:** Simulating Multiple Inheritance Using Aggregation  300

**Figure 7.1:** Working around Single Inheritances with Aggregation and Interfaces  345

**Figure 8.1:** Value Types Contain the Data Directly  352
**Figure 8.2:** Reference Types Point to the Heap  354

**Figure 9.1:** Identity  390
**Figure 9.2:** XML Comments As Tips in Visual Studio IDE  414
Figures

Figure 12.1: Delegate Types Object Model 513
Figure 12.2: Anonymous Function Terminology 516
Figure 12.3: The Lambda Expression Tree Type 536
Figure 12.4: Unary and Binary Expression Tree Types 537

Figure 13.1: Delegate Invocation Sequence Diagram 553
Figure 13.2: Multicast Delegates Chained Together 555
Figure 13.3: Delegate Invocation with Exception Sequence Diagram 556

Figure 14.1: A Class Diagram of IEnumerator<T> and IEnumerator Interfaces 584
Figure 14.2: Sequence of Operations Invoking Lambda Expressions 599
Figure 14.3: Venn Diagram of Inventor and Patent Collections 603

Figure 16.1: Generic Collection Interface Hierarchy 645
Figure 16.2: List<> Class Diagrams 647
Figure 16.3: Dictionary Class Diagrams 654
Figure 16.4: SortedList<> and SortedDictionary<> Class Diagrams 661
Figure 16.5: Stack<T> Class Diagram 662
Figure 16.6: Queue<T> Class Diagram 662
Figure 16.7: LinkedList<T> and LinkedListNode<T> Class Diagrams 663
Figure 16.8: Sequence Diagram with yield return 672

Figure 17.1: MemberInfo Derived Classes 691
Figure 17.2: BinaryFormatter Does Not Encrypt Data 715

Figure 18.1: Clock Speeds over Time 732
Figure 18.2: CancellationTokenSource and CancellationToken Class Diagrams 771

Figure 20.1: Pointers Contain the Address of the Data 864

Figure 21.1: Compiling C# to Machine Code 882
Figure 21.2: Assemblies with the Modules and Files They Reference 890

Figure C.1: APM Parameter Distribution 912
Figure C.2: Delegate Parameter Distribution to BeginInvoke() and EndInvoke() 925
Tables

**Table 1.1:** C# Keywords  5  
**Table 1.2:** C# Comment Types  24  
**Table 1.3:** C# and .NET Versions  29  

**Table 2.1:** Integer Types  36  
**Table 2.2:** Floating-Point Types  38  
**Table 2.3:** decimal Type  38  
**Table 2.4:** Escape Characters  47  
**Table 2.5:** string Static Methods  52  
**Table 2.6:** string Methods  52  
**Table 2.7:** Array Highlights  73  
**Table 2.8:** Common Array Coding Errors  86  

**Table 3.1:** Control Flow Statements  108  
**Table 3.2:** Relational and Equality Operators  120  
**Table 3.3:** Conditional Values for the XOR Operator  122  
**Table 3.4:** Preprocessor Directives  153  
**Table 3.5:** Operator Order of Precedence  160  

**Table 4.1:** Common Namespaces  165  
**Table 4.2:** Common Exception Types  210  

**Table 6.1:** Why the New Modifier?  308  
**Table 6.2:** Members of System.Object  320
Tables

Table 7.1: Comparing Abstract Classes and Interfaces 348
Table 8.1: Boxing Code in CIL 363
Table 9.1: Accessibility Modifiers 409
Table 12.1: Lambda Expression Notes and Examples 521
Table 14.1: Simpler Standard Query Operators 618
Table 14.2: Aggregate Functions on System.Linq.Enumerable 618
Table 17.1: Deserialization of a New Version Throws an Exception 717
Table 18.1: List of Available TaskContinuationOptions Enums 758
Table 18.2: Control Flow within Each Task 785
Table 19.1: Sample Pseudocode Execution 818
Table 19.2: Interlocked’s Synchronization-Related Methods 829
Table 19.3: Execution Path with ManualResetEvent Synchronization 838
Table 19.4: Concurrent Collection Classes 840
Table 21.1: Primary C# Compilers 880
Table 21.2: Common C#-Related Acronyms 895
Foreword

Welcome to one of the greatest collaborations you could dream of in the world of C# books—and probably far beyond! Mark Michaelis’ Essential C# series was already a classic when, for the previous edition, he teamed up with famous C# blogger Eric Lippert—a masterstroke.

You may think of Eric as writing blogs and Mark as writing books, but that is not how I first got to know them.

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 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, and busily pulling together his own picture.

As a newcomer to the C# community, I think I may have met a lot of people for the first time at that conference—people with whom I have since formed great relationships. But to be honest, I don’t remember it—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 because he is forthright and never afraid to speak his mind. If something passes the “Mark Test,” then we know we can start feeling pretty good about it!

These are the same qualities that make Mark such a great writer. He goes right to the essence and communicates with great integrity, no sugarcoating, and a keen eye for practical value and real-world problems.

Eric is, of course, my former colleague of seven years on the C# team. He’d been there much longer than I had, and the first I recall of him, he was explaining to the team how to untangle a bowl of spaghetti. More precisely, our C# compiler code base at the time was in need of some serious architectural TLC and was exceedingly hard to add new features to—something we desperately needed to be able to do with LINQ. Eric had been investigating what kind of architecture we ought to have (Phases! We didn’t even really have those!), and more importantly, how to get from here to there, step by step. The remarkable thing was that as complex as this was, and as new as I was to the team and the code base, I immediately understood what he was saying!

You may recognize from his blogs the super-clear and well-structured untangling of the problem, the convincing clarity of enumerated solutions, and the occasional unmitigated hilarity. Well, you don’t know the half of it! Every time Eric was grappling with a complex issue and was sharing his thoughts about it with the team, his emails about it were just as meticulous and every bit as hilarious. You fundamentally couldn’t ignore an issue raised by Eric because you couldn’t wait to read his prose about it. They were even purple, too! So I essentially got to enjoy a continuous supply of what amounts to unpublished installments of his blog, as well as, of course, his pleasant and insightful presence as a member of the C# compiler team and language design team. In his post-Microsoft days, Eric has continued to be a wonderful, insightful voice with a lot more influence on our decisions than he probably knows.
In summary, I am truly grateful to get to work with these two amazing people on a regular basis: Eric to help keep my thinking straight and Mark to help keep me honest. They share a great gift of providing clarity and elucidation, and by combining their “inside” and “outside” perspective on C#, their book reaches a new level of completeness. No one will help you get C# 6 like these two gentlemen do.

Enjoy!

—Mads Torgersen
C# Program Manager
Microsoft
This page intentionally left blank
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 is organized in such a way as to take you through similar paradigm shifts.

The beginning chapters of Essential C# 6.0 take you through sequential programming structure, in which statements are written in the order in which they are executed. The problem with this model is that complexity increases exponentially as the requirements increase. To reduce this complexity, code blocks may be 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, growing programs may quickly become unwieldy and require further abstraction. Object-oriented programming, discussed in Chapter 5, was a response intended to rectify this situation. In subsequent chapters of this book, 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 17) 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#, it 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# 6.0 and .NET Framework 4.6.

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.

A number of topics are not covered in this book. You won’t find coverage of topics such as ASP.NET, Entity Framework, smart client development such as WPF, distributed programming, and so on. Although these topics are relevant to the .NET Framework, to do them justice requires books of their own. Fortunately, Addison-Wesley’s .NET Development Series provides a wealth of writing on these topics. Essential C# 6.0 focuses on C# and the types within the Base Class Library. Reading this book will prepare you to focus on and develop expertise in any of the areas covered by the rest of the series.

Target Audience for This Book

My challenge with this book was to keep advanced developers awake while not abandoning beginners by using terms 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 arsenal—another arrow in their quiver, as it were. 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—one 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 well throughout your programming career.
• **Structured programmers:** Just as it’s best to learn a foreign language through immersion, so learning a computer language is most effective when you begin using that language before you know all of its intricacies. In this vein, this book begins with a tutorial that will be comfortable for those familiar with structured programming, and by the end of Chapter 4, developers in this category should feel at home writing basic control flow programs. However, the key to excellence for C# developers is not just memorizing syntax. Rather, 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 5’s Beginner Topics introduce classes and object-oriented development. The role filled by 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++ and Java programmers, and many experienced 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 similar to the C and C++ styles of languages that you already know.

• **C# professionals:** For those already versed in C#, this book provides a convenient reference for less frequently encountered syntax. Furthermore, it provides answers to language details and subtleties that are seldom addressed. Most importantly, 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, 4.0, 5.0, and now 6.0, some of the most prominent enhancements are as follows:

  – Implicitly typed variables (see Chapter 2)
  – Extension methods (see Chapter 5)
  – Partial methods (see Chapter 5)
  – Anonymous types (see Chapter 11)
  – Generics (see Chapter 11)
  – Lambda statements and expressions (see Chapter 12)
- Expression trees (see Chapter 12)
- Standard query operators (see Chapter 14)
- Query expressions (see Chapter 15)
- Dynamic programming (Chapter 17)
- Multithreaded programming with the Task Programming Library and async (Chapter 18)
- Parallel query processing with PLINQ (Chapter 18)
- Concurrent collections (Chapter 19)

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

**Features of This Book**

*Essential C# 6.0* is a language book that adheres to the core C# Language 6.0 Specification. To help you understand the various C# constructs, it provides numerous examples demonstrating each feature. Accompanying each concept are guidelines and best practices, ensuring that code compiles, avoids likely pitfalls, and achieves maximum maintainability.

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

**C# Coding Guidelines**

One of the more significant features in *Essential C# 6.0* is the inclusion of C# coding guidelines, as shown in the following example taken from Chapter 16:

**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 is able to 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 facilitate maintenance well into the future. The coding guidelines
highlight some of the key principles that readers will want to be sure to
incorporate into their development.

Code Samples
The code snippets in most of this text can run on any implementation of
the Common Language Infrastructure (CLI), including the Mono, DNX
Core, and Microsoft .NET platforms. Platform- or vendor-specific libraries
are seldom used, except when communicating important concepts relevant
only to those platforms (appropriately handling the single-threaded user
interface of Windows, for example). Any code that specifically requires C#
3.0, 4.0, or 5.0 compliance is called out in the C# version, and separate
indexes at the end of the book.

Here is a sample code listing.

Listing 1.17: Swapping the Indexed Placeholders and Corresponding Variables

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

The formatting is as follows.

- Comments are shown in italics.
  
  /* Display a greeting to the console
    using composite formatting. */

- Keywords are shown in bold.
  
  static void Main()

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

```csharp
System.Console.Write /* No new line */ (}
Highlighting can appear on an entire line or on just a few characters within a line.

    System.Console.WriteLine(
        "Your full name is {0} {1}.",
    
• Incomplete listings contain an ellipsis to denote irrelevant code that has been omitted.

    // ...

• Console output is the output from a particular listing that appears following the listing.

    OUTPUT 1.4

    >HeyYou.exe
    Hey you!
    Enter your first name: Inigo
    Enter your last name: Montoya

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.

You can find sample code at Intellitect.com/essentialcsharp and at informit.com/mstechseries. In addition, the code is available on Github—see http://itl.tc/EssentialCSharpSCC. Instructions for downloading the tools to compile the source code as well as the compilation instructions themselves are found in Appendix A.

You can also access the errata at http://Intellitect.com/essentialcsharp.

**Mind Maps**

Each chapter’s introduction includes a **mind map**, which serves as an outline that provides an at-a-glance reference to each chapter’s content. Here is an example (taken from Chapter 5).
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 code blocks and tabs will help you navigate through the text.

- Beginner Topics provide definitions or explanations specifically targeted to entry-level programmers.
- Advanced Topics enable experienced developers to focus on the material that is most relevant to them.
- Callout notes highlight key principles 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.
- Page-edge **begin** and **end tabs** denote material specific to C# versions; where that material continues over multiple pages, just the version number appears in the tab.
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# 6.0. Chapters 1–4 introduce structured programming, which enable you to start writing simple functioning code immediately. Chapters 5–9 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 11–13 introduce additional complexity-reducing constructs, handling common patterns needed by virtually all modern programs. This leads to dynamic programming with reflection and attributes, which is used extensively for threading and interoperability in the chapters that follow.

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

Here is a description of each chapter (in this list, chapter numbers shown in bold italics indicate the presence of C# 3.0–5.0 material).

- **Chapter 1—Introducing C#:** After presenting the C# HelloWorld program, this chapter proceeds to dissect it. This should familiarize readers with the look and feel of a C# program and provide details on how to compile and debug their own programs. Chapter 1 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#. This includes coverage of two type categories, value types and reference types, along with conversion between types and support for arrays.

- **Chapter 3—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 4—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 a parameter. Default parameter support was added in C# 4.0, and this chapter explains how to use this support.

- **Chapter 5—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 6—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 7—Interfaces:** This chapter demonstrates how interfaces are used to define the “versionable” interaction contract between classes. C# includes both explicit and implicit interface member implementation, enabling an additional encapsulation level not supported by most other languages.

- **Chapter 8—Value Types:** Although it is more common to define reference types, it is sometimes necessary to define value types that behave in a fashion similar to the primitive types built into C#. This chapter describes how to define structures, while exposing the idiosyncrasies they may introduce.

- **Chapter 9—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 the process of defining namespaces and the use of XML comments, and discusses how to design classes for garbage collection.

- **Chapter 10—Exception Handling:** This chapter expands on the exception-handling introduction from Chapter 4 and describes how exceptions follow a hierarchy that supports the creation of custom exceptions. It also includes some best practices on exception handling.

- **Chapter 11—Generics:** Generics is perhaps the core feature missing from C# 1.0. This chapter fully covers this 2.0 feature. In addition, C# 4.0 added support for covariance and contravariance—something covered in the context of generics in this chapter.
• **Chapter 12—Delegates and Lambda Expressions:** Delegates begin clearly distinguishing C# from its predecessors by defining patterns for handling events within code. This practice virtually eliminates the need for writing routines that poll. Lambda expressions are the key concept that make C# 3.0’s LINQ possible. Chapter 12 explains how lambda expressions build on the delegate construct by providing a more elegant and succinct syntax. This chapter forms the foundation for the new collection API discussed next.

• **Chapter 13—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 14—Collection Interfaces with Standard Query Operators:** The simple, yet elegantly powerful changes introduced in C# 3.0 begin to shine in this chapter, as we take a look at the extension methods of the new Enumerable class. This class makes available an entirely new collection API known as the standard query operators, which is discussed in detail here.

• **Chapter 15—LINQ with Query Expressions:** Using standard query operators alone results in some long statements that can be challenging to decipher. However, query expressions provide an alternative syntax that matches closely with SQL, as described in this chapter.

• **Chapter 16—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 17—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. It removed all type checking until runtime, a significant expansion of what can be done with C#.
• **Chapter 18—Multithreading**: Most modern programs require the use of threads to execute long-running tasks while ensuring they provide an active response to simultaneous events. As programs become more sophisticated, they must take additional precautions to protect data in these dynamic environments. Programming multithreaded applications is complex. This chapter discusses how to work with threads and provides best practices to avoid the problems that plague multithreaded applications.

• **Chapter 19—Thread Synchronization**: Building on the preceding chapter, Chapter 19 demonstrates some of the built-in threading pattern support that can simplify the explicit control of multithreaded code.

• **Chapter 20—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 21—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.

• **Appendix A—Downloading and Installing the C# Compiler and the CLI Platform**: This appendix provides instructions for setting up a C# compiler and the platform on which to run the code, Microsoft .NET or Mono.

• **Appendix B—Tic-Tac-Toe Source Code Listing**: This appendix provides a full listing of the Tic-Tac-Toe program referred to in Chapters 3 and 4.

• **C# 3.0, 4.0, 5.0, and 6.0 Indexes**: These indexes provide a quick reference for the features added in C# 3.0 through 6.0. They are specifically designed to help programmers quickly update their language skills to a more recent version.
Appendix C, Interfacing with Multithreading Patterns Prior to the TPL and C# 6.0, and Appendix D, Timers Prior to the Async/Await Pattern of C# 6.0, can be found on the book’s website, http://www.informit.com/title/9780134141046. Teaching resources that accompany this book will be made available to qualified instructors through Pearson’s Instructor Resource Center.

I hope that you find this book to be a great resource in establishing your C# expertise, and that you continue to reference it for the more obscure areas of C# and its inner workings.

—Mark Michaelis

Blog: http://IntelliTect.com/mark
Twitter: @Intellitect, @MarkMichaelis
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. By far, my family has made the biggest sacrifice to allow me to complete this project. 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. I would like to say it got easier with each edition but, alas, no: As the kids got older, life became more hectic, and without me Elisabeth was stretched to the breaking point almost all the time. A huge “Sorry” and ginormous “Thank You!”

Many technical editors reviewed each chapter in minute detail to ensure technical accuracy. I was often amazed by the subtle errors these folks managed to catch: Paul Bramsman, Kody Brown, Ian Davis, Doug Dechow, Gerard Frantz, Thomas Heavey, Anson Horton, Brian Jones, Shane Kercheval, Angelika Langer, Eric Lippert, John Michaelis, Jason Morse, Nicholas Paldino, Jon Skeet, Michael Stokesbary, Robert Stokesbary, John Timney, and Stephen Toub.

Eric is no less than amazing. His grasp of the C# vocabulary 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 no longer continues to fester. Eric has painstakingly reviewed every Essential C# chapter with amazing detail and precision. I am extremely grateful for
his contribution to making this book even better than the first two editions. Thanks, Eric! I can’t imagine anyone better for the job. You deserve all the credit for raising the bar from good to great.

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

Thanks to everyone at Addison-Wesley for their patience in working with me in spite of my occasional tendency to focus on everything else except the manuscript. Thanks to Vicki Rowland, Ellie Bru, Curt Johnson, and Joan Murray. Joan deserves a special medal for her patience given the number of times I delayed not only providing deliverables but even responding to emails. Vicki is no less than amazing in her ability to work with technical authors. I was so appreciative of the updated, fully stylized manuscripts she provided following the Essential C# 5.0 publication. It made writing Essential C# 6.0 so much easier than my updates of the prior editions.

Thanks also to Mads Torgersen, for his willingness to write the Foreword. Even if only half of what he says is true, I am greatly honored.
About the Authors

Mark Michaelis is the founder of IntelliTect and serves as the Chief Technical Architect and Trainer. Since 1996, he has been a Microsoft MVP for C#, Visual Studio Team System, and the Windows SDK; in 2007, he was recognized as a Microsoft Regional Director. He also serves on several Microsoft software design review teams, including C#, the Connected Systems, Office/SharePoint, and Visual Studio. Mark speaks at developer conferences and has written numerous articles and other books. He 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 stays busy with his family or training for another Ironman (having completed his first in 2008). Mark lives in Spokane, Washington, with his wife, Elisabeth, and three children, Benjamin, Hanna, and Abigail.

Eric Lippert works on tools for static analysis of C# at Coverity, now a part of Synopsys. Before joining Coverity, he was a principal developer on the C# compiler team at Microsoft. When not blogging or editing books about C#, Eric does his best to keep his tiny sailboat upright. He lives in Seattle, Washington, with his wife, Leah.
This page intentionally left blank
In this chapter, you will learn about operators, control flow statements, and the C# preprocessor. **Operators** provide syntax for performing different calculations or actions appropriate for the operands within the calculation. **Control flow statements** provide the means for conditional logic within a program or looping over a section of code multiple times. After introducing the `if` control flow statement, the chapter looks at the concept of Boolean expressions, which are embedded within many control flow statements. Included is mention of how integers cannot be converted...
(even explicitly) to bool and the advantages of this restriction. The chapter ends with a discussion of the C# preprocessor directives.

### Operators

Now that you have been introduced to the predefined data types (refer to Chapter 2), you can begin to learn more about how to use these data types in combination with operators to perform calculations. For example, you can make calculations on variables that you have declared.

#### BEGINNER TOPIC

**Operators**

Operators are used to perform mathematical or logical operations on values (or variables) called operands to produce a new value, called the result. For example, in Listing 3.1 the subtraction operator, -, is used to subtract two operands, the numbers 4 and 2. The result of the subtraction is stored in the variable `difference`.

**Listing 3.1: A Simple Operator Example**

```csharp
int difference = 4 - 2;
```

Operators are generally classified into three categories—unary, binary, and ternary, corresponding to the number of operands (one, two, and three, respectively). This section covers some of the most basic unary and binary operators. The ternary operators are introduced later in the chapter.

**Plus and Minus Unary Operators (+, -)**

Sometimes you may want to change the sign of a numerical value. In these cases, the unary minus operator (-) comes in handy. For example, Listing 3.2 changes the total current U.S. debt to a negative value to indicate that it is an amount owed.
Listing 3.2: Specifying Negative Values

```csharp
//National debt to the penny
decimal debt = -18125876697430.99M;
```

Using the minus operator is equivalent to subtracting the operand from zero.

The unary plus operator (+) rarely has any effect on a value. It is a superfluous addition to the C# language and was included for the sake of symmetry.

**Arithmetic Binary Operators (+, -, *, /, %)**

Binary operators require two operands. C# uses infix notation for binary operators: The operator appears between the left and right operands. The result of every binary operator other than assignment must be used somehow—for example, by using it as an operand in another expression such as an assignment.

---

**Language Contrast: C++—Operator-Only Statements**

In contrast to the rule mentioned previously, C++ will allow a single binary expression to form the entirety of a statement, such as `4+5`, to compile. In C#, only assignment, call, increment, decrement, and object creation expressions are allowed to be the entirety of a statement.

The subtraction example in Listing 3.3 illustrates the use of a binary operator—more specifically, an arithmetic binary operator. The operands appear on each side of the arithmetic operator, and then the calculated value is assigned. The other arithmetic binary operators are addition (+), division (/), multiplication (*), and remainder (%)—sometimes called the mod operator.

---

1. As of February 5, 2015, according to www.treasurydirect.gov.
2. The unary + operator is defined to take operands of type int, uint, long, ulong, float, double, and decimal (and nullable versions of those types). Using it on other numeric types such as short will convert its operand to one of these types as appropriate.
Chapter 3: Operators and Control Flow

**Listing 3.3: Using Binary Operators**

```csharp
class Division
{
    static void Main()
    {
        int numerator;
        int denominator;
        int quotient;
        int remainder;

        System.Console.Write("Enter the numerator: ");
        numerator = int.Parse(System.Console.ReadLine());

        System.Console.Write("Enter the denominator: ");
        denominator = int.Parse(System.Console.ReadLine());

        quotient = numerator / denominator;
        remainder = numerator % denominator;

        System.Console.WriteLine($"{numerator} / {denominator} = {quotient} with remainder
                                {remainder}";
    }
}
```

Output 3.1 shows the results of Listing 3.3.

**Output 3.1**

```
Enter the numerator: 23
Enter the denominator: 3
23 / 3 = 7 with remainder 2
```

In the highlighted assignment statements, the division and remainder operations are executed before the assignments. The order in which operators are executed is determined by their **precedence** and **associativity**. The precedence for the operators used so far is as follows:

1. *, /, and % have highest precedence.
2. + and - have lower precedence.
3. = has the lowest precedence of these six operators.

Therefore, you can assume that the statement behaves as expected, with the division and remainder operators executing before the assignment.
If you forget to assign the result of one of these binary operators, you will receive the compile error shown in Output 3.2.

**Output 3.2**

... error CS0201: Only assignment, call, increment, decrement, and new object expressions can be used as a statement

---

**BEGINNER TOPIC**

**Parentheses, Associativity, Precedence, and Evaluation**

When an expression contains multiple operators, it can be unclear precisely what the operands of each operator are. For example, in the expression $x+y*z$, clearly the expression $x$ is an operand of the addition and $z$ is an operand of the multiplication. But is $y$ an operand of the addition or the multiplication?

**Parentheses** allow you to unambiguously associate an operand with its operator. If you wish $y$ to be a summand, you can write the expression as $(x+y)*z$; if you want it to be a multiplicand, you can write $x+(y*z)$.

However, C# does not require you to parenthesize every expression containing more than one operator; instead, the compiler can use associativity and precedence to figure out from the context which parentheses you have omitted. **Associativity** determines how similar operators are parenthesized; **precedence** determines how dissimilar operators are parenthesized.

A binary operator may be “left-associative” or “right-associative,” depending on whether the expression “in the middle” belongs to the operator on the left or the right. For example, $a-b-c$ is assumed to mean $(a-b)-c$, and not $a-(b-c)$; subtraction is therefore said to be “left-associative.” Most operators in C# are left-associative; the assignment operators are right-associative.

When the operators are dissimilar, the **precedence** for those operators is used to determine which side the operand in the middle belongs to. For example, multiplication has higher precedence than addition and, therefore, the expression $x+y*z$ is evaluated as $x+(y*z)$ rather than $(x+y)*z$.

It is often still a good practice to use parentheses to make the code more readable even when use of parentheses does not change the meaning of the
expression. For example, when performing a Celsius-to-Fahrenheit temperature conversion, \((c*9.0/5.0)+32.0\) is easier to read than \(c*9.0/5.0+32.0\), even though the parentheses are completely unnecessary.

**Guidelines**

**DO** use parentheses to make code more readable, particularly if the operator precedence is not clear to the casual reader.

Clearly, operators of higher precedence must execute before adjoining operators of lower precedence: in \(x+y*z\), the multiplication must be executed before the addition because the result of the multiplication is the left-hand operand of the addition. However, it is important to realize that precedence and associativity affect only the order in which the *operators* themselves are executed; they do not in any way affect the order in which the *operands* are evaluated.

Operands are always evaluated from left to right in C#. In an expression with three method calls, such as \(A()\+B()\*C()\), first \(A()\) is evaluated, then \(B()\), then \(C()\); then the multiplication operator determines the product; and finally the addition operator determines the sum. Just because \(C()\) is involved in a multiplication and \(A()\) is involved in a lower-precedence addition does not imply that method invocation \(C()\) happens before method invocation \(A()\).

**Language Contrast: C++: Evaluation Order of Operands**

In contrast to the rule mentioned here, the C++ specification allows an implementation broad latitude to decide the evaluation order of operands. When given an expression such as \(A()\+B()\*C()\), a C++ compiler can choose to evaluate the function calls in any order, just so long as the product is one of the summands. For example, a legal compiler could evaluate \(B()\), then \(A()\), then \(C()\), then the product, and finally the sum.
**Using the Addition Operator with Strings**

Operators can also work with non-numeric operands. For example, it is possible to use the addition operator to concatenate two or more strings, as shown in Listing 3.4.

**Listing 3.4: Using Binary Operators with Non-numeric Types**

```csharp
class FortyTwo
{
    static void Main()
    {
        short windSpeed = 42;
        System.Console.WriteLine(
            "The original Tacoma Bridge in Washington\nwas "
            + "brought down by a "
            + windSpeed + " mile/hour wind."
        );
    }
}
```

Output 3.3 shows the results of Listing 3.4.

**Output 3.3**

The original Tacoma Bridge in Washington was brought down by a 42 mile/hour wind.

Because sentence structure varies among languages in different cultures, developers should be careful not to use the addition operator with strings that possibly will require localization. Similarly, although we can embed expressions within a string using C# 6.0’s string interpolation, localization to other languages still requires moving the string to a resource file, neutralizing the string interpolation. For this reason, you should use the addition operator sparingly, favoring composite formatting when localization is a possibility.

**Guidelines**

**DO** favor composite formatting over use of the addition operator for concatenating strings when localization is a possibility.
Using Characters in Arithmetic Operations

When introducing the char type in Chapter 2, we mentioned that even though it stores characters and not numbers, the char type is an integral type ("integral" means it is based on an integer). It can participate in arithmetic operations with other integer types. However, interpretation of the value of the char type is not based on the character stored within it, but rather on its underlying value. The digit 3, for example, is represented by the Unicode value 0x33 (hexadecimal), which in base 10 is 51. The digit 4 is represented by the Unicode value 0x34, or 52 in base 10. Adding 3 and 4 in Listing 3.5 results in a hexadecimal value of 0x67, or 103 in base 10, which is the Unicode value for the letter g.

Listing 3.5: Using the Plus Operator with the char Data Type

```csharp
int n = '3' + '4';
char c = (char)n;
System.Console.WriteLine(c); // Writes out g.
```

Output 3.4 shows the result of Listing 3.5.

Output 3.4

g

You can use this trait of character types to determine how far two characters are from each other. For example, the letter f is three characters away from the letter c. You can determine this value by subtracting the letter c from the letter f, as Listing 3.6 demonstrates.

Listing 3.6: Determining the Character Difference between Two Characters

```csharp
int distance = 'f' - 'c';
System.Console.WriteLine(distance);
```

Output 3.5 shows the result of Listing 3.6.

Output 3.5

3
**Special Floating-Point Characteristics**

The binary floating-point types, `float` and `double`, have some special characteristics, such as the way they handle precision. This section looks at some specific examples, as well as some unique floating-point type characteristics.

A `float`, with seven decimal digits of precision, can hold the value 1,234,567 and the value 0.1234567. However, if you add these two `floats` together, the result will be rounded to 1234567, because the exact result requires more precision than the seven significant digits that a `float` can hold. The error introduced by rounding off to seven digits can become large compared to the value computed, especially with repeated calculations. (See also the Advanced Topic, “Unexpected Inequality with Floating-Point Types,” later in this section.)

Internally, the binary floating-point types actually store a binary fraction, not a decimal fraction. Consequently, “representation error” inaccuracies can occur with a simple assignment, such as `double number = 140.6F`. The exact value of 140.6 is the fraction $\frac{703}{5}$, but the denominator of that fraction is not a power of 2, so it cannot be represented exactly by a binary floating-point number. The value actually represented is the closest fraction with a power of 2 in the denominator that will fit into the 16 bits of a `float`. Since the `double` can hold a more accurate value than the `float` can store, the C# compiler will actually evaluate this expression to `double number = 140.600006103516` because 140.600006103516 is the closest binary fraction to 140.6 as a `float`. This fraction is slightly larger than 140.6 when represented as a `double`.

**Guidelines**

**AVOID** binary floating-point types when exact decimal arithmetic is required; use the `decimal` floating-point type instead.

**ADVANCED TOPIC**

**Unexpected Inequality with Floating-Point Types**

Because floating-point numbers can be unexpectedly rounded off to non-decimal fractions, comparing floating-point values for equality can be quite confusing. Consider Listing 3.7.
Listing 3.7: Unexpected Inequality Due to Floating-Point Inaccuracies

```csharp
decimal decimalNumber = 4.2M;
double doubleNumber1 = 0.1F * 42F;
double doubleNumber2 = 0.1D * 42D;
float floatNumber = 0.1F * 42F;

Trace.Assert(decimalNumber != (decimal)doubleNumber1);
// 1. Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    "{decimalNumber} != {(decimal)doubleNumber1}"
);

Trace.Assert((double)decimalNumber != doubleNumber1);
// 2. Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    "{(double)decimalNumber} != {doubleNumber1}"
);

Trace.Assert((float)decimalNumber != floatNumber);
// 3. Displays: (float)4.2M != 4.2F
System.Console.WriteLine(
    "(float){(float)decimalNumber}M != {floatNumber}F"
);

Trace.Assert(doubleNumber1 != (double)floatNumber);
// 4. Displays: 4.20000006258488 != 4.2000028610229
System.Console.WriteLine(
    "{doubleNumber1} != {(double)floatNumber}"
);

Trace.Assert(doubleNumber1 != doubleNumber2);
// 5. Displays: 4.20000006258488 != 4.2
System.Console.WriteLine(
    "{doubleNumber1} != {doubleNumber2}"
);

Trace.Assert(floatNumber != doubleNumber2);
// 6. Displays: 4.2F != 4.2D
System.Console.WriteLine(
    "{floatNumber}F != {doubleNumber2}D"
);

Trace.Assert((double)4.2F != 4.2D);
// 7. Displays: 4.19999980926514 != 4.2
System.Console.WriteLine(
    "{(double)4.2F} != {4.2D}"
);

Trace.Assert(4.2F != 4.2D);
// 8. Displays: 4.2F != 4.2D
System.Console.WriteLine(
    "{4.2F}F != {4.2D}D"
);
```

Output 3.6 shows the results of Listing 3.7.
The `Assert()` methods alert the developer whenever arguments evaluate to `false`. However, of all the comparisons in this code listing, none of them are in fact equal. In spite of the apparent equality of the values in the code listing, they are not actually equivalent due to the inaccuracies associated with `float` values.

**Guidelines**

**AVOID** using equality conditionals with binary floating-point types. Either subtract the two values and see if their difference is less than a tolerance, or use the `decimal` type.

You should be aware of some additional unique floating-point characteristics as well. For instance, you would expect that dividing an integer by zero would result in an error—and it does with data types such as `int` and `decimal`. The `float` and `double` types, however, allow for certain special values. Consider Listing 3.8, and its resultant output, Output 3.7.

**Listing 3.8: Dividing a Float by Zero, Displaying NaN**

```csharp
float n=0f;
// Displays: NaN
System.Console.WriteLine(n / 0);
```

**Output 3.7**

`NaN`
In mathematics, certain mathematical operations are undefined, including dividing zero by itself. In C#, the result of dividing the float zero by zero results in a special “Not a Number” value; all attempts to print the output of such a number will result in NaN. Similarly, taking the square root of a negative number with System.Math.Sqrt(-1) will result in NaN.

A floating-point number could overflow its bounds as well. For example, the upper bound of the float type is approximately $3.4 \times 10^{38}$. Should the number overflow that bound, the result would be stored as “positive infinity” and the output of printing the number would be Infinity. Similarly, the lower bound of a float type is $-3.4 \times 10^{38}$, and computing a value below that bound would result in “negative infinity,” which would be represented by the string -Infinity. Listing 3.9 produces negative and positive infinity, respectively, and Output 3.8 shows the results.

**Listing 3.9: Overflowing the Bounds of a float**

```csharp
// Displays: -Infinity
System.Console.WriteLine(-1f / 0);
// Displays: Infinity
System.Console.WriteLine(3.402823E+38f * 2f);
```

**Output 3.8**

```
-Infinity
Infinity
```

Further examination of the floating-point number reveals that it can contain a value very close to zero, without actually containing zero. If the value exceeds the lower threshold for the float or double type, the value of the number can be represented as “negative zero” or “positive zero,” depending on whether the number is negative or positive, and is represented in output as -0 or 0.

**Compound Assignment Operators (+=, -=, *=, /=, %=)**

Chapter 1 discussed the simple assignment operator, which places the value of the right-hand side of the operator into the variable on the left-hand side. Compound assignment operators combine common binary operator calculations with the assignment operator. For example, consider Listing 3.10.
In this assignment, first you calculate the value of \( x + 2 \) and then you assign the calculated value back to \( x \). Since this type of operation is performed relatively frequently, an assignment operator exists to handle both the calculation and the assignment with one operator. The \( += \) operator increments the variable on the left-hand side of the operator with the value on the right-hand side of the operator, as shown in Listing 3.11.

This code, therefore, is equivalent to Listing 3.10.

Numerous other “compound assignment” operators exist to provide similar functionality. You can also use the assignment operator the with subtraction, multiplication, division, and remainder operators (as demonstrated in Listing 3.12).

C# includes special unary operators for incrementing and decrementing counters. The increment operator, \(++\), increments a variable by one each time it is used. In other words, all of the code lines shown in Listing 3.13 are equivalent.

---

**Listing 3.10: Common Increment Calculation**

```c
int x = 123;
x = x + 2;
```

**Listing 3.11: Using the += Operator**

```c
int x = 123;
x += 2;
```

**Listing 3.12: Other Assignment Operator Examples**

```c
x -= 2;
x /= 2;
x *= 2;
x %= 2;
```

**Increment and Decrement Operators (++, --)**

**Listing 3.13: Increment Operator**

```c
spaceCount = spaceCount + 1;
spaceCount += 1;
spaceCount++;
```
Similarly, you can decrement a variable by one using the \textbf{decrement operator}, \texttt{--}. Therefore, all of the code lines shown in Listing 3.14 are also equivalent.

\textbf{Listing 3.14: Decrement Operator}

\begin{verbatim}
lines = lines - 1;
lines -= 1;
lines--;  
\end{verbatim}

\textbf{BEGINNER TOPIC}

\textbf{A Decrement Example in a Loop}

The increment and decrement operators are especially prevalent in loops, such as the \texttt{while} loop described later in the chapter. For example, Listing 3.15 uses the decrement operator to iterate backward through each letter in the alphabet.

\textbf{Listing 3.15: Displaying Each Character's Unicode Value in Descending Order}

\begin{verbatim}
char current;
int unicodeValue;

// Set the initial value of current.
current = 'z';

do {
    // Retrieve the Unicode value of current.
    unicodeValue = current;
    System.Console.Write("\{current\}={unicodeValue}\t");

    // Proceed to the previous letter in the alphabet;
    current--;
}
while(current >= 'a');
\end{verbatim}

Output 3.9 shows the results of Listing 3.15.

\textbf{Output 3.9}

\begin{verbatim}
z=122  y=121  x=120  w=119  v=118  u=117  t=116  s=115  r=114  
q=113  p=112  o=111  n=110  m=109  l=108  k=107  j=106  i=105  
h=104  g=103  f=102  e=101  d=100  c=99  b=98  a=97
\end{verbatim}
The increment and decrement operators are used in Listing 3.15 to control how many times a particular operation is performed. In this example, notice that the increment operator is also used on a character (char) data type. You can use increment and decrement operators on various data types as long as some meaning is assigned to the concept of the “next” or “previous” value for that data type.

We saw that the assignment operator first computes the value to be assigned, and then performs the assignment. The result of the assignment operator is the value that was assigned. The increment and decrement operators are similar: They compute the value to be assigned, perform the assignment, and result in a value. It is therefore possible to use the assignment operator with the increment or decrement operator, though doing so carelessly can be extremely confusing. See Listing 3.16 and Output 3.10 for an example.

**Listing 3.16: Using the Post-Increment Operator**

```csharp
int count = 123;
int result;
result = count++;
System.Console.WriteLine($"result = {result} and count = {count}");
```

**Output 3.10**

```
result = 123 and count = 124
```

You might be surprised that result was assigned the value that was count before count was incremented. Where you place the increment or decrement operator determines whether the assigned value should be the value of the operand before or after the calculation. If you want the value of result to be the value assigned to count, you need to place the operator before the variable being incremented, as shown in Listing 3.17.

**Listing 3.17: Using the Pre-Increment Operator**

```csharp
int count = 123;
int result;
result = ++count;
System.Console.WriteLine($"result = {result} and count = {count}");
```
Chapter 3: Operators and Control Flow

Output 3.11 shows the results of Listing 3.17.

**Output 3.11**

```
result = 124 and count = 124
```

In this example, the increment operator appears before the operand, so the result of the expression is the value assigned to the variable after the increment. If count is 123, `++count` will assign 124 to count and produce the result 124. By contrast, the postfix increment operator `count++` assigns 124 to count and produces the value that count held before the increment: 123. Regardless of whether the operator is postfix or prefix, the variable count will be incremented before the value is produced; the only difference is which value is produced. The difference between prefix and postfix behavior is illustrated in Listing 3.18. The resultant output is shown in Output 3.12.

**Listing 3.18: Comparing the Prefix and Postfix Increment Operators**

```csharp
class IncrementExample
{
    static void Main()
    {
        int x = 123;
        // Displays 123, 124, 125.
        System.Console.WriteLine($"{x++}, {x++}, {x}");
        // x now contains the value 125.
        // Displays 126, 127, 127.
        System.Console.WriteLine($"{++x}, {++x}, {x}");
        // x now contains the value 127.
    }
}
```

**Output 3.12**

```
123, 124, 125
126, 127, 127
```

As Listing 3.18 demonstrates, where the increment and decrement operators appear relative to the operand can affect the result produced by the expression. The result of the prefix operators is the value that the variable had before it was incremented or decremented. The result of the postfix
operators is the value that the variable had after it was incremented or decremented. Use caution when embedding these operators in the middle of a statement. When in doubt as to what will happen, use these operators independently, placing them within their own statements. This way, the code is also more readable and there is no mistaking the intention.

Language Contrast: C++—Implementation-Defined Behavior

Earlier we discussed how the operands in an expression can be evaluated in any order in C++, whereas they are always evaluated from left to right in C#. Similarly, in C++ an implementation may legally perform the side effects of increments and decrements in any order. For example, in C++ a call of the form $M(x++, x++)$, where $x$ begins as 1, can legally call either $M(1,2)$ or $M(2,1)$ at the whim of the compiler. In contrast, C# will always call $M(1,2)$ because C# makes two guarantees: (1) The arguments to a call are always computed from left to right, and (2) the assignment of the incremented value to the variable always happens before the value of the expression is used. C++ makes neither guarantee.

Guidelines

**AVOID** confusing usage of the increment and decrement operators.  
**DO** be cautious when porting code between C, C++, and C# that uses increment and decrement operators; C and C++ implementations need not follow the same rules as C#.

Advanced Topic

Thread-Safe Incrementing and Decrementing

In spite of the brevity of the increment and decrement operators, these operators are not atomic. A thread context switch can occur during the execution of the operator and can cause a race condition. You could use a lock statement to prevent the race condition. However, for
simple increments and decrements, a less expensive alternative is to use the thread-safe `Increment()` and `Decrement()` methods from the `System.Threading.Interlocked` class. These methods rely on processor functions for performing fast thread-safe increments and decrements. See Chapter 19 for more details.

**Constant Expressions and Constant Locals**

The preceding chapter discussed literal values, or values embedded directly into the code. It is possible to combine multiple literal values in a constant expression using operators. By definition, a constant expression is one that the C# compiler can evaluate at compile time (instead of evaluating it when the program runs) because it is composed entirely of constant operands. Constant expressions can then be used to initialize constant locals, which allow you to give a name to a constant value (similar to the way local variables allow you to give a name to a storage location). For example, the computation of the number of seconds in a day can be a constant expression that is then used in other expressions by name.

The `const` keyword in Listing 3.19 declares a constant local. Since a constant local is by definition the opposite of a variable—“constant” means “not able to vary”—any attempt to modify the value later in the code would result in a compile-time error.

**Guidelines**

*DO NOT* use a constant for any value that can possibly change over time. The value of pi and the number of protons in an atom of gold are constants; the price of gold, the name of your company, and the version number of your program can change.

Note that the expression assigned to `secondsPerWeek` in Listing 3.19 is a constant expression because all the operands in the expression are also constants.
Introducing Flow Control

Later in this chapter is a code listing (Listing 3.45) that shows a simple way to view a number in its binary form. Even such a simple program, however, cannot be written without using control flow statements. Such statements control the execution path of the program. This section discusses how to change the order of statement execution based on conditional checks. Later on, you will learn how to execute statement groups repeatedly through loop constructs.

A summary of the control flow statements appears in Table 3.1. Note that the General Syntax Structure column indicates common statement use, not the complete lexical structure. An embedded-statement in Table 3.1 may be any statement other than a labeled statement or a declaration, but it is typically a block statement.

Each C# control flow statement in Table 3.1 appears in the tic-tac-toe³ program and is available in Appendix B and for download with the rest of the source code listings from the book. The program displays the tic-tac-toe board, prompts each player, and updates with each move.

The remainder of this chapter looks at each statement in more detail. After covering the if statement, it introduces code blocks, scope, Boolean expressions, and bitwise operators before continuing with the remaining control flow statements. Readers who find Table 3.1 familiar because of C#’s similarities to other languages can jump ahead to the section titled “C# Preprocessor Directives” or skip to the “Summary” section at the end of the chapter.

---

3. Known as noughts and crosses to readers outside the United States.
### Table 3.1: Control Flow Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>General Syntax Structure</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>if statement</td>
<td>if(boolean-expression)</td>
<td>if (input == &quot;quit&quot;)</td>
</tr>
<tr>
<td></td>
<td>embedded-statement</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System.Console.WriteLine(&quot;Game end&quot;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return;</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>embedded-statement</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System.Console.WriteLine(&quot;Game end&quot;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>return;</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>embedded-statement</td>
<td>GetNextMove();</td>
</tr>
<tr>
<td>while statement</td>
<td>while(boolean-expression)</td>
<td>while(count &lt; total)</td>
</tr>
<tr>
<td></td>
<td>embedded-statement</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System.Console.WriteLine(&quot;count = {count}&quot;));</td>
</tr>
<tr>
<td></td>
<td></td>
<td>count++;</td>
</tr>
<tr>
<td>do while statement</td>
<td>do embedded-statement</td>
<td>do</td>
</tr>
<tr>
<td></td>
<td>while(boolean-expression)</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System.Console.WriteLine(&quot;Enter name:&quot;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>input = System.Console.ReadLine();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>while(input != &quot;exit&quot;);</td>
</tr>
<tr>
<td>Statement</td>
<td>General Syntax Structure</td>
<td>Example</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| for statement    | for(for-initializer;  
                    boolean-expression;  
                    for-iterator)  
                    embedded-statement | for (int count = 1;  
                    count <= 10;  
                    count++)  
                    {  
                    System.Console.WriteLine("count = {count}"));  
                    } |
| foreach statement| foreach(type identifier in  
                    expression)  
                    embedded-statement | foreach (char letter in email)  
                    {  
                    if(!insideDomain)  
                    {  
                    if (letter == '@')  
                    {  
                    insideDomain = true;  
                    }  
                    continue;  
                    }  
                    System.Console.Write(letter);  
                    } |
| continue statement| continue;                                             | continues                                                                 |
### Table 3.1: Control Flow Statements, (continued)

<table>
<thead>
<tr>
<th>Statement</th>
<th>General Syntax Structure</th>
<th>Example</th>
</tr>
</thead>
</table>
| switch statement  | `switch(governing-type-expression)`

  ```
  ...
  case const-expression:
    statement-list
    jump-statement
  default:
    statement-list
    jump-statement
  ```

  ```
  switch(input)
  {
    case "exit":
    case "quit":
      System.Console.WriteLine(  
        "Exiting app....");
      break;
    case "restart":
      Reset();
      goto case "start";
    case "start":
      GetMove();
      break;
    default:
      System.Console.WriteLine(  
        input);
      break;
  }
  ```

| break statement   | `break;`                                                                                 |         |
| goto statement    | `goto identifier;`                                                                      |         |
|                   | `goto case const-expression;`                                                           |         |
|                   | `goto default;`                                                                          |         |
**If Statement**

The `if` statement is one of the most common statements in C#. It evaluates a **Boolean expression** (an expression that results in either true or false) called the **condition**. If the condition is true, the **consequence statement** is executed. An if statement may optionally have an else clause that contains an **alternative statement** to be executed if the condition is false. The general form is as follows:

```csharp
if (condition)
    consequence-statement
else
    alternative-statement
```

**Listing 3.20: if/else Statement Example**

```csharp
class TicTacToe // Declares the TicTacToe class.
{
    static void Main() // Declares the entry point of the program.
    {
        string input;

        // Prompt the user to select a 1- or 2-player game.
        System.Console.Write(
            "1 – Play against the computer\n" +
            "2 – Play against another player.\n" +
            "Choose:"
        );
        input = System.Console.ReadLine();

        if(input=="1")
            // The user selected to play the computer.
            System.Console.WriteLine(
                "Play against computer selected."
            );
        else
            // Default to 2 players (even if user didn't enter 2).
            System.Console.WriteLine(
                "Play against another player."
            );
    }
}
```

In Listing 3.20, if the user enters 1, the program displays "Play against computer selected." Otherwise, it displays "Play against another player."
Nested if

Sometimes code requires multiple if statements. The code in Listing 3.21 first determines whether the user has chosen to exit by entering a number less than or equal to 0; if not, it checks whether the user knows the maximum number of turns in tic-tac-toe.

**Listing 3.21: Nested if Statements**

```csharp
1. class TicTacToeTrivia
2. {
3.     static void Main()
4.     {
5.         int input; // Declare a variable to store the input.
6.          
7.         System.Console.Write(
8.             "What is the maximum number " +
9.             "of turns in tic-tac-toe?" +
10.             "(Enter 0 to exit.): ");
11.          
12.         // int.Parse() converts the ReadLine() // return to an int data type.
13.         input = int.Parse(System.Console.ReadLine());
14.          
15.         if (input <= 0) // line 16
16.             // Input is less than or equal to 0.
17.             System.Console.WriteLine("Exiting...");
18.         else
19.             if (input < 9) // line 20
20.                 // Input is less than 9.
21.                 System.Console.WriteLine("Tic-tac-toe has more than {input}" +
22.                     " maximum turns.");
23.             else
24.                 if(input > 9) // line 26
25.                     // Input is greater than 9.
26.                     System.Console.WriteLine("Tic-tac-toe has fewer than {input}" +
27.                         " maximum turns.");
28.                 else
30.                     System.Console.WriteLine( // line 33
31.                         "Correct, tic-tac-toe " +
32.                         "has a maximum of 9 turns.");
33.         }
34.     }
35. }
```

Output 3.13 shows the results of Listing 3.21.
Output 3.13

What is the maximum number of turns in tic-tac-toe? (Enter 0 to exit.): 9
Correct, tic-tac-toe has a maximum of 9 turns.

Assume the user enters 9 when prompted at line 14. Here is the execution path:

1. Line 16: Check if input is less than 0. Since it is not, jump to line 20.
2. Line 20: Check if input is less than 9. Since it is not, jump to line 26.
3. Line 26: Check if input is greater than 9. Since it is not, jump to line 33.
4. Line 33: Display that the answer was correct.

Listing 3.21 contains nested if statements. To clarify the nesting, the lines are indented. However, as you learned in Chapter 1, whitespace does not affect the execution path. If this code was written without the indenting and without the newlines, the execution would be the same. The code that appears in the nested if statement in Listing 3.22 is equivalent to Listing 3.21.

Listing 3.22: if/else Formatted Sequentially

```csharp
if (input < 0)
    System.Console.WriteLine("Exiting...");
else if (input < 9)
    System.Console.WriteLine("Tic-tac-toe has more than \{input\} maximum turns.");
else if (input < 9)
    System.Console.WriteLine("Tic-tac-toe has less than \{input\} maximum turns.");
else
    System.Console.WriteLine("Correct, tic-tac-toe has a maximum \{input\} of 9 turns.");
```

Although the latter format is more common, in each situation you should use the format that results in the clearest code.

Both of the if statement listings omit the braces. However, as discussed next, this is not in accordance with the guidelines, which advocate the use of code blocks except, perhaps, in the simplest of single-line scenarios.
Chapter 3: Operators and Control Flow

Code Blocks ({})

In the previous if statement examples, only one statement follows if and else: a single System.Console.WriteLine(), similar to Listing 3.23.

Listing 3.23: if Statement with No Code Block

```csharp
if (input < 9)
    System.Console.WriteLine("Exiting");
```

With curly braces, however, we can combine statements into a single statement called a block statement or code block, allowing the grouping of multiple statements into a single statement that is the consequence. Take, for example, the highlighted code block in the radius calculation in Listing 3.24.

Listing 3.24: if Statement Followed by a Code Block

```csharp
class CircleAreaCalculator
{
    static void Main()
    {
        double radius; // Declare a variable to store the radius.
        double area;   // Declare a variable to store the area.

        System.Console.Write("Enter the radius of the circle: ");

        // double.Parse converts the ReadLine() // return to a double.
        radius = double.Parse(System.Console.ReadLine());
        if (radius >= 0)
        {
            // Calculate the area of the circle.
            area = Math.PI * radius * radius;
            System.Console.WriteLine(
                "$\text{The area of the circle is: \{ area : 0.00 \}}\)\);"
        }
        else
        {
            System.Console.WriteLine(
                "$\{ radius \} \text{ is not a valid radius.}\);"
        }
    }
}
```

Output 3.14 shows the results of Listing 3.24.
In this example, the if statement checks whether the radius is positive. If so, the area of the circle is calculated and displayed; otherwise, an invalid radius message is displayed.

Notice that in this example, two statements follow the first if. However, these two statements appear within curly braces. The curly braces combine the statements into a code block, which is itself a single statement.

If you omit the curly braces that create a code block in Listing 3.24, only the statement immediately following the Boolean expression executes conditionally. Subsequent statements will execute regardless of the if statement’s Boolean expression. The invalid code is shown in Listing 3.25.

### Listing 3.25: Relying on Indentation, Resulting in Invalid Code

```csharp
if(radius >= 0)
    area = Math.PI * radius * radius;
System.Console.WriteLine($"The area of the circle is: { area:0.00}");
```

In C#, indentation is used solely to enhance the code readability. The compiler ignores it, so the previous code is semantically equivalent to Listing 3.26.

### Listing 3.26: Semantically Equivalent to Listing 3.25

```csharp
if(radius >= 0)
{
    area = Math.PI * radius * radius;
}
System.Console.WriteLine($"The area of the circle is: { area:0.00}");
```

Programmers should take great care to avoid subtle bugs such as this, perhaps even going so far as to always include a code block after a control flow statement, even if there is only one statement. A widely accepted coding guideline is to avoid omitting braces, except possibly for the simplest of single-line if statements.
Although unusual, it is possible to have a code block that is not lexically a direct part of a control flow statement. In other words, placing curly braces on their own (without a conditional or loop, for example) is legal syntax.

In Listing 3.25 and Listing 3.26, the value of pi was represented by the PI constant in the System.Math class. Instead of hardcoding a value, such as 3.14 for constants such as pi and Euler’s constant (e), code should use System.Math.PI and System.Math.E.

**Guidelines**

**AVOID** omitting braces, except for the simplest of single-line `if` statements.

### Code Blocks, Scopes, and Declaration Spaces

Code blocks are often referred to as “scopes,” but the two terms are not exactly interchangeable. The *scope* of a named thing is the region of source code in which it is legal to refer to the thing by its unqualified name. The scope of a local variable, for example, is exactly the text of the code block that encloses it, which explains why it is common to refer to code blocks as “scopes.”

Scopes are often confused with declaration spaces. A *declaration space* is a logical container of named things in which two things may not have the same name. A code block defines not only a scope, but also a local variable declaration space. It is illegal for two local variable declarations with the same name to appear in the same declaration space. Similarly, it is not possible to declare two methods with the signature of `Main()` within the same class. (This rule is relaxed somewhat for methods: Two methods may have the same name in a declaration space provided that they have different signatures. The signature of a method includes its name and the number and types of its parameters.) Within a block, a local variable can be mentioned by name and must be the unique thing that is declared with that name in the block. Outside the declaring block, there is no way to refer to a local variable by its name; the local variable is said to be “out of scope” outside the block.
In summary, a scope is used to determine what thing a name refers to; a declaration space determines when two things declared with the same name conflict with each other. In Listing 3.27, declaring the local variable message inside the block statement embedded in the if statement restricts its scope to the block statement only; the local variable is “out of scope” when its name is used later on in the method. To avoid an error, you must declare the variable outside the block.

**Listing 3.27: Variables Inaccessible outside Their Scope**

```csharp
class Program
{
    static void Main(string[] args)
    {
        int playerCount;
        System.Console.Write(
            "Enter the number of players (1 or 2):\n\n            playerCount = int.Parse(System.Console.ReadLine());
        if (playerCount != 1 && playerCount != 2)
            {
                string message = "You entered an invalid number of players.";
            }
        else
            {
                // ...
            }
        // Error: message is not in scope.
        System.Console.WriteLine(message);
    }
}
```

Output 3.15 shows the results of Listing 3.27.

**Output 3.15**

```plaintext
...
Program.cs(18,26): error CS0103: The name 'message' does not exist in the current context
```

The declaration space in which a local variable’s name must be unique encompasses all the child code blocks textually enclosed within the block that originally declared the local. The C# compiler prevents the name of a local variable declared immediately within a method code block (or as a
parameter) from being reused within a child code block. In Listing 3.27, because `args` and `playerCount` are declared within the method code block, they cannot be declared again anywhere within the method.

The name `message` refers to this local variable throughout the scope of the local variable—that is, the block immediately enclosing the declaration. Similarly, `playerCount` refers to the same variable throughout the block containing the declaration, including within both of the child blocks that are the consequence and the alternative of the `if` statement.

**Language Contrast: C++—Local Variable Scope**

In C++, a local variable declared in a block is in scope from the point of the declaration statement through the end of the block. Thus an attempt to refer to the local variable before its declaration will fail to find the local variable because that variable is not in scope. If there is another thing with that name “in scope,” the C++ language will resolve the name to that thing, which might not be what you intended. In C#, the rule is subtly different: A local variable is in scope throughout the entire block in which it is declared, but it is illegal to refer to the local variable before its declaration. That is, the attempt to find the local variable will succeed, and the usage will then be treated as an error. This is just one of C#'s many rules that attempt to prevent errors common in C++ programs.

**Boolean Expressions**

The parenthesized condition of the `if` statement is a **Boolean expression**. In Listing 3.28, the condition is highlighted.

**Listing 3.28: Boolean Expression**

```csharp
if (input < 9)
{
    // Input is less than 9.
    System.Console.WriteLine($"Tic-tac-toe has more than \{ input \} maximum turns.");
}
// ...
```
Boolean expressions appear within many control flow statements. Their key characteristic is that they always evaluate to true or false. For input < 9 to be allowed as a Boolean expression, it must result in a bool. The compiler disallows x = 42, for example, because this expression assigns x and results in the value that was assigned, instead of checking whether the value of the variable is 42.

Language Contrast: C++—Mistakenly Using = in Place of ==

C# eliminates a coding error commonly found in C and C++. In C++, Listing 3.29 is allowed.

**Listing 3.29: C++, But Not C#, Allows Assignment As a Condition**

```
if (input = 9)  // Allowed in C++, not in C#.
    System.Console.WriteLine(
        "Correct, tic-tac-toe has a maximum of 9 turns.");
```

Although at first glance this code appears to check whether input equals 9, Chapter 1 showed that = represents the assignment operator, not a check for equality. The return from the assignment operator is the value assigned to the variable—in this case, 9. However, 9 is an int, and as such it does not qualify as a Boolean expression and is not allowed by the C# compiler. The C and C++ languages treat integers that are nonzero as true, and integers that are zero as false. C#, by contrast, requires that the condition actually be of a Boolean type; integers are not allowed.

Relational and Equality Operators

Relational and equality operators determine whether a value is greater than, less than, or equal to another value. Table 3.2 lists all the relational and equality operators. All are binary operators.

The C# syntax for equality uses ==, just as many other programming languages do. For example, to determine whether input equals 9, you use input  ==  9. The equality operator uses two equal signs to distinguish it from the assignment operator, =. The exclamation point signifies NOT in C#, so to test for inequality you use the inequality operator, !=.
TABLE 3.2: Relational and Equality Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>input&lt;9;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>input&gt;9;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>input&lt;=9;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>input&gt;=9;</td>
</tr>
<tr>
<td>==</td>
<td>Equality operator</td>
<td>input==9;</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality operator</td>
<td>input!=9;</td>
</tr>
</tbody>
</table>

Relational and equality operators always produce a bool value, as shown in Listing 3.30.

LISTING 3.30: Assigning the Result of a Relational Operator to a bool Variable

```csharp
bool result = 70 > 7;
```

In the tic-tac-toe program (see Appendix B), you use the equality operator to determine whether a user has quit. The Boolean expression of Listing 3.31 includes an OR (||) logical operator, which the next section discusses in detail.

LISTING 3.31: Using the Equality Operator in a Boolean Expression

```csharp
if (input == "" || input == "quit")
{
    System.Console.WriteLine("Player {currentPlayer} quit!!");
    break;
}
```

**Logical Boolean Operators**

The **logical operators** have Boolean operands and produce a Boolean result. Logical operators allow you to combine multiple Boolean expressions to form more complex Boolean expressions. The logical operators are |, ||, &, &&, and ^, corresponding to OR, AND, and exclusive OR. The | and & versions of OR and AND are rarely used for Boolean logic, for reasons which we discuss in this section.
**OR Operator (||)**

In Listing 3.31, if the user enters quit or presses the Enter key without typing in a value, it is assumed that she wants to exit the program. To enable two ways for the user to resign, you can use the logical OR operator, `||`. The `||` operator evaluates Boolean expressions and results in a true value if either operand is true (see Listing 3.32).

**Listing 3.32: Using the OR Operator**

```csharp
if ((hourOfDay > 23) || (hourOfDay < 0))
    System.Console.WriteLine("The time you entered is invalid.");
```

It is not necessary to evaluate both sides of an OR expression, because if either operand is true, the result is known to be true regardless of the value of the other operand. Like all operators in C#, the left operand is evaluated before the right one, so if the left portion of the expression evaluates to true, the right portion is ignored. In the example in Listing 3.32, if `hourOfDay` has the value 33, then `(hourOfDay > 23)` will evaluate to true and the OR operator will ignore the second half of the expression, short-circuiting it. Short-circuiting an expression also occurs with the Boolean AND operator. (Note that the parentheses are not necessary here; the logical operators are of higher precedence than the relational operators. However, it is clearer to the novice reader to parenthesize the subexpressions for clarity.)

**AND Operator (&&)**

The Boolean AND operator, `&&`, evaluates to true only if both operands evaluate to true. If either operand is false, the result will be false. Listing 3.33 writes a message if the given variable is both greater than 10 and less than 24.\(^4\) Similarly to the OR operator, the AND operator will not always evaluate the right side of the expression. If the left operand is determined to be false, the overall result will be false regardless of the value of the right operand, so the runtime skips evaluating the right operand.

**Listing 3.33: Using the AND Operator**

```csharp
if ((10 < hourOfDay) && (hourOfDay < 24))
    System.Console.WriteLine("Hi-Ho, Hi-Ho, it's off to work we go.");
```

---

4. The typical hours that programmers work each day.
Chapter 3: Operators and Control Flow

**Exclusive OR Operator (\^)\**
The caret symbol, \(^\), is the “exclusive OR” (XOR) operator. When applied to two Boolean operands, the XOR operator returns true only if exactly one of the operands is true, as shown in Table 3.3.

<table>
<thead>
<tr>
<th>Left Operand</th>
<th>Right Operand</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Unlike the Boolean AND and Boolean OR operators, the Boolean XOR operator does not short-circuit: It always checks both operands, because the result cannot be determined unless the values of both operands are known. Note that the XOR operator is exactly the same as the Boolean inequality operator.

**Logical Negation Operator (!)\**
The logical negation operator, or NOT operator, !, inverts a bool value. This operator is a unary operator, meaning it requires only one operand. Listing 3.34 demonstrates how it works, and Output 3.16 shows the result.

**Listing 3.34: Using the Logical Negation Operator**
```csharp
bool valid = false;
bool result = !valid;
// Displays "result = True".
System.Console.WriteLine($"result = { result }");
```

**Output 3.16**
```
result = True
```

At the beginning of Listing 3.34, valid is set to false. You then use the negation operator on valid and assign the value to result.
**Conditional Operator (?:)**

In place of an if-else statement used to select one of two values, you can use the **conditional** operator. The conditional operator uses both a question mark and a colon; the general format is as follows:

```plaintext
condition ? consequence : alternative
```

The conditional operator is a “ternary” operator because it has three operands: condition, consequence, and alternative. (As it is the only ternary operator in C#, it is often called “the ternary operator,” but it is clearer to refer to it by its name than by the number of operands it takes.) Like the logical operators, the conditional operator uses a form of short-circuiting. If the condition evaluates to true, the conditional operator evaluates only consequence. If the conditional evaluates to false, it evaluates only alternative. The result of the operator is the evaluated expression.

Listing 3.35 illustrates the use of the conditional operator. The full listing of this program appears in Appendix B.

**Listing 3.35: Conditional Operator**

```csharp
class TicTacToe
{
    static string Main()
    {
        // Initially set the currentPlayer to Player 1
        int currentPlayer = 1;

        // ...

        for (int turn = 1; turn <= 10; turn++)
        {
            // ...

            // Switch players
            currentPlayer = (currentPlayer == 2) ? 1 : 2;
        }
    }
}
```

The program swaps the current player. To do so, it checks whether the current value is 2. This is the “conditional” portion of the conditional expression. If the result of the condition is true, the conditional operator results in the “consequence” value 1. Otherwise, it results in the “alternative”
value 2. Unlike an if statement, the result of the conditional operator must be assigned (or passed as a parameter); it cannot appear as an entire statement on its own.

Guidelines

CONSIDER using an if/else statement instead of an overly complicated conditional expression.

The C# language requires that the consequence and alternative expressions in a conditional operator be typed consistently, and that the consistent type be determined without examination of the surrounding context of the expression. For example, `f ? "abc" : 123` is not a legal conditional expression because the consequence and alternative are a string and a number, neither of which is convertible to the other. Even if you say `object result = f ? "abc" : 123;` the C# compiler will flag this expression as illegal because the type that is consistent with both expressions (that is, `object`) is found outside the conditional expression.

**Null-Coalescing Operator (??)**

The null-coalescing operator is a concise way to express “If this value is null, then use this other value.” It has the following form:

```
expression1 ?? expression2
```

The null-coalescing operator also uses a form of short-circuiting. If `expression1` is not null, its value is the result of the operation and the other expression is not evaluated. If `expression1` does evaluate to null, the value of `expression2` is the result of the operator. Unlike the conditional operator, the null-coalescing operator is a binary operator.

Listing 3.36 illustrates the use of the null-coalescing operator.

**Listing 3.36: Null-Coalescing Operator**

```csharp
string fileName = GetFileName();
// ...
string fullName = fileName ?? "default.txt";
// ...
```
In this listing, we use the null-coalescing operator to set `fullName` to "default.txt" if `fileName` is null. If `fileName` is not null, `fullName` is simply assigned the value of `fileName`.

The null-coalescing operator “chains” nicely. For example, an expression of the form `x ?? y ?? z` results in `x` if `x` is not null; otherwise, it results in `y` if `y` is not null; otherwise, it results in `z`. That is, it goes from left to right and picks out the first non-null expression, or uses the last expression if all the previous expressions were null.

The null-coalescing operator was added to C# in version 2.0, along with nullable value types. This operator works on both operands of nullable value types and reference types.

**Null-Conditional Operator (?.)**

Whenever you invoke a method on a value that is null, the runtime will throw a System.NullReferenceException, which almost always indicates an error in the programming logic. In recognition of the frequency of this pattern (that is, checking for null before invoking a member), C# 6.0 introduces the “?.” operator, known as the null-conditional operator:

**Listing 3.37: Null-Conditional Operator**

```csharp
class Program
{
    static void Main(string[] args)
    {
        if (args?.Length == 0)
        {
            System.Console.WriteLine(
                "ERROR: File missing. "
                + "Use:\n\tfind.exe file:<filename>");
        }
        else
        {
            if (args[0]?.ToLower().StartsWith("file:"))??false)
            {
                string fileName = args[0]?.Remove(0, 5);
                // ...
            }
        }
    }
}
```
The null-conditional operator checks whether the operand (the first `args` in Listing 3.37) is null prior to invoking the method or property (`Length` in the first example in this listing). The logically equivalent explicit code would be the following (although in the C# 6.0 syntax the value of `args` is evaluated only once):

\[(\text{args} \neq \text{null}) \ ? \ (\text{int?})\text{args}.\text{Length} \ : \ \text{null}\]

What makes the null-conditional operator especially convenient is that it can be chained. If, for example, you invoke `args[0].ToLower().StartsWith("file:"), both `ToLower()` and `StartsWith()` will be invoked only if `args[0]` is not null. When expressions are chained, if the first operand is null, the expression evaluation is short-circuited, and no further invocation within the expression call chain will occur.

Be careful, however, that you don’t unintentionally neglect additional null-conditional operators. Consider, for example, what would happen if (hypothetically, in this case) `args[0].ToLower()` could also return null. In this scenario, a `NullReferenceException` would occur upon invocation of `StartsWith()`. This doesn’t mean you must use a chain of null-conditional operators, but rather that you should be intentional about the logic. In this example, because `ToLower()` can never be null, no additional null-conditional operator is necessary.

An important thing to note about the null-conditional operator is that, when utilized with a member that returns a value type, it always returns a nullable version of that type. For example, `args?.\text{Length}` returns an `int?`, not simply an `int`. Similarly, `args[0].ToLower().StartsWith("file:")` returns a `bool?` (a `Nullable<bool>`). Also, because an if statement requires a `bool` data type, it is necessary to follow the `StartsWith()` expression with the null-coalescing operator (??).

Although perhaps a little peculiar (in comparison to other operator behavior), the return of a nullable value type is produced only at the end of the call chain. Consequently, calling the dot (".”) operator on `\text{Length}` allows invocation of only `\text{int}` (not `\text{int?}`) members. However, encapsulating `args?.\text{Length}` in parentheses—thereby forcing the `\text{int?}` result via parentheses operator precedence—will invoke the `\text{int?}` return and make the `\text{Nullable<T>}` specific members (`HasValue` and `Value`) available.

Null-conditional operators can also be used in combination with an index operator, as shown in Listing 3.38.
In this listing, the first and second elements of `args` are assigned to their respective variables only if `args` is not null. If it is, null will be assigned instead.

Unfortunately, this example is naïve, if not dangerous, because the null-conditional operator gives a false sense of security, implying that if `args` isn’t null, then the element must exist. Of course, this isn’t the case: The element may not exist even if `args` isn’t null. Also, because checking for the element count with `args?.Length` verifies that `args` isn’t null, you never really need to use the null-conditional operator when indexing the collection after checking the length.

In conclusion, you should avoid using the null-conditional operator in combination with the index operator if the index operator throws an `IndexOutOfRangeException` for nonexistent indexes. Doing so leads to a false sense of code validity.

## Advanced Topic

**Leveraging the Null-Conditional Operator with Delegates**

The null-conditional operator is a great feature on its own. However, using it in combination with a delegate invocation resolves a C# pain point that has existed since C# 1.0. Notice in the code near the top of the next page how the `PropertyChange` event handler is assigned to a local copy (`propertyChanged`) before we check the value for null and finally fire the event. This is the easiest thread-safe way to invoke events without running the risk that an event unsubscribe will occur between the time when the check for null occurs and the time when the event is fired. Unfortunately, this approach is non-intuitive, and frequently code
neglects to follow this pattern—with the result of throwing inconsistent `NullReferenceExceptions`. Fortunately, with the introduction of the null-conditional operator in C# 6.0, this issue has been resolved.

With C# 6.0, the check for a delegate value changes from

```csharp
PropertyChangedEventHandler propertyChanged = PropertyChanged;
if (propertyChanged != null)
{
    propertyChanged(this,
        new PropertyChangedEventArgs(nameof(Name)));
}
```

to simply

```csharp
PropertyChanged?.Invoke(propertyChanged(
    this, new PropertyChangedEventArgs(nameof(Name))));
```

Because an event is just a delegate, the pattern of invoking a delegate via the null-conditional operator and an `Invoke()` is always possible.

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

An additional set of operators that is common to virtually all programming languages is the set of operators for manipulating values in their binary formats: the bit operators.

**BEGINNER TOPIC**

**Bits and Bytes**

All values within a computer are represented in a binary format of 1s and 0s, called **binary digits (bits)**. Bits are grouped together in sets of eight, called **bytes**. In a byte, each successive bit corresponds to a value of 2 raised to a power, starting from $2^0$ on the right and moving to $2^7$ on the left, as shown in Figure 3.1.

```
0 0 0 0 0 0 0 0
2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
```

**Figure 3.1:** Corresponding Placeholder Values
In many scenarios, particularly when dealing with low-level or system services, information is retrieved as binary data. To manipulate these devices and services, you need to perform manipulations of binary data.

In Figure 3.2, each box corresponds to a value of 2 raised to the power shown. The value of the byte (8-bit number) is the sum of the powers of 2 of all of the eight bits that are set to 1.

| 0 | 0 | 0 | 0 | 1 | 1 | 1 |

\[ 7 = 4 + 2 + 1 \]

**Figure 3.2: Calculating the Value of an Unsigned Byte**

The binary translation just described is significantly different for signed numbers. Signed numbers (`long`, `short`, `int`) are represented using a “two’s complement” notation. This practice ensures that addition continues to work when adding a negative number to a positive number, as though both were positive operands. With this notation, negative numbers behave differently from positive numbers. Negative numbers are identified by a 1 in the leftmost location. If the leftmost location contains a 1, you add the locations with 0s rather than the locations with 1s. Each location corresponds to the negative power of 2 value. Furthermore, from the result, it is also necessary to subtract 1. This is demonstrated in Figure 3.3.

| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

\[ -7 = -4 - 2 + 0 - 1 \]

**Figure 3.3: Calculating the Value of a Signed Byte**

Therefore, 1111 1111 1111 1111 corresponds to −1, and 1111 1111 1111 1001 holds the value −7. The binary representation 1000 0000 0000 0000 corresponds to the lowest negative value that a 16-bit integer can hold.
Shift Operators (<<, >>, <<=, >>=)

Sometimes you want to shift the binary value of a number to the right or left. In executing a left shift, all bits in a number’s binary representation are shifted to the left by the number of locations specified by the operand on the right of the shift operator. Zeroes are then used to backfill the locations on the right side of the binary number. A right-shift operator does almost the same thing in the opposite direction. However, if the number is a negative value of a signed type, the values used to backfill the left side of the binary number are 1s and not 0s. The shift operators are >> and <<, known as the right-shift and left-shift operators, respectively. In addition, there are combined shift and assignment operators, <<= and >>=.

Consider the following example. Suppose you had the int value -7, which would have a binary representation of 1111 1111 1111 1111 1111 1111 1111 1001. In Listing 3.39, you right-shift the binary representation of the number –7 by two locations.

Listing 3.39: Using the Right-Shift Operator

```csharp
int x;
x = (-7 >> 2); // 11111111111111111111111111111001 becomes
              // 11111111111111111111111111111110
// Write out "x is -2."
System.Console.WriteLine($"x = { x }.");
```

Output 3.17 shows the results of Listing 3.39.

**Output 3.17**

```
x = -2.
```

Because of the right shift, the value of the bit in the rightmost location has “dropped off” the edge and the negative bit indicator on the left shifts by two locations to be replaced with 1s. The result is -2.

Although legend has it that x << 2 is faster than x * 4, you should not use bit-shift operators for multiplication or division. This difference might have held true for certain C compilers in the 1970s, but modern compilers and modern microprocessors are perfectly capable of optimizing arithmetic. Using shifting for multiplication or division is confusing and frequently leads to errors when code maintainers forget that the shift operators are lower precedence than the arithmetic operators.
Bitwise Operators (&, |, ^)

In some instances, you might need to perform logical operations, such as AND, OR, and XOR, on a bit-by-bit basis for two operands. You do this via the & |, and ^ operators, respectively.

BEGINNER TOPIC

Logical Operators Explained

If you have two numbers, as shown in Figure 3.4, the bitwise operations will compare the values of the locations beginning at the leftmost significant value and continuing right until the end. The value of “1” in a location is treated as “true,” and the value of “0” in a location is treated as “false.”

12: 0 0 0 0 1 1 0 0

7: 0 0 0 0 0 1 1 1

Figure 3.4: The Numbers 12 and 7 Represented in Binary

Therefore, the bitwise AND of the two values in Figure 3.4 would entail the bit-by-bit comparison of bits in the first operand (12) with the bits in the second operand (7), resulting in the binary value 000000100, which is 4. Alternatively, a bitwise OR of the two values would produce 00001111, the binary equivalent of 15. The XOR result would be 00001011, or decimal 11.

Listing 3.40 demonstrates the use of these bitwise operators. The results of Listing 3.40 appear in Output 3.18.

Listing 3.40: Using Bitwise Operators

```csharp
byte and, or, xor;
and = 12 & 7;  // and = 4
or = 12 | 7;   // or = 15
xor = 12 ^ 7;  // xor = 11
System.Console.WriteLine(
  "$and = { and } \nor = { or }\nxor = { xor }\n$";
```
In Listing 3.40, the value 7 is the mask; it is used to expose or eliminate specific bits within the first operand using the particular operator expression. Note that, unlike the AND (&&) operator, the & operator always evaluates both sides even if the left portion is false. Similarly, the | version of the OR operator is not “short-circuiting.” It always evaluates both operands even if the left operand is true. The bit versions of the AND and OR operators, therefore, are not short-circuiting.

To convert a number to its binary representation, you need to iterate across each bit in a number. Listing 3.41 is an example of a program that converts an integer to a string of its binary representation. The results of Listing 3.41 appear in Output 3.19.

**LISTING 3.41: Getting a String Representation of a Binary Display**

```csharp
class BinaryConverter
{
    static void Main()
    {
        const int size = 64;
        ulong value;
        char bit;

        System.Console.Write("Enter an integer: ");
        // Use long.Parse() to support negative numbers
        // Assumes unchecked assignment to ulong.
        value = (ulong)long.Parse(System.Console.ReadLine());

        // Set initial mask to 100...
        ulong mask = 1UL << size - 1;
        for (int count = 0; count < size; count++)
        {
            bit = ((mask & value) != 0) ? '1': '0';
            System.Console.Write(bit);
            // Shift mask one location over to the right
            mask >>= 1;
        }
        System.Console.WriteLine();
    }
}
```
Within each iteration of the for loop in Listing 3.41 (as discussed later in this chapter), we use the right-shift assignment operator to create a mask corresponding to each bit position in value. By using the & bit operator to mask a particular bit, we can determine whether the bit is set. If the mask test produces a nonzero result, we write 1 to the console; otherwise, we write 0. In this way, we create output describing the binary value of an unsigned long.

Note also that the parentheses in (mask & value) != 0 are necessary because inequality is higher precedence than the AND operator. Without the explicit parentheses, this expression would be equivalent to mask & (value != 0), which does not make any sense; the left side of the & is a ulong and the right side is a bool.

This particular example is provided for learning purposes only. There is actually a built-in CLR method, System.Convert.ToString(value, 2) that does such a conversion. In fact, the second argument specifies the base (for example, 2 for binary, 10 for decimal, or 16 for hexadecimal), allowing for more than just conversion to binary.

**Bitwise Compound Assignment Operators (&=, |=, ^=)**

Not surprisingly, you can combine these bitwise operators with assignment operators as follows: &=, |=, and ^=. As a result, you could take a variable, OR it with a number, and assign the result back to the original variable, which Listing 3.42 demonstrates.

**Listing 3.42: Using Logical Assignment Operators**

```csharp
byte and = 12, or = 12, xor = 12;
and &= 7; // and = 4
or |= 7; // or = 15
xor ^= 7; // xor = 11
System.Console.WriteLine($"and = { and } \nor = { or }\nxor = { xor }");
```

The results of Listing 3.42 appear in Output 3.20.
Combining a bitmap with a mask using something like `fields & mask` clears the bits in `fields` that are not set in the `mask`. The opposite, `fields & ~mask`, clears out the bits in `fields` that are set in `mask`.

**Bitwise Complement Operator (~)**

The bitwise complement operator takes the complement of each bit in the operand, where the operand can be an `int`, `uint`, `long`, or `ulong`. The expression `~1`, therefore, returns the value with binary notation `1111 1111 1111 1111 1111 1111 1111 1110`, and `~(1<<31)` returns the number with binary notation `0111 1111 1111 1111 1111 1111 1111 1111`.

**Control Flow Statements, Continued**

Now that we’ve described Boolean expressions in more detail, we can more clearly describe the control flow statements supported by C#. Many of these statements will be familiar to experienced programmers, so you can skim this section looking for details specific to C#. Note in particular the `foreach` loop, as this may be new to many programmers.

**The while and do/while Loops**

Thus far you have learned how to write programs that do something only once. However, computers can easily perform similar operations multiple times. To do this, you need to create an instruction loop. The first instruction loop we will discuss is the `while` loop, because it is the simplest conditional loop. The general form of the `while` statement is as follows:

```csharp
while (condition)
    statement
```

The computer will repeatedly execute the statement that is the “body” of the loop as long as the condition (which must be a Boolean expression) evaluates to `true`. If the condition evaluates to `false`, code execution skips...
the body and executes the code following the loop statement. Note that statement will continue to execute even if it causes the condition to become false. The loop exits only when the condition is reevaluated “at the top of the loop.” The Fibonacci calculator shown in Listing 3.43 demonstrates the while loop.

**Listing 3.43: while Loop Example**

```csharp
class FibonacciCalculator
{
    static void Main()
    {
        decimal current;
        decimal previous;
        decimal temp;
        decimal input;

        System.Console.Write("Enter a positive integer:");

        // decimal.Parse convert the ReadLine to a decimal.
        input = decimal.Parse(System.Console.ReadLine());

        // Initialize current and previous to 1, the first // two numbers in the Fibonacci series.
        current = previous = 1;

        // While the current Fibonacci number in the series is // less than the value input by the user.
        while (current <= input)
        {
            temp = current;
            current = previous + current;
            previous = temp; // Executes even if previous // statement caused current to exceed input
        }

        System.Console.WriteLine(
            "$\text{The Fibonacci number following this is } \{ \text{current} \}\$");
    }
}
```

A Fibonacci number is a member of the Fibonacci series, which includes all numbers that are the sum of the previous two numbers in the series, beginning with 1 and 1. In Listing 3.43, you prompt the user for an integer. Then you use a while loop to find the first Fibonacci number that is greater than the number the user entered.
BEGINNER TOPIC

When to Use a while Loop

The remainder of this chapter considers other statements that cause a block of code to execute repeatedly. The term loop body refers to the statement (frequently a code block) that is to be executed within the while statement, since the code is executed in a “loop” until the exit condition is achieved. It is important to understand which loop construct to select. You use a while construct to iterate while the condition evaluates to true. A for loop is used most appropriately whenever the number of repetitions is known, such as when counting from 0 to n. A do/while is similar to a while loop, except that it will always execute the loop body at least once.

The do/while loop is very similar to the while loop except that a do/while loop is preferred when the number of repetitions is from 1 to n and n is not known when iterating begins. This pattern frequently occurs when prompting a user for input. Listing 3.44 is taken from the tic-tac-toe program.

Listing 3.44: do/while Loop Example

```csharp
    // Repeatedly request player to move until he
    // enters a valid position on the board.
    bool valid;
    do
    {
        valid = false;

        // Request a move from the current player.
        System.Console.Write("
        Player {currentPlayer}: Enter move:";
        input = System.Console.ReadLine();

        // Check the current player's input.
        // ...
    } while (!valid);
```

In Listing 3.44, you initialize valid to false at the beginning of each iteration, or loop repetition. Next, you prompt and retrieve the number the user input. Although not shown here, you then check whether the input was correct, and if it was, you assign valid equal to true. Since the code uses a do/while statement rather than a while statement, the user will be prompted for input at least once.

The general form of the do/while loop is as follows:
do
  statement
while (condition);

As with all the control flow statements, a code block is generally used as the single statement to allow multiple statements to be executed as the loop body. However, any single statement except for a labeled statement or a local variable declaration can be used.

**The for Loop**
The for loop iterates a code block until a specified condition is reached. In that way, it is very similar to the while loop. The difference is that the for loop has built-in syntax for initializing, incrementing, and testing the value of a counter, known as the **loop variable**. Because there is a specific location in the loop syntax for an increment operation, the increment and decrement operators are frequently used as part of a for loop.

Listing 3.45 shows the for loop used to display an integer in binary form (functionality the equivalent calling the BCL static function `System.Convert.ToString()` with a `toBase` value of 2). The results of this listing appear in Output 3.21.

**LISTING 3.45: Using the for Loop**
```csharp
class BinaryConverter
{
    static void Main()
    {
        const int size = 64;
        ulong value;
        char bit;

        System.Console.Write("Enter an integer: ");
        // Use long.Parse() so as to support negative numbers.
        // Assumes unchecked assignment to ulong.
        value = (ulong)long.Parse(System.Console.ReadLine());

        // Set initial mask to 100....
        ulong mask = 1UL << size - 1;
        for (int count = 0; count < size; count++)
        {
            bit = ((mask & value) > 0) ? '1': '0';
            System.Console.Write(bit);
            // Shift mask one location over to the right
            mask >>= 1;
        }
    }
}
```
Listing 3.45 performs a bit mask 64 times, once for each bit in the number. The three parts of the for loop header first declare and initialize the variable count, then describe the condition that must be met for the loop body to be executed, and finally describe the operation that updates the loop variable. The general form of the for loop is as follows:

```
for (initial ; condition ; loop)
  statement
```

Here is a breakdown of the for loop.

- The initial section performs operations that precede the first iteration. In Listing 3.45, it declares and initializes the variable count. The initial expression does not have to be a declaration of a new variable (though it frequently is). It is possible, for example, to declare the variable beforehand and simply initialize it in the for loop, or to skip the initialization section entirely by leaving it blank. Variables declared here are in scope throughout the header and body of the for statement.

- The condition portion of the for loop specifies an end condition. The loop exits when this condition is false exactly like the while loop does. The for loop will execute the body only as long as the condition evaluates to true. In the preceding example, the loop exits when count is greater than or equal to 64.

- The loop expression executes after each iteration. In the preceding example, count++ executes after the right shift of the mask (mask >>= 1), but before the condition is evaluated. During the sixty-fourth iteration, count is incremented to 64, causing the condition to become false, and therefore terminating the loop.

- The statement portion of the for loop is the “loop body” code that executes while the conditional expression remains true.

If you wrote out each for loop execution step in pseudocode without using a for loop expression, it would look like this:
1. Declare and initialize count to 0.
2. If count is less than 64, continue to step 3; otherwise, go to step 7.
3. Calculate bit and display it.
4. Shift the mask.
5. Increment count by 1.
6. Jump back to line 2.
7. Continue the execution of the program after the loop.

The for statement doesn’t require any of the elements in its header. The expression for(;;){ ... } is perfectly valid; although there still needs to be a means to escape from the loop so that it will not continue to execute indefinitely. (If the condition is missing, it is assumed to be the constant true.)

The initial and loop expressions have an unusual syntax to support loops that require multiple loop variables, as shown in Listing 3.46.

**Listing 3.46: for Loop Using Multiple Expressions**

```csharp
for (int x = 0, y = 5; ((x <= 5) && (y >=0 )); y--, x++)
{
    System.Console.Write(
        $\{ x \}{ ((x > y) ? '>' : '<' )}{ y }\t"
    );
}
```

The results of Listing 3.46 appear in Output 3.22.

**Output 3.22**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0&lt;5</td>
<td>1&lt;4</td>
<td>2&lt;3</td>
<td>3&gt;2</td>
<td>4&gt;1</td>
<td>5&gt;0</td>
</tr>
</tbody>
</table>

Here the initialization clause contains a complex declaration that declares and initializes two loop variables, but this is at least similar to a declaration statement that declares multiple local variables. The loop clause is quite unusual, as it can consist of a comma-separated list of expressions, not just a single expression.

**Guidelines**

**CONSIDER** refactoring the method to make the control flow easier to understand if you find yourself writing for loops with complex conditionals and multiple loop variables.
The for loop is little more than a more convenient way to write a while loop; you can always rewrite a for loop like this:

```csharp
{  
    initial;
    while (condition)  
    {
        statement;
        Loop;
    }
}
```

**Guidelines**

**DO** use the for loop when the number of loop iterations is known in advance and the “counter” that gives the number of iterations executed is needed in the loop.

**DO** use the while loop when the number of loop iterations is not known in advance and a counter is not needed.

**The foreach Loop**

The last loop statement in the C# language is foreach. The foreach loop iterates through a collection of items, setting a loop variable to represent each item in turn. In the body of the loop, operations may be performed on the item. A nice property of the foreach loop is that every item is iterated over exactly once; it is not possible to accidentally miscount and iterate past the end of the collection, as can happen with other loops.

The general form of the foreach statement is as follows:

```
foreach(type variable in collection)
    statement
```

Here is a breakdown of the foreach statement:

- **type** is used to declare the data type of the variable for each item within the collection. It may be `var`, in which case the compiler infers the type of the item from the type of the collection.
- **variable** is a read-only variable into which the foreach loop will automatically assign the next item within the collection. The scope of the variable is limited to the body of the loop.
• collection is an expression, such as an array, representing any number of items.
• statement is the loop body that executes for each iteration of the loop.

Consider the foreach loop in the context of the simple example shown in Listing 3.47.

**Listing 3.47: Determining Remaining Moves Using the foreach Loop**

class TicTacToe // Declares the TicTacToe class.
{
    static void Main() // Declares the entry point of the program.
    {
        // Hardcode initial board as follows
        // ---+---+---
        //  1 | 2 | 3
        // ---+---+---
        //  4 | 5 | 6
        // ---+---+---
        //  7 | 8 | 9
        // ---+---+---
        char[] cells = {
            '1', '2', '3', '4', '5', '6', '7', '8', '9'
        };
        System.Console.Write(
            "The available moves are as follows: ");
        // Write out the initial available moves
        foreach (char cell in cells)
        {
            if (cell != 'O' && cell != 'X')
            {
                System.Console.Write("{ cell } ");
            }
        }
    }
}

Output 3.23 shows the results of Listing 3.47.

**Output 3.23**
The available moves are as follows: 1 2 3 4 5 6 7 8 9

When the execution engine reaches the foreach statement, it assigns to the variable cell the first item in the cells array—in this case, the value
'1'. It then executes the code within the block that makes up the foreach loop body. The if statement determines whether the value of cell is '0' or 'X'. If it is neither, the value of cell is written out to the console. The next iteration then assigns the next array value to cell, and so on.

Note that the compiler prevents modification of the variable (cell) during the execution of a foreach loop. Also, the loop variable has a subtly different behavior in C# 5 and higher than it did in previous versions; the difference is apparent only when the loop body contains a lambda expression or anonymous method that uses the loop variable. See Chapter 12 for details.

**BEGINNER TOPIC**

Where the switch Statement Is More Appropriate

Sometimes you might compare the same value in several continuous if statements, as shown with the input variable in Listing 3.48.

**Listing 3.48: Checking the Player's Input with an if Statement**

```csharp
// ...
bool valid = false;

// Check the current player's input.
if( (input == "1") ||
    (input == "2") ||
    (input == "3") ||
    (input == "4") ||
    (input == "5") ||
    (input == "6") ||
    (input == "7") ||
    (input == "8") ||
    (input == "9") )
{
    // Save/move as the player directed.
    // ...

    valid = true;
}
else if( (input == "") || (input == "quit") )
{
    valid = true;
}
else
{
    System.Console.WriteLine(
```
This code validates the text entered to ensure that it is a valid tic-tac-toe move. If the value of `input` were 9, for example, the program would have to perform nine different evaluations. It would be preferable to jump to the correct code after only one evaluation. To enable this, you use a `switch` statement.

**The switch Statement**

A `switch` statement is simpler to understand than a complex `if` statement when you have a value that must be compared against many different constant values. The `switch` statement looks like this:

```csharp
switch (expression)
{
    case constant:
        statements
    default:
        statements
}
```

Here is a breakdown of the `switch` statement:

- `expression` is the value that is being compared against the different constants. The type of this expression determines the “governing type” of the switch. Allowable governing data types are `bool`, `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, any `enum` type (covered in Chapter 8), the corresponding nullable types of each of those value types, and `string`.
- `constant` is any constant expression compatible with the governing type.
- A group of one or more case labels (or the default label) followed by a group of one or more statements is called a `switch section`. The pattern given previously has two switch sections; Listing 3.49 shows a `switch` statement with three switch sections.
• statements is one or more statements to be executed when the expression equals one of the constant values mentioned in a label in the switch section. The end point of the group of statements must not be reachable. Typically the last statement is a jump statement such as a break, return, or goto statement.

Guidelines

DO NOT use continue as the jump statement that exits a switch section. This is legal when the switch is inside a loop, but it is easy to become confused about the meaning of break in a later switch section.

A switch statement should have at least one switch section; switch(x){} is legal but will generate a warning. Also, the guideline provided earlier (see page 116) was to avoid omitting braces in general. One exception to this rule of thumb is to omit braces for case and break statements because these keywords serve to indicate the beginning and end of a block, so no braces are needed.

Listing 3.49, with a switch statement, is semantically equivalent to the series of if statements in Listing 3.48.

**Listing 3.49: Replacing the if Statement with a switch Statement**

```csharp
static bool ValidateAndMove(
    int[] playerPositions, int currentPlayer, string input)
{
    bool valid = false;

    // Check the current player’s input.
    switch (input)
    {
    case "1":
    case "2":
    case "3":
    case "4":
    case "5":
    case "6":
    case "7":
    case "8":
    case "9":
        // Save/move as the player directed.
        ...
        valid = true;
        break;
    ```
case "":
case "quit":
    valid = true;
break;
default:
    // If none of the other case statements
    // is encountered then the text is invalid.
    System.Console.WriteLine(
        "\nERROR: Enter a value from 1-9."
        + "Push ENTER to quit");
break;
}

return valid;
}

In Listing 3.49, input is the test expression. Since input is a string, the
governing type is string. If the value of input is one of the strings 1, 2, 3,
4, 5, 6, 7, 8, or 9, the move is valid and you change the appropriate cell to
match that of the current user’s token (X or O). Once execution encounters
a break statement, control leaves the switch statement.

The next switch section describes how to handle the empty string or the
string quit; it sets valid to true if input equals either value. The default
switch section is executed if no other switch section had a case label that
matched the test expression.

Language Contrast: C++—switch Statement Fall-Through

In C++, if a switch section does not end with a jump statement, control
“falls through” to the next switch section, executing its code. Because un-
intended fall-through is a common error in C++, C# does not allow control
to accidentally fall through from one switch section to the next. The C#
designers believed it was better to prevent this common source of bugs and
encourage better code readability than to match the potentially confusing
C++ behavior. If you do want one switch section to execute the statements
of another switch section, you may do so explicitly with a goto statement,
as demonstrated later in this chapter.

There are several things to note about the switch statement:
A switch statement with no switch sections will generate a compiler warning, but the statement will still compile.

Switch sections can appear in any order; the default section does not have to appear last. In fact, the default switch section does not have to appear at all—it is optional.

The C# language requires that the end point of every switch section, including the last section, be unreachable. This means that switch sections usually end with a break, return, throw, or goto.

Jump Statements

It is possible to alter the execution path of a loop. In fact, with jump statements, it is possible to escape out of the loop or to skip the remaining portion of an iteration and begin with the next iteration, even when the loop condition remains true. This section considers some of the ways to jump the execution path from one location to another.

The break Statement

To escape out of a loop or a switch statement, C# uses a break statement. Whenever the break statement is encountered, control immediately leaves the loop or switch. Listing 3.50 examines the foreach loop from the tic-tac-toe program.

Listing 3.50: Using break to Escape Once a Winner Is Found

class TicTacToe // Declares the TicTacToe class.
{
    static void Main() // Declares the entry point of the program.
    {
        int winner = 0;
        // Stores locations each player has moved.
        int[] playerPositions = { 0, 0 };

        // Hardcoded board position.
        // X | 2 | 0
        // ------+
        // 0 | 0 | 6
        // ------+
        // X | X | X
        playerPositions[0] = 449;
        playerPositions[1] = 28;
// Determine if there is a winner.
int[] winningMasks = {
  7, 56, 448, 73, 146, 292, 84, 273
};

// Iterate through each winning mask to determine
// if there is a winner.
foreach (int mask in winningMasks)
{
  if ((mask & playerPositions[0]) == mask)
  {
    winner = 1;
    break;
  }
  else if ((mask & playerPositions[1]) == mask)
  {
    winner = 2;
    break;
  }
}
System.Console.WriteLine($"Player { winner } was the winner");

Output 3.24 shows the results of Listing 3.50.

Output 3.24
Player 1 was the winner

Listing 3.50 uses a break statement when a player holds a winning position. The break statement forces its enclosing loop (or a switch statement) to cease execution, and control moves to the next line outside the loop. For this listing, if the bit comparison returns true (if the board holds a winning position), the break statement causes control to jump and display the winner.

BEGINNER TOPIC

Bitwise Operators for Positions
The tic-tac-toe example (the full listing is available in Appendix B) uses the bitwise operators to determine which player wins the game. First, the code
saves the positions of each player into a bitmap called playerPositions. (It uses an array so that the positions for both players can be saved.)

To begin, both playerPositions are 0. As each player moves, the bit corresponding to the move is set. If, for example, the player selects cell 3, shifter is set to 3 - 1. The code subtracts 1 because C# is zero based and you need to adjust for 0 as the first position instead of 1. Next, the code sets position, the bit corresponding to cell 3, using the shift operator \(0000000000000001 \ll \text{shifter}\), where shifter now has a value of 2. Lastly, it sets playerPositions for the current player (subtracting 1 again to shift to zero based) to \(0000000000000000\). Listing 3.51 uses |\= so that previous moves are combined with the current move.

**Listing 3.51: Setting the Bit That Corresponds to Each Player’s Move**

```csharp
int shifter; // The number of places to shift
            // over to set a bit.
int position; // The bit that is to be set.

// int.Parse() converts "input" to an integer.
// "int.Parse(input) - 1" because arrays
// are zero based.
shifter = int.Parse(input) - 1;

// Shift mask of 00000000000000000000000000000001
// over by cellLocations.
position = 1 << shifter;

// Take the current player cells and OR them to set the
// new position as well.
// Since currentPlayer is either 1 or 2,
// subtract 1 to use currentPlayer as an
// index in a zero based array.
playerPositions[currentPlayer - 1] |= position;
```

Later in the program, you can iterate over each mask corresponding to winning positions on the board to determine whether the current player has a winning position, as shown in Listing 3.50.

**The continue Statement**

You might have a block containing a series of statements within a loop. If you determine that some conditions warrant executing only a portion of these statements for some iterations, you can use the continue statement to jump to the end of the current iteration and begin the next iteration. The
continue statement exits the current iteration (regardless of whether additional statements remain) and jumps to the loop condition. At that point, if the loop conditional is still true, the loop will continue execution.

Listing 3.52 uses the continue statement so that only the letters of the domain portion of an email are displayed. Output 3.25 shows the results of Listing 3.52.

**Listing 3.52: Determining the Domain of an Email Address**

```csharp
class EmailDomain
{
    static void Main()
    {
        string email;
        bool insideDomain = false;
        System.Console.WriteLine("Enter an email address: ");
        email = System.Console.ReadLine();
        System.Console.WriteLine("The email domain is: ");

        // Iterate through each letter in the email address.
        foreach (char letter in email)
        {
            if (!insideDomain)
            {
                if (letter == '@')
                {
                    insideDomain = true;
                }
                continue;
            }
        
        System.Console.Write(letter);
        }
    }
}
```

**Output 3.25**

```
Enter an email address:
mark@dotnetprogramming.com
The email domain is: dotnetprogramming.com
```

In Listing 3.52, if you are not yet inside the domain portion of the email address, you can use a continue statement to move control to the end of the loop, and process the next character in the email address.
You can almost always use an `if` statement in place of a `continue` statement, and this is usually more readable. The problem with the `continue` statement is that it provides multiple flows of control within a single iteration, which compromises readability. In Listing 3.53, the sample has been rewritten, replacing the `continue` statement with the `if/else` construct to demonstrate a more readable version that does not use the `continue` statement.

**Listing 3.53: Replacing a continue Statement with an if Statement**

```csharp
foreach (char letter in email)
{
    if (insideDomain)
    {
        System.Console.Write(letter);
    }
    else
    {
        if (letter == '@')
        {
            insideDomain = true;
        }
    }
}
```

**The goto Statement**

Early programming languages lacked the relatively sophisticated “structured” control flows that modern languages such as C# have as a matter of course, and instead relied upon simple conditional branching (`if`) and unconditional branching (`goto`) statements for most of their control flow needs. The resultant programs were often hard to understand. The continued existence of a `goto` statement within C# seems like an anachronism to many experienced programmers. However, C# supports `goto`, and it is the only method for supporting fall-through within a `switch` statement. In Listing 3.54, if the `/out` option is set, code execution jumps to the default case using the `goto` statement, and similarly for `/f`.

**Listing 3.54: Demonstrating a switch with goto Statements**

```csharp
// ...
static void Main(string[] args)
{
    bool isOutputSet = false;
```
bool isFiltered = false;

foreach (string option in args)
{
    switch (option)
    {
    case "/out":
        isOutputSet = true;
        isFiltered = false;
        goto default;
    case "/f":
        isFiltered = true;
        isRecursive = false;
        goto default;
    default:
        if (isRecursive)
        {
            // Recurse down the hierarchy
            // ...
        }
        else if (isFiltered)
        {
            // Add option to list of filters.
            // ...
        }
        break;
    }
}
// ...
// ...

Output 3.26 shows how to execute the code shown in Listing 3.54.

**Output 3.26**

C:\SAMPLES>Generate /out fizbottle.bin /f ".xml" ".wsdl"

To branch to a switch section label other than the default label, you can use the syntax `goto case constant;`, where constant is the constant associated with the case label you wish to branch to. To branch to a statement that is not associated with a switch section, precede the target statement with any identifier followed by a colon; you can then use that identifier with the `goto` statement. For example, you could have a labeled statement `myLabel : Console.WriteLine();`. The statement `goto myLabel;` would
then branch to the labeled statement. Fortunately, C# prevents you from using \texttt{goto} to branch \textit{into} a code block; instead, \texttt{goto} may be used only to branch within a code block or to an enclosing code block. By making these restrictions, C# avoids most of the serious \texttt{goto} abuses possible in other languages.

In spite of the improvements, use of \texttt{goto} is generally considered to be inelegant, difficult to understand, and symptomatic of poorly structured code. If you need to execute a section of code multiple times or under different circumstances, either use a loop or extract code to a method of its own.

\begin{center}
\textbf{Guidelines}
\end{center}

\textit{AVOID} using \texttt{goto}.

\section*{C# Preprocessor Directives}

Control flow statements evaluate expressions at runtime. In contrast, the C# preprocessor is invoked during compilation. The preprocessor commands are directives to the C# compiler, specifying the sections of code to compile or identifying how to handle specific errors and warnings within the code. C# preprocessor commands can also provide directives to C# editors regarding the organization of code.

\section*{Language Contrast: C++—Preprocessing}

Languages such as C and C++ use a \texttt{preprocessor} to perform actions on the code based on special tokens. Preprocessor directives generally tell the compiler how to compile the code in a file and do not participate in the compilation process itself. In contrast, the C# compiler handles “preprocessor” directives as part of the regular lexical analysis of the source code. As a result, C# does not support preprocessor macros beyond defining a constant. In fact, the term preprocessor is generally a misnomer for C#.
Each preprocessor directive begins with a hash symbol (#), and all preprocessor directives must appear on one line. A newline rather than a semicolon indicates the end of the directive.

A list of each preprocessor directive appears in Table 3.4.

<table>
<thead>
<tr>
<th>Table 3.4: Preprocessor Directives</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statement or Expression</strong></td>
</tr>
</tbody>
</table>
| #if directive | #if preprocessor-expression code endif | #if CSHARP2PLUS
                   Console.Clear();
                   #endif |
| #elif directive | #if preprocessor-expression1 code
                   #elif preprocessor-expression2 code
                   #endif | #if LINUX
                   ...
                   #elif WINDOWS
                   ...
                   #endif |
| #else directive | #if code
                   #else code
                   #endif | #if CSHARP1
                   ...
                   #else
                   ...
                   #endif |
| #define directive | #define conditional-symbol | #define CSHARP2PLUS |
| #undef directive | #undef conditional-symbol | #undef CSHARP2PLUS |
| #error directive | #error preproc-message | #error Buggy implementation |
| #warning directive | #warning preproc-message | #warning Needs code review |
| #pragma directive | #pragma warning | #pragma warning disable 1030 |
| #line directive | #line org-line new-line
                  #line default | #line 467 "TicTacToe.cs"
                   #line default |
| #region directive | #region pre-proc-message code
                   #endregion | #region Methods...
                   #endregion |
Excluding and Including Code (#if, #elif, #else, #endif)

Perhaps the most common use of preprocessor directives is in controlling when and how code is included. For example, to write code that could be compiled by both C# 2.0 and later compilers and the prior version 1.0 compilers, you would use a preprocessor directive to exclude C# 2.0–specific code when compiling with a version 1.0 compiler. You can see this in the tic-tac-toe example and in Listing 3.55.

Listing 3.55: Excluding C# 2.0 Code from a C# 1.x Compiler

```csharp
#if CSHARP2PLUS
System.Console.Clear();
#endif
```

In this case, you call the `System.Console.Clear()` method, which is available only in CLI 2.0 and later versions. Using the #if and #endif preprocessor directives, this line of code will be compiled only if the preprocessor symbol `CSHARP2PLUS` is defined.

Another use of the preprocessor directive would be to handle differences among platforms, such as surrounding Windows- and Linux-specific APIs with `WINDOWS` and `LINUX` #if directives. Developers often use these directives in place of multiline comments (/*...*/ because they are easier to remove by defining the appropriate symbol or via a search and replace.

A final common use of the directives is for debugging. If you surround code with an `#if DEBUG`, you will remove the code from a release build on most IDEs. The IDEs define the `DEBUG` symbol by default in a debug compile and `RELEASE` by default for release builds.

To handle an else-if condition, you can use the #elif directive within the #if directive, instead of creating two entirely separate #if blocks, as shown in Listing 3.56.

Listing 3.56: Using #if, #elif, and #endif Directives

```csharp
#if LINUX
...
#elif WINDOWS
...
#endif
```
Defining Preprocessor Symbols (#define, #undef)

You can define a preprocessor symbol in two ways. The first is with the #define directive, as shown in Listing 3.57.

Listing 3.57: A #define Example

```csharp
#define CSHARP2PLUS
```

The second method uses the define option when compiling for .NET, as shown in Output 3.27.

Output 3.27

```cmd
>csc.exe /define:CSHARP2PLUS TicTacToe.cs
```

Output 3.28 shows the same functionality using the Mono compiler.

Output 3.28

```cmd
>mcs.exe -define:CSHARP2PLUS TicTacToe.cs
```

To add multiple definitions, separate them with a semicolon. The advantage of the define compiler option is that no source code changes are required, so you may use the same source files to produce two different binaries.

To undefine a symbol, you use the #undef directive in the same way you use #define.

Emitting Errors and Warnings (#error, #warning)

Sometimes you may want to flag a potential problem with your code. You do this by inserting #error and #warning directives to emit an error or a warning, respectively. Listing 3.58 uses the tic-tac-toe sample to warn that the code does not yet prevent players from entering the same move multiple times. The results of Listing 3.58 appear in Output 3.29.

Listing 3.58: Defining a Warning with #warning

```csharp
#warning "Same move allowed multiple times."
```
Output 3.29

Performing main compilation...
...\tictactoe.cs(471,16): warning CS1030: #warning: "Same move allowed multiple times."

Build complete -- 0 errors, 1 warnings

By including the #warning directive, you ensure that the compiler will report a warning, as shown in Output 3.29. This particular warning is a way of flagging the fact that there is a potential enhancement or bug within the code. It could be a simple way of reminding the developer of a pending task.

Turning Off Warning Messages (#pragma)

Warnings are helpful because they point to code that could potentially be troublesome. However, sometimes it is preferred to turn off particular warnings explicitly because they can be ignored legitimately. C# 2.0 and later compilers provide the preprocessor #pragma directive for just this purpose (see Listing 3.59).

Listing 3.59: Using the Preprocessor #pragma Directive to Disable the #warning Directive

```csharp
#pragma warning disable 1030
```

Note that warning numbers are prefixed with the letters CS in the compiler output. However, this prefix is not used in the #pragma warning directive. The number corresponds to the warning error number emitted by the compiler when there is no preprocessor command.

To reenable the warning, #pragma supports the restore option following the warning, as shown in Listing 3.60.

Listing 3.60: Using the Preprocessor #pragma Directive to Restore a Warning

```csharp
#pragma warning restore 1030
```

In combination, these two directives can surround a particular block of code where the warning is explicitly determined to be irrelevant.

Perhaps one of the most common warnings to disable is CS1591. This warning appears when you elect to generate XML documentation using the
C# Preprocessor Directives

/doc compiler option, but you neglect to document all of the public items within your program.

**nowarn:**<warn list> Option

In addition to the #pragma directive, C# compilers generally support the nowarn:<warn list> option. This achieves the same result as #pragma, except that instead of adding it to the source code, you can insert the command as a compiler option. The nowarn option affects the entire compilation, whereas the #pragma option affects only the file in which it appears. Turning off the CS1591 warning, for example, would appear on the command line as shown in Output 3.30.

**Output 3.30**

```shell
> csc /doc:generate.xml /nowarn:1591 /out:generate.exe Program.cs
```

**Specifying Line Numbers (#line)**

The #line directive controls on which line number the C# compiler reports an error or warning. It is used predominantly by utilities and designers that emit C# code. In Listing 3.61, the actual line numbers within the file appear on the left.

**Listing 3.61: The #line Preprocessor Directive**

```
124  #line 113 "TicTacToe.cs"
125  #warning "Same move allowed multiple times."
126  #line default
```

Including the #line directive causes the compiler to report the warning found on line 125 as though it was on line 113, as shown in the compiler error message in Output 3.31.

**Output 3.31**

```
Performing main compilation...
...	ictactoe.cs(113,18): warning CS1030: #warning: '"Same move allowed multiple times."'
Build complete -- 0 errors, 1 warnings
```
Chapter 3: Operators and Control Flow

Following the #line directive with default reverses the effect of all prior #line directives and instructs the compiler to report true line numbers rather than the ones designated by previous uses of the #line directive.

Hints for Visual Editors (#region, #endregion)

C# contains two preprocessor directives, #region and #endregion, that are useful only within the context of visual code editors. Code editors, such as Microsoft Visual Studio, can search through source code and find these directives to provide editor features when writing code. C# allows you to declare a region of code using the #region directive. You must pair the #region directive with a matching #endregion directive, both of which may optionally include a descriptive string following the directive. In addition, you may nest regions within one another.

Listing 3.62 shows the tic-tac-toe program as an example.

Listing 3.62: #region and #endregion Preprocessor Directives

```csharp
#region Display Tic-tac-toe Board

#if CSHARP2PLUS
    System.Console.Clear();
#endif

// Display the current board;
border = 0; // set the first border (border[0] = "|")

// Display the top line of dashes.
// ("\n---+---+---\n")
System.Console.Write(borders[2]);
foreach (char cell in cells)
{
    // Write out a cell value and the border that comes after it.
    System.Console.Write($" { cell } { borders[border] }");

    // Increment to the next border.
    border++;

    // Reset border to 0 if it is 3.
    if (border == 3)
    {
        border = 0;
    }
}
#endregion Display Tic-tac-toe Board
```
These preprocessor directives are used, for example, with Microsoft Visual Studio. Visual Studio examines the code and provides a tree control to open and collapse the code (on the left-hand side of the code editor window) that matches the region demarcated by the \#region directives (see Figure 3.5).

![Figure 3.5: Collapsed Region in Microsoft Visual Studio .NET](image_url)

**SUMMARY**

This chapter began by introducing the C# operators related to assignment and arithmetic. Next, we used the operators along with the const keyword to declare constants. Coverage of all the C# operators was not sequential, however. Before discussing the relational and logical comparison operators, the chapter introduced the if statement and the important concepts of code blocks and scope. To close out the coverage of operators, we discussed the bitwise operators, especially regarding masks. We also discussed other control flow statements such as loops, switch, and goto, and ended the chapter with a discussion of the C# preprocessor directives.
Chapter 3: Operators and Control Flow

Operator precedence was discussed earlier in the chapter; Table 3.5 summarizes the order of precedence across all operators, including several that are not yet covered.

**Table 3.5: Operator Order of Precedence**

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>x.y f(x) a[x] x++ x-- new</td>
</tr>
<tr>
<td></td>
<td>typeof(T) checked(x) unchecked(x) default(T)</td>
</tr>
<tr>
<td></td>
<td>nameof(x) delegate{} delegate()</td>
</tr>
<tr>
<td>Unary</td>
<td>+ - ! ~ ++x --x (T)x await x</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>Additive</td>
<td>+ -</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>Relational and type testing</td>
<td>&lt; &gt; &lt;= &gt;= is as</td>
</tr>
<tr>
<td>Equality</td>
<td>== !=</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>Logical XOR</td>
<td>^</td>
</tr>
<tr>
<td>Logical OR</td>
<td></td>
</tr>
<tr>
<td>Conditional AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Conditional OR</td>
<td></td>
</tr>
<tr>
<td>Null coalescing</td>
<td>??</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
</tr>
<tr>
<td>Assignment and lambda</td>
<td>= *= /= %= += -= &lt;&lt;= &gt;&gt;= &amp;= ^=</td>
</tr>
</tbody>
</table>

* Rows appear in order of precedence from highest to lowest.

Perhaps one of the best ways to review all of the content covered in Chapters 1–3 is to look at the tic-tac-toe program found in Appendix B. By reviewing this program, you can see one way in which you can combine all that you have learned into a complete program.
Index

Operators
- (minus sign)
  arithmetic subtraction operator, 91–92
  delegate operator, 551–552
  precedence, 92
  subtraction operator, overloading, 397–399
  unary operator, 90–91
( ) (parentheses)
  for code readability, 93–94
  grouping operands and operators, 93–94
  guidelines, 94
_ (underscore)
  in identifier names, 7
  line continuation character, 11
  in variable names, 15
{ } (curly braces)
  formatting code, 13
  forming code blocks, 114–116
  in methods, 9, 10–11
  omitting, 116
  as string literals, 54
@ (at sign)
  coding verbatim strings, 48
  inserting literal backslashes, 49
  keyword prefix, 8
+ (plus sign)
  addition operator, overloading, 397–399
  arithmetic binary operator, 91–92
  with char type data, 96
  concatenating strings, 95
  delegate operator, 551–552
determining distance between two characters, 96
with non-numeric operands, 95
precedence, 92
unary operator, 90–91
+= (plus sign, equal)
  binary/assignment operator, 399
  delegate operator, 550–552
-= (minus sign, equal)
  binary/assignment operator, 399
  delegate operator, 550–552
- (hyphens), in identifier names, 7
__ (two underscores), in keyword names, 8
; (semicolon), ending statements, 6–7, 11
.... (periods), download progress indicator, 779
" " (double quotes), coding string literals, 48
[ ] (square brackets), array declaration, 72–74
\ (backslashes), as literals, 49
$ (dollar sign), string interpolation, 48
$@ (dollar sign, at sign), string interpolation, 50
< > (angle brackets), in XML, 25
& (ampersand) AND operator, 131, 132
&& (ampersands) AND operator, 121
= (equal sign) assignment operator, 16, 118
== (equality operator), C++ vs. C#, 118 assigning variables, 16
definition, 16
precedence, 92
... (ellipsis) binary/assignment operator, 399
*= (asterisk, equal sign) binary/assignment operator, 399
%= (percent sign, equal) binary/assignment operator, 399
/= (slash, equal) binary/assignment operator, 399
&= (ampersand, equal sign) compound assignment operator, 133–134
^= (caret, equal sign) compound assignment operator, 133–134
|= (vertical bar, equal sign) compound assignment operator, 133–134
?: (question mark, colon) conditional operator, 123–124
-- (minus signs) decrement operator
C++ vs. C#, 105
decrement, in a loop, 102–105
description, 101–102
guidelines, 105
Lock statement, 105–106
postfix increment operator, 104–105
post-increment operator, 103
prefix increment operator, 104–105
pre-increment operator, 103–104
race conditions, 105–106
thread safety, 105–106
/ (forward slash) division operator
description, 91–92
overloading, 396–399
precedence, 92
. (dot) dot operator, 126, 871
== (equal signs) equality operator
overloading, 396–397
in place of = (equal sign) assignment operator, 119–120
/ (forward slash) in XML, 25
++ (plus signs) increment operator
C++ vs. C#, 105
decrement, in a loop, 102–105
description, 101–102
guidelines, 105
Lock statement, 105–106
postfix increment operator, 104–105
post-increment operator, 103
prefix increment operator, 104–105
pre-increment operator, 103–104
race conditions, 105–106
thread safety, 105–106
!= (exclamation point, equal sign) inequality operator
overloading, 362, 396–397
testing for inequality, 119–120
< (less than sign) less than operator, 120, 396–397
<= (less than or equal to) less than or equal operator, 120, 396–397
! (exclamation point) logical NOT operator, 122
% (percent sign) modulo, 91–92, 397–399
* (asterisk) multiplication operator, 91–92, 397–399
? (question mark) nullable modifier, 64–65, 459
?? (question marks) null-coalescing operator, 124–125, 126
?. (question mark, dot) null-conditional operator, 125–128
| (vertical bar) OR operator, 131, 132, 397–399
|| (vertical bars) OR operator, 121
<<= (shift left assignment operator, 130
<= (less than or equal to) shift left assignment operator, 130
> (greater than sign), greater than operator, 120, 396–397
>= (greater than or equal) greater than or equal operator, 120, 396–397
^ (caret) XOR operator, 122, 131, 397–399
\\ (single backslash character), escape sequence, 46
> (greater than sign), greater than operator, 120, 396–397
>= (greater than or equal) greater than or equal operator, 120, 396–397
=> (equal sign, greater than) lambda operator, 517, 524
>> (shift right), shift right operator, 130, 397–399
>>= (shift right assignment operator, 130
~ (tilde) bitwise complement operator, 134

A
Abort() method, 745–746
Aborting threads, 745–746
Abstract classes. See also Derivation.
defining, 314–331
definition, 314
derived from System.Object, 320–321
vs. interfaces, 338
polymorphism, 318–320
Abstract members
defining, 315–317
definition, 314
“is a” relationships, 317
overriding, 317
virtual, 317
Access modifiers. See also Encapsulation.
definition, 233–235
on getters and setters, 251–252
purpose of, 236
Action delegates, 524–525
Activation frame, 182–183
Add() method
appending items to lists, 648
inserting dictionary elements, 654–655
System.Threading.Interlocked class, 829
thread synchronization, 829–830
Addresses. See Pointers and addresses.
Aggregate functions, 618
AggregateException, 557–558, 762–765
AggregateException.Flatten() method, 780
AggregateException.Handle() method, 764, 780
Aggregation
derivation, 299–301
interfaces, 343–344
multiple inheritance, interfaces, 343–344
Aliasing, namespaces, 179–180. See also using directive.
AllocExecutionBlock() method, 857
AllowMultiple member, 707
Alternative flow control statements, 111
Ampersand, equal sign (&=) compound assignment operator, 133–134
Ampersand (&) AND operator, 131, 132
Ampersands (&&) AND operator, 121
Angle brackets (< >), in XML, 25
Anonymous functions
definition, 516, 517
guidelines, 533
Anonymous methods. See also Lambda expressions.
definition, 522
guidelines, 523
internals, 527–528
parameterless, 523
passing, 522–523
Anonymous types
definition, 61, 572
explicit local variables, 263–265
generating, 578
implicit local variables, 572–576
in query expressions, 625–626
type incompatibilities, 576–577
type safety, 576–577
var keyword, 572–576
Antecedent tasks, 757
Apartment-threading models, 846
APIs (application programming interfaces)
calls from P/Invoke, wrappers, 860–861
definition, 25
deprecated, 712
as frameworks, 26
Append method, 58
AppendFormat method, 58
Appending items to collections, 648
Applicable method calls, 201
Applications, compiling, 3
Appointment, 291–292
__ arglist keyword, 8
ArgumentNullException, 434–435, 436
ArgumentOutOfRangeException, 435, 436
Arguments
calling methods, 163, 167–168
named, calling methods, 199
Arity (number of type parameters), 471–472
Array accessor, 78–79
Array declaration
C++ vs. C#, 72–74
code example, 78–79
Java vs. C#, 72–74
Array types, constraint limitations, 484
ArrayList type, 363–365
Arrays. See also Collections; Lists;
TicTacToe game.
accessing elements of, 73, 78–79
of arrays, 78
assigning values to, 73–76
binary search, 81–83
BinarySearch() method, 81–83
Clear() method, 81–83
clearing, 81–83
common errors, 86–87
converting collections to, 646
description, 71–72
Arrays (continued)
  designating individual items, 71
  exceeding the bounds of, 80–81
  GetLength() method, 83–84
  indexers, defining, 665–666
  instantiating, 74–76
  jagged, 78, 79, 81
  length, getting, 80–81
  Length member, 80
  multidimensional, 74, 77–79
  number of dimensions, 72
  number of items, getting, 80–81
  as operator, 322–323
  palindromes, 84–86
  rank, 72, 83–84
  Reverse() method, 85–86
  reversing, 81–82
  reversing strings, 84–86
  searching, 81–83, 651–652
  size, specifying, 75
  sorting, 81–82, 82–83
  strings as, 84–86
  swapping data elements, 79
  three-dimensional, 77–78
  ToCharArray() method, 85–86
  two-dimensional, 74, 79. See also TicTacToe game.
  type defaults, 73
  unsafe covariance, 497–498
AsParallel() method, 595
AspNetSynchronizationContext, 794
Assemblies, compiling, 3–4
Assembly, definition, 3–4
Assembly attributes, 697–698
Assert() method, 97
Association, 225–227, 269
Associativity of operators, 92, 93–94
Asterisk, equal sign (*=) binary / assignment operator, 399
Asterisk (*) multiplication operator, 91–92, 397–399
async keyword
  purpose of, 786
  task-based asynchronous pattern, 781–786
  Windows UI, 795–798
  in WinRT, 876
Asynchronous continuations, 756–762
Asynchronous delays, 745
Asynchronous high-latency operations with the TPL, 777–781
Asynchronous lambdas, 786–787
Asynchronous methods, 787–791
Asynchronous operations, 736, 741–743
Asynchronous tasks. See Multithreading, asynchronous tasks.
AsyncState, 755
At sign (@)
  coding verbatim strings, 48
  inserting literal backslashes, 49
  keyword prefix, 8
Atomic operations, threading problems, 738
Atomicity of reading and writing to variables, 819
AttachedToParent enum, 758
Attributes
  adding encryption, 715–716
  adding metadata about assemblies, 697–698
  alias command-line options, 701
  AllowMultiple member, 707
  assembly, 697–698
  CIL for, 718–719
  custom, 699–700
  custom serialization, 714–715
  decorating properties with, 696–697
  definition, 683
  deserializing objects, 714–715
  duplicate names, 707
  guidelines, 699, 700, 705, 707
  initializing with a constructor, 701–705
  vs. interfaces, 349
  named parameters, 707
  no-oping a call, 710–711
  Parse() method, 709
  predefined, 709
  pseudoattributes, 719
  retrieving, 700–702
  return, 698–699
  serialization-related, 713–714
  setting bits or fields in metadata tables. See Pseudoattribute.
  uses for, 696
  versioning, 716–718
  warning about deprecated APIs, 712
Automatically implemented properties
  description, 240–242
  initializing, 242
  internals, 254
  NextId implementation, 273
  read-only, 248, 280
Average() method, 618
await keyword
non-Task<T> values, 791–792
task-based asynchronous pattern, 781–786
Windows UI, 795–798

await operator
description, 797–798
multithreading with System.Threading.Thread class, 745
in WinRT, 876

B
Backslashes (\), as literals, 49
Base classes, inheritance, 302
Base classes, overriding. See also Derivation.
accessing a base member, 312–313
base keyword, 313
brittle base class, 307–311
constructors, 313–314
fragile base class, 307–311
introduction, 302
new modifier, 307–311
override keyword, 304, 313
sealed modifier, 311–312
sealing virtual members, 311–312
virtual methods, 302–307
virtual modifier, 302–307
base keyword, 313
Base members, accessing, 312–313
Base type, 220
BCL (Base Class Library), 27, 28, 893, 895
Binary digits, definition, 128
Binary display, string representation of, 132
Binary floating-point types, precision, 97
Binary operators, 397–399
Binary search of arrays, 81–83
BinaryExpression, 535
BinarySearchTree() method
bitwise complement of, 652
searching a list, 651–652
searching arrays, 81–83
BinaryTreeNode<T>, 476-477, 670–671
Bits, definition, 128
Bitwise complement of BinarySearchTree() method, 652
Bitwise operators
<< (less than signs), shift left operator, 130
<<= (less than, equal signs), shift left assignment operator, 130
& (ampersand) AND operator, 131, 132, 378
&= (ampersand, equal sign) compound assignment operator, 133-134
^= (caret, equal sign) compound assignment operator, 133-134
|= (vertical line, equal sign) compound assignment operator, 133-134
| (vertical bar) OR operator, 131, 132, 378
^ (caret) XOR operator, 131
>> (greater than signs), shift right operator, 130
>>= (greater than, equal signs), shift right assignment operator, 130
~ (tilde) bitwise complement operator, 134
binary digits, definition, 128
bits, definition, 128
bytes, definition, 128
introduction, 128–129
logical operators, 131–133
masks, 132
multiplication and division with bit shifting, 130
string representation of a binary display, 132

Block statements. See Code blocks.
BlockingCollection<T>, 840
bool (Boolean) types
description, 45
returning from lambda expressions, 520

Boolean expressions. See also Bitwise operators.
< (less than sign), less than operator, 120
<= (less than, equal sign), less than or equal operator, 120
== (equal signs) equality operator, 119–120
!= (exclamation point, equal sign) inequality operator, 119–120
> (greater than sign), greater than operator, 120
>= (greater than, equal sign), greater than or equal operator, 120
definition, 118–119
equality operators, 119–120
evaluating. See if statements.
example, 118
relational operators, 119-120
Boolean expressions, logical operators
  ! (exclamation point), logical negation operator, 122
  ?: (question mark, colon), conditional operator, 123–124
  ?. (question mark, dot), null-conditional operator, 125–128
  ?? (question marks), null-coalescing operator, 124–125, 126
  && (ampersands), AND operator, 121
  ^ (caret), XOR operator, 122
  || (vertical lines), OR operator, 121
  . (dot) dot operator, 126
Booleans, introduction, 120

Binary operators
  = (assignment operator) vs. == (equality operator), 118
  array declaration, 72
  delete operator, 224
  deterministic destruction, 427, 884
  explicit deterministic resource cleanup, 224
  garbage collection, 884
  global methods, 171
  global variables and functions, 266
  header files, 174
  implicit deterministic resource cleanup, 224
  implicit nondeterministic resource cleanup, 224
  implicit overriding, 304
  increment/decrement operators, 105
  local variable scope, 118
  main() method, 10
  method calls during construction, 307
  multiple inheritance, 299
  operator order of precedence, 105
  operator-only statements, 91
  order of operations, 94
  partial methods, 174
  pointer declaration, 865
  preprocessing, 152
  pure virtual functions, 317
  string concatenation at compile time, 50
  switch statement fall-through, 145
  var keyword, 575
  Variant, 575
  void*, 575
  void type, 59

Boxing
  avoiding during method calls, 369–370
  code examples, 363–364
  introduction, 362–363
  InvalidCastException, 366
  performance, 365
  synchronizing code, 366–368
  unboxing, 363–364, 366
  value types in the lock statement, 366–368

Break() method, 808
break statement, 110, 146–147
Breaking parallel loop iterations, 808
Brittle base class, 307–311
BubbleSort() method, 506–510
byte type, 36
Bytes, definition, 128

C
  C language
    pointer declaration, vs. C#, 865
    similarities to C#, 2
  C# language
    case sensitivity, 2
    compiler, 3
    definition, 895
  C++ language, similarities to C#, 2
  C++ language vs. C#
    = (assignment operator) vs. ==
      (equality operator), 118
    array declaration, 72
    delete operator, 224
    deterministic destruction, 427, 884
    explicit deterministic resource cleanup, 224
    garbage collection, 884
    global methods, 171
    global variables and functions, 266
    header files, 174
    implicit deterministic resource cleanup, 224
    implicit nondeterministic resource cleanup, 224
    implicit overriding, 304
    increment/decrement operators, 105
    local variable scope, 118
    main() method, 10
    method calls during construction, 307
    multiple inheritance, 299
    operator order of precedence, 105
    operator-only statements, 91
    order of operations, 94
    partial methods, 174
    pointer declaration, 865
    preprocessing, 152
    pure virtual functions, 317
    string concatenation at compile time, 50
    switch statement fall-through, 145
    var keyword, 575
    Variant, 575
    void*, 575
    void type, 59
  Caching data in class collections, 600
  Calculate() method, 753
  Call site, 182–183
  Call stack, 182–183
  Caller, 163
  Calling
    constructors, 255–256, 261–262
    methods. See Methods, calling.
    object initializers, 257–259
  CamelCase, 7
  Cancel() method, 770–771
  Canceling
    parallel loop iterations, 805–806
    PLINQ queries, 811–813
    tasks. See Multithreading, canceling tasks.
    CancellationToken, 769–772
    CancellationTokenSource, 770
    CancellationTokenSource.Cancel() method, 770–771
  Capacity() method, 647
  Captured variables, 528–530
  Capturing loop variables, 531–533
Caret, equal sign (\(^=\)) compound assignment operator, 133–134
Caret (\(^\)\)) XOR operator, 122, 131, 397–399
Cartesian products, 608, 638–639
Casing
  formats for identifiers, 6–7
  local variables, 15
Cast operator
  defining, 295–296
  definition, 65–66
  overloading, 402
Casting
  between arrays of enums, 375
  between base and derived types, 293–294
  definition, 65
  explicit cast, 65, 294
  implicit base type casting, 293–294
  with inheritance chains, 294–295
  inside generic methods, 490–491
  type conversion without, 69–70
Catch blocks
  catching different exception types, 436–437
  description, 205–206
  empty, 442–443
  general, 440–442
  with no type, 211–212
Catch clause, 762
Catch() method, 439
Catching exceptions
  catch blocks, 436–437
  code sample, 204–205
  conditional clauses, 438
  definition, 204–209
  description, 436
  different exception types, 436–437
  empty catch blocks, 442–443
  exception conditions, 438
  general catch blocks, 440–442
  rethrowing existing exceptions, 438–439
  switch statements, 436
  when clauses, 438
Central processing unit (CPU),
  definition, 734
Chaining
  constructors, 261–262
  inheritance, 292–293
  multicast delegates, 555
  tasks, 757
Changing strings, 56–57
char (character) types, 14, 45
Checked block example, 67
Checked conversions, 66–68
Checking for
  null guidelines, 549
  multicast delegates, 548–549
Child type, 220
Church, Alonzo, 523–524
CIL (Common Intermediate Language).
  See also CLI (Common Language Infrastructure).
  compiling C# source code into, 26, 891–892
  compiling into machine code, 878
  custom attributes, 894–895
  definition, 895
  disassembling, tools for, 33. See also
  ILDASM.
  managed execution, 26–28
  metadata, 894–895
  reflection, 894–895
  sample output, 31–33
  source languages, 31–33
CIL disassembler. See ILDASM.
Class collections. See also IEnumerable
  interface; IEnumerable\(\langle T\rangle\) interface.
  cleaning up after iteration, 586–587
  error handling, 587
  iterating over using while(), 584
  resource cleanup, 587
  sharing state, 585
  sorting, 601–603
Class collections, foreach loops
  with arrays, 385
  code example, 586
  with IEnumerable interface, 587
  with IEnumerable\(\langle T\rangle\) interface, 583–585
  modifying collections during, 587–588
Class collections, sorting. See also
  Standard query operators, sorting.
  ascending order ThenBy(), 601–603
  ascending order with OrderBy(), 601–603
  descending order with
    OrderByDescending(), 602
  descending order with
    ThenByDescending(), 602
Class definition
  definition, 8
  guidelines, 8
  naming conventions, 8
  syntax, 8–9
Index

class keyword, 478–479
Class libraries
definition, 404
referencing, 404–405
Class members, definition, 224
Class type
combining with class or struct, 482
constraints, 477–478
class vs. struct, 677
Classes
abstract. See Abstract classes.
adding instance methods, 276
association, 225–227, 269
association with methods, 163
base. See Base classes.
within classes. See Nested, classes.
declaring, 221–224
definition, 222–223
derived. See Derivation.
fields, 225
guidelines, 222
identifying support for generics, 692–693
inextensible, 273–275
instance fields, 225–227
instance methods, 227–228
instantiating, 221–224
vs. interfaces, 347–348
member variables, 225
nested, 281–283
partial, 284–285
polymorphism, 221
private members, 236
refactoring, 290–291
sealed, 301–302
spanning multiple files, Java vs. C#, 4.
See also Partial methods.
splitting across multiple files. See
Partial methods.
static, 273–275
Clear() method, 81–83, 154
Clearing arrays, 81–83
CLI (Common Language Infrastructure).
See also CIL (Common Intermediate
Language); VES (Virtual Execution
System).
application domains, 888
assemblies, 888–891
compilers, 879–881
contents of, 879
CTS (Common Type System), 892
definition, 878, 895
description, 878–879
implementations, 879
managed execution, 26–27
manifests, 888–891
modules, 888–891
objects, 892
values, 892
xcopy deployment, 891
Closed over variables, 528–530
Closures, 531
CLR (Common Language Runtime), 896.
See also Runtime.
CLS (Common Language Specification)
BCL (Base Class Library), 893
definition, 896
description, 893
FCL (Framework Class Library), 893
managed execution, 27
CLU language, 667–668
Code access security, 27
Code blocks, 114–116
Code readability
vs. brevity, 169
indenting with whitespace, 12–13
Coding the observer pattern with
multicast delegates
checking for null, 548–549
connecting publisher with subscribers,
546–547
defining subscriber methods, 544–545
defining the publisher, 545–546
delegate operators, 550–552. See also
specific operators.
getting a list of subscribers, 557–558
guidelines, checking for null, 548–549
invoking a delegate, 547
method returns, 558
multicast delegate internals, 554
new delegate instances, 550
passing by reference, 558
removing delegates from a chain,
550–552
sequential invocation, 552, 554
thread safe delegate invocation, 550
Cold tasks, 752
Collect() method, 419
Collection classes
dictionary collections, 653–657
linked list collections, 663
list collections, 646–649
queue collections, 662
sorted collections, 660–661
sorting lists, 649–650
stack collections, 661–662
Collection initializers
definition, 578
description, 258–259
initializing anonymous type arrays, 579–582
initializing collections, 579
Collection interfaces, customizing
appending items to, 648
comparing dictionary keys, 658–659
converting to arrays, 646
counting collection elements, 646
dictionary class vs. list, 644–646
finding even elements, 653
finding multiple items, 652–653
generic hierarchy, 645
inserting new elements, 651–652
lists vs. dictionaries, 644–645
order of elements, 657
removing elements, 649
search element not found, 651–652
searching arrays, 651–652
searching collections, 651–653
specifying an indexer, 644–646
Collection interfaces with standard query operators
caching data, 600
counting elements with Count(), 595–596
defered execution, 597–598, 600–601
definition, 588
filtering with Where(), 591–592, 597–598
guidelines, 596
projecting with Select(), 592–594
queryable extensions, 619
race conditions, 595
running LINQ queries in parallel, 594–595
sample classes, 588–591
sequence diagram, 599
table of, 618
Collections. See also Anonymous types; Arrays; Class collections; Lists.
discarding duplicate members, 639–640
filtering, 622
projecting, 622
returning distinct members, 639–640
Collections, customizing
accessing elements without modifying the stack, 661–662
appending items to, 648
counting elements of, 646
empty, 666–667
FIFO (first in, first out), 662
finding even elements, 653
finding multiple items, 652–653
indexers, defining, 665–666
inserting new elements, 651–652, 661–662
LIFO (last in, first out), 661
order of elements, 657
removing elements, 649
requirements for equality comparisons, 659–660
search element not found, 651–652
searching, 651–653
Collections, sorting. See also Standard query operators, sorting.
by file size, 633–634
by key, 660–661
with query expressions, 632–633
by value, 660–661
COM threading model, controlling, 846
Combine() method
combining delegates, 552
constraint limitations, 484
event internals, 568
vs. Swap() method, 186
CommandLine, 281–285
CommandLineAliasAttribute, 701–702
CommandLineInfo, 687–691, 696
CommandLineSwitchRequiredAttribute, 699–701
Comments
vs. clear code, 24
delimited, 24
guidelines, 24
multi-line, 154
preprocessor directives as, 154
single-line, 24
types of, 24
XML delimited, 24
XML single-line, 24
Common Intermediate Language (CIL).
See CIL (Common Intermediate Language).
Common Language Infrastructure (CLI).
See CLI (Common Language Infrastructure).
The Common Language Infrastructure
Annotated Standard, 26
Common Language Runtime. See Run-time.
Common Language Runtime (CLR), 896. See also Run-time.
Common Type System (CTS), 27, 892, 896
Compare() method, 45, 505
CompareExchange() method, 828–829
CompareExchange<T> method, 829
CompareTo() method, 477, 649–650
Comparing dictionary keys, 658–659
for equality, float type, 97–100
Comparison operators, 396–397
ComparisonHandler delegate, 509, 510–512, 514–515
Compatible method calls, 201
Compile() method, 535
Compilers
C# language, 3
CLI (Common Language Infrastructure), 879–882
CoreCLR compiler, 880
csc.exe compiler, 3
DotGNU Portable NET compiler, 880
JIT (just-in-time) compiler, 881
mcs.exe compiler, 3
Microsoft Silverlight compiler, 880
Mono, 3
Mono compiler, 3, 880
.NET Compact Framework, 880
.NET Micro Framework, 880
Shared Source CLI, 880
Windows Desktop CLR, 880
Compiling applications, 3
assemblies, 3–4
C# source code into CIL, 891–892
into CIL, 26
jitting, 881
just-in-time, 26, 881
with the Mono compiler, 3
NGEN tool, 881
Complex memory models, threading problems, 739
Composite formatting, 21
Compress() method, 326–327
Concat() method, 618
Concatenating strings, 95
Concrete classes, 314, 317
Concurrent collection classes, 840–841
Concurrent operations, definition, 736
ConcurrentBag<T>, 840
ConcurrentDictionary<T>, 840
ConcurrentQueue<T>, 840
ConcurrentStack<T>, 840
Conditional clauses, catching exceptions, 438
expressions, guidelines, 124
logical operators, overloading, 400
Conditions, 111
ConnectionState, 372
Consequence statements, 111
Console executables, 404
Console input, 18–19
Console output
commenting code, 22–24
comments, types of, 24
composite formatting, 21
format items, 21
format strings, 21
formatting with string interpolation, 20
overview, 19–22
ConsoleListControl, 327–331, 336
const field, encapsulation, 277–278
const keyword, 106–107
Constant expressions, 106–107
Constant locals, 106–107
Constants declaring, 107
definition, 106
guidelines, 106
vs. variables, guidelines, 106
Constraints on type parameters. See also Contravariance; Covariance.
class type constraints, 477–478
constructor constraints, 480
generic methods, 481–482
inheritance, 480–482
interface type constraints, 476–477
introduction, 473–476
multiple constraints, 479
non-nullable value types, 478–479
reference types, 478–479
Constraints on type parameters, limitations
array types, 484
combining class type with class, 482
combining class type with struct, 482
don constructors, 484-486
delegate types, 484
enumerated types, 484
operator constraints, 482-483
OR criteria, 483
restricting inheritance, 482
sealed types, 484
Construction initializers, 261-262
Constructor constraints, 480
Constructors
calling, 255-256
calling one from another, 261-262
centralizing initialization, 262-263
chaining, 261-262
collection initializers, 258-259
constraints, 484-486
declaring, 255-256
default, 256-257
definition, 255
exception propagation from, 428
finalizers, 259
in generic types, declaring, 468-469
guidelines, 261
introduction, 254
new operator, 256
object initializers, 257-259
overloading, 259-261
 overriding base classes, 313-314
static, 271-272
Contains() method, 651-652, 661-662
ContainsValue() method, 656
Context switch, definition, 736
Context switching, 737
Contextual keywords, 6, 679-680
Continuation clauses, query expressions, 637-638
Continuation tasks, 757
continue statement
description, 148-150
guidelines, 144
syntax, 109
ContinueWith() method, 756-758, 760-761, 764-765, 779
Contracts vs. inheritance, 340-341
Contravariance
definition, 495
delegates, 526-527
enabling with in modifier, 495-497
Control flow. See also Flow control.
guidelines, 139
misconceptions, 784
task continuation, 755
within tasks, 784-786
Conversion operators, overloading, 401, 403
Converting
collections to arrays, 646
enums to and from strings, 375-377
between interfaces and implementing
types, 338
types. See Types, conversions between.
Cooler objects, 544-546
Cooperative cancellation, definition, 769
Copy() method, 269-271, 275-277
CopyTo() method, 646
CoreCLR compiler, 880
Count() method, 595-596, 618
Count property, 596, 646
CountdownEvent, 839-840
Counting
class collection elements with Count(), 595-596
collection elements, 646
lines within a file, example, 192-193, 194-197
CountLines() method, 163
Covariance
definition, 491
delegates, 526-527
enabling with out modifier, 492-494
guidelines, 498
introduction, 491-492
preventing, 492
type safety, 498
unsafe covariance in arrays, 497-498
Covariant conversion
definition, 491
restrictions, 494
.cpp file, C++ vs. C#, 174
CPU (central processing unit),
definition, 734
Create() factory method, 723-724
Create() method, 471-472
.cs file extension, 3
csc.exe compiler, 3
CTS (Common Type System), 27, 892, 896
Curly braces ({ })
formatting code, 13
forming code blocks, 114-116
in methods, 9, 10-11
omitting, 116
as string literals, 54
Index

CurrentTemperature property, 546–547

Custom asynchronous methods, 787–791

Custom dynamic objects, 726–728

Custom serialization attributes, 714–715

Customizing collection interfaces. See Collection interfaces, customizing.
collections. See Collections, customizing.
ev ents, 568–569
IEnunmerable interface. See Iterators.
IEnunmerable<T> interface. See Iterators.
IEnumerator<T> interface. See Iterators.

Customizing, exceptions
defining, 446–448
guidelines, 448
serializable exceptions, 449

D
Data
on interfaces, 327
retrieval from files, 233–235
Data persistence, 232–233
Data types. See Types.
DataStorage, 232–235

Deadlocks. See also Thread synchronization.
avoiding, 831–832
causes of, 832
non-reentrant locks, 832
prerequisites for, 832
threading problems, 740

Deallocating memory, finalizers, 423
Debugging with preprocessor directives, 154
declaration spaces, 116–118
Declaring
arrays, 72–74, 78–79
classes, 221–224
constants, 107
constructors, 255–256
delegate types, 510
events, 560–562
finalizers, 422
generic classes, 464
generic delegate types, events, 562
instance fields, 225–226
local variables, 13–14
methods. See Methods, declaring.
properties, 238–240
Decrement() method, 829–830

Default constructors, 256–257
Default keyword, 73
default operator, 360–361
Default types, 40–42
Default(bool) keyword, 73
DefaultIfEmpty(), 613–614
Default(int) keyword, 73
Deferred execution
implementing, 630–631
query expressions, 627–630
standard query operators, 597–598, 600–601
#define preprocessor directive, 153, 155
Delay() method, 745, 845–846
Del aying code execution. See Thread synchronization, timers.
delegate keyword, 510, 554
Delegate operators, 550–552. See also specific operators.
Delegate types, 484, 508–510

Delegates
-= (minus sign, equal sign), delegate operator, 550
BubbleSort() example, 506–510
contravariance, 526–527
covariance, 526–527
creating with a statement lambda, 517–518
defered execution, 630–631
definition, 506
executing unsafe code, 872–873
vs. expression trees, 536–537
general purpose, 524–525
guidelines, 525
immutability, 513
instantiating, 510–512
internals, 513–516
invocation sequence diagram, 553
mapping to function pointers, 861–862
method group conversion, 512
method names as arguments, 511–512
multicast, 543, 547. See also Coding the observer pattern with multicast delegates.
nesting, 510
with the null-conditional operator, 128
passing with expression lambdas, 520
structural equality, 526–527
synchronous, 751
System.Action, 524–525
System.Func, 524–525
vs. tasks, 751
thread safety, 550
delete operator, C++ vs. C#, 224
Delimited comments, 24
DenyChildAttach enum, 758
Deprecated APIs, 712
Dequeue() method, 662
Dereferencing pointers, 869–871
Derivation
abstract classes. See Abstract classes.
aggregation, 299–301
casting between base and derived
types, 293–294
casting with inheritance chains, 294–295
classes derived from System.Object, 320–321
data conversion with the as operator, 322–323
defining custom conversions, 295–296
determining the underlying type, 321–322
explicit cast, 294
extension methods, 299
implicit base type casting, 293–294
implicit conversion, 294
inheritance chains, 292–293
“is a” relationships, 293–294
is operator, 321–322
multiple inheritance, simulating, 299–301
overriding base classes. See Base
classes, overriding.
private access modifier, 296–297
protected access modifier, 297–298
refactoring a class, 290–291
sealed classes, 301–302
single inheritance, 299–301
Derived types, 220–221
Deserialize() method, 714
Deserializing
documents, 716–718
objects, 714–715
Deterministic finalization with the using
statement, 423–426
Diagramming multiple inheritance, 345
Dictionaries, 664–666
Dictionary classes
customized collection sorting, 649–651
definition, 653
diagrams, 654
hash tables, 656
inserting elements, 654
vs. list, 644–646
list collections, 647–649
vs. lists, 644–646
removing elements, 656
Dictionary collection classes, 644–646
Dictionary collections, 653–657
Dictionary keys, comparing, 658–659
Directives. See Preprocessor directives.
DirectoryCountLines() method, 193–199
Directory.GetFiles() method, 625, 632
DirectoryInfoExtension.Copy()
method, 269–271, 275–277
DirectoryInfo.GetFiles() method, 275, 607
Disassembling CIL, tools for, 33. See also
ILDASM.
Discarding duplicate collection members,
639–640
DispatcherSynchronizationContext, 794
Disposable tasks, 774
Dispose() method, 424–425, 426–427
Distinct() method, 618, 639–641
Dividing by zero, 99–100
Division with bit shifting, 130
.d11 file extension, 4
DLL (Dynamic Load Library) file
extension, 4
do while loops, 108, 134–137
Documents
deserializing, 716–718
serializing, 716–718
versioning, 716–718
XML, 416–418
Dollar sign, at sign ($@), string
interpolation, 50
Dollar sign ($), string interpolation, 48
Dot (.) dot operator, 126, 871
DotGNU Portable NET compiler, 880
Double quotes (“ “), coding string
literals, 48
double type, precision, 97
Double.TryParse() method, 215
Dropping namespaces. See using
directive.
Dump() method, 337
Duplicate names for attributes, 707
dynamic as System.Object, 723–724
Dynamic binding, 724–725
dynamic directive, 721
Dynamic Load Library (DLL) file
extension, 4
Dynamic member invocation, 722
dynamic principles and behaviors, 721–723
dynamic type. See Programming with dynamic objects.

E
#elsif preprocessor directive, 153, 154
Eliminating namespaces. See using directive.
Ellipsis (…) binary/assignment operator, 399
Else clauses, 111
#else preprocessor directive, 153, 154
E-mail domain, determining, 149
Empty catch blocks, 442–443
Empty<T> method, 667
Encapsulation. See also Access modifiers.

const field, 277–278
definition, 224
description, 223–224
information hiding, 235–237
introduction, 219
public constants, permanent values, 278–281
read-only fields, 279–281
readonly modifier, 279–281
of types, 407–408
Encryption for documents, 715–716
#endif preprocessor directive, 154
#region preprocessor directive, 158–159
Enqueue() method, 662
Enter() method, 366, 821, 823, 825–826
EntityBase, 477–478
EntityBase<T>, 481–482
EntityDictionary, 479, 500–501
EntityDictionary<T>, 477–478
EntityDictionary<TKey, TValue>, 480
Enumerated types, constraint limitations, 484
enums
casting between arrays of, 375
characteristics of, 371–372
conversion to and from strings, 375–377
defining, 372
definition, 371–372
as flags, 377–381
FlagsAttribute, 380–381
joining values, 378
replacing Boolean values, 372
type compatibility, 374–375
underlying type, 373
enums, guidelines
creating enums, 374
default type, 373
enum flags, 379
string conversions, 377
Equal sign, greater than (=>) lambda operator, 517, 524
Equal sign (=) assignment operator
vs. == (equality operator), C++ vs. C#, 118
assigning variables, 16
definition, 16
precedence, 92
Equal signs (==) equality operator
overloading, 396–397
in place of = (equal sign) assignment operator, 119–120
Equality operators, 119–120
Equals() equality operator
implementing, 392–395
overloading, 388–395
overriding, 392–395
requirements for equality comparisons, 659–660
Equals() method, overloading, 362
Error handling. See also Exception handling.

APIs, 855–857
class collections, 587
multicast delegates, 554–556
P/Invoke, 855–857
Error messages, disabling/restoring, 156–157
#error preprocessor directive, 153, 155–156
Errors. See also Exception handling, emitting with directives, 155–156
infinite recursion error, 194
reporting. See Exception handling.
Escape sequences
\ (single backslash character), 46
displaying a smiley face, 47–48
list of, 47
\n, newline character, 46, 48
\t, tab character, 46
for Unicode characters, 46–48
Even() method, 653
Event keyword, 558, 561
Event notification
with multiple threads, 830
thread-safe, 831
Events. See also Coding the observer pattern.
coding conventions, 562–564
customizing, 568–569
declaring, 560–562
declaring generic delegate types, 562
encapsulating the publication, 560
encapsulating the subscription, 559–560
generic delegates, 564–566
guidelines, 564, 566
internals, 564–566
Exception conditions, 438
Exception handling. See also Errors; specific exceptions.
with AggregateException, parallel loop iterations, 803–804
appropriate use of, 214–215
basic procedures, 202–203
catch blocks, 205–206, 211–212
catching exceptions, 204–209
common exception types, 210. See also specific types.
examples, 202–203, 204–205
for expected situations, 214–215
finally blocks, 207–208
general catch blocks, 211–212
guidelines, 212, 215, 436, 442–446
handling order, 207
Java vs. C#, 440
multiple exception types, 433–436
numeric conversion, 215–216
ParseException method, 215–216
passing null exceptions, 434
program flow, 206
reporting errors. See throw statement; Throwing exceptions.
task-based asynchronous pattern, 779–780
throwing exceptions, 203–204, 212–215.
See also throw statements.
trapping errors, 203–209
try blocks, 205–206
TryParse() method, 215–216
unhandled exceptions, 203–204
Exception propagation from constructors, resource cleanup, 428
Exceptions, custom. See also Errors.
defining, 446–448
guidelines, 448
serializable exceptions, 449
Exchange<T> method, 829
Exclamation point, equal sign (!=)
inquality operator
overloading, 362, 396–397
testing for inequality, 119–120
Exclamation point (!) logical NOT operator, 122
Excluding/including code with preprocessor directives, 154
ExecuteSynchronously enum, 759
Execution time, definition, 27
Exit() method, 366, 823, 825–826
Exiting a switch section, guidelines, 144
Explicit cast, 65–66
Explicit deterministic resource cleanup,
C++ vs. C#, 224
Explicit implementation of interfaces, 334, 336–337
Explicit member implementation, 334–335
Explicitly declared parameter types, 518
Expression bodied methods, 174
Expression lambdas, 520
Expression trees
BinaryExpression, 535
building LINQ queries, 537–538
Compile() method, 535
deferred execution, 630–631
definition, 533
vs. delegates, 536–537
examining, 538–541
lambda expressions as data, 534–535
LoopExpression, 535
MethodCallExpression, 535
NewExpression, 535
as object graphs, 535–536
ParameterExpression, 535
UnaryExpression, 535
Extensible Markup Language (XML).
See XML (Extensible Markup Language).
Extension methods
definition, 276
derivation, 299
inheritance, 299
on interfaces, 341–343
introduction, 275–277
reflection support for, 723
requirements, 276
extern alias directive, 413
extern methods, 851
External functions
  calling with P/Invoke, 858–860
declarating with P/Invoke, 850–851

F
Factory methods, generic types, 471–472
FailFast() method, 435, 436
FCL (Framework Class Library), 893, 896
Fibonacci calculator, 135
Fibonacci numbers/series, 135
Fields
  declaring as volatile, 828
  getter/setter methods, 237–238
  guidelines, 242–244
  identifying owner of, 228–229
  marking as private, 237–238
  virtual, properties as, 249–250
File extensions, 3. See also specific extensions.
FileInfo collections, projecting, 633–634
FileInfo object, 593, 625, 633
Filename matching class name, Java vs. C#,
Files
  data persistence, 232–233
  data retrieval, 233–235
  storage and loading, 232–235
FileSettingsProvider, 341
FileStream property, 429
Filtering collections
  definition, 622
  filtering criteria, 631–632
  predicates, 631
  query expressions, 631–632
Finalization
  guidelines, 428
  resource cleanup, 421–427
Finalization queue, resource cleanup, 426
Finalizers
  deallocating memory, 423
  declaring, 422
  description, 259, 421
deterministic finalization with the using statement, 423–426
Finally blocks, 207–208
FindAll() method, 652–653
Finding
  even elements in collections, 653
  multiple items in collections, 652–653
fixed statement, 867–868
Fixing (pinning) data, 866–868
Flags, enums, 377–381
FlagsAttribute, 380–381, 708–709
Flatten() method, 780
Flattening a sequence of sequences, 638–639
fnNewProtect, 854
float type, 97–100
Floating-point types
  binary float, 39
decimal, 38–39
double, 37–38
  for financial calculations. See decimal type.
float, 37–38
Flow control. See also Control flow.
definition, 734
introduction, 107
Flow control statements. See also specific statements.
  alternative, 111
  block statements. See Code blocks.
  Boolean expressions, evaluating. See if statement.
break, 110
code blocks, 114–116. See also Scope.
  combining. See Code blocks.
  conditions, 111
  consequence, 111
  continue, 109
declaration spaces, 116–118
declaration, 89
do while, 108, 134–137
for, 109, 137–140
foreach, 109, 140–142
goto, 110
if, 108, 111, 114
if/else, examples, 111, 113
indentation, 115
nested if, 112–113
scopes, 116–118. See also Code blocks.
switch, 110, 142–146
ture/false evaluation, 111
while, 108, 134–135, 136–137
For loops
  CIL equivalent for, 582–583
description, 136–140
  parallel, 819–820
for statements, 109, 137–140
foreach loops, class collections
  with arrays,
code example, 586
  with IEnumerable interface, 587
  with IEnumerable interface, 583–585
modifying collections during, 587–588
forEach() method, 806
foreach statement, 109, 140–142
Formal declaration, methods. See
     Methods, declaring.
     Formal parameter declaration, 171–172
     Formal parameter list, 172
     Format items, 21
     Format() method, 51
     Format strings, 21
     FormatMessage() method, 855
Formatting
     numbers as hexadecimal, 43
     with string interpolation, 20
     strings, 54
Forward slash (/) division operator
     description, 91–92
     overloading, 397–399
     precedence, 92
Forward slash (/) in XML, 25
Fragile base class, 307–311
Framework Class Library (FCL), 893, 896
Frameworks, definition, 26
from clause, 623–624, 638–639
FromCurrentSynchronizationContext() method, 793
Full outer join, definition, 604
Func delegates, 524–525
Function pointers, mapping to delegates, 861–862
Functions
     global, C++ vs. C#, 266
     pure virtual, 317

G
Garbage collection
     Collect() method, 419
     introduction, 418
     managed execution, 28
     in .NET, 418–419
     resource cleanup, 426–427
     root references, 418
     strong references, 420–421
     weak references, 420–421
Garbage collector, definition, 223
GC.ReRegisterFinalize() method, 429
General catch blocks, 211–212, 440–442
General purpose delegates, 524–525
Generic classes
     declaring, 464
     type parameters, 464
undo, with a generic Stack class, 461
Generic delegates, events, 564–566
Generic internals
     CIL representation, 498–500
     instantiating based on reference types, 500–501
     instantiating based on value types, 500–501
     introduction, 498
Generic methods
     casting inside, 490–491
     constraints, 481–482
     constraints on type parameters, 481–482
     guidelines, 491
     introduction, 486–487
     specifying constraints, 489–490
     type inference, 487–489
Generic types
     arity (number of type parameters), 471–472
     benefits of, 464–465
     constraints. See Constraints on type
     parameters.
     constructors, declaring, 468–469
     Create() method, 471–472
     factory methods, 471–472
     finalizers, declaring, 468–469
     generic classes, 461
     guidelines, 473
     implementing, 467
     interfaces, description, 466–467
     interfaces, implementing multiple
     versions, 467–468
     introduction, 461–462
     multiple type parameters, 470–471
     nesting, 472–473
     overloading a type definition, 471–472
     parameterized types, 461
     specifying default values, 469–470
     structs, 466–467
     Tuple class, 471–472
     type parameter naming guidelines, 465–466
Generic types, generic classes
     declaring, 464
     type parameters, 464
undo, with a generic Stack class, 461
Generic types, reflection on
     classes, identifying support for
     generics, 692–693
     determining parameter types, 692
Generic types, reflection on (continued)
methods, identifying support for generics, 692–693
type parameters for generic classes or methods, 693–694
typeof operator, 692
Generics, C# without generics
multiple undo operations, 456–459
nullable value types, 459–461
System.Collections.Stack class, 456–459
type safety, 459
using value types, 459
get keyword, 240
GetCurrentProcess() method, 851
GetCustomAttributes() method, 701–702
GetDynamicMemberNames() method, 729
GetEnumerator() method, 588, 669
GetFiles() method, 163, 275, 607, 625, 632
GetFirstName() method, 240
GetFullName() method, 170–174
GetGenericArguments() method, 693–694
GetInvocationList() method, 557–558
GetLastError API, 855
GetLastObjectName() method, 240
GetLength() method, 83–84
GetName() method, 227–234
GetProperties() method, 685–686
GetResponse() method, 776
GetResponseAsync() method, 779
GetReverseEnumerator() method, 680–681
GetSetting() method, 339–341
GetSummary() method, 317–320
GetSwitches() method, 702–704
GetUserInput() method, 170, 172, 182
GetValue() method, 691
GhostDoc tool, 418
Global methods, C++ vs. C#, 171
Global variables and functions, C++ vs. C#, 266
goto statement, 110, 150–152
Greater than, equal sign (>=), greater than or equal operator, 120, 396–397
Greater than sign (>), greater than operator, 120, 396–397
Greater than signs, equal (>=) shift right assignment operator, 130
Greater than signs (>>), shift right operator, 130, 397–399
GreaterThan() method, 512
group by clause, 635–637
group clause, 623
GroupBy() method, 610–611
Grouping query results, 634–637
GroupJoin() method, 611–614
Guest computer, definition, 872
.h file, C++ vs. C#, 174
Handle() method, 764, 780
Hardcoding values, 40–42
Hash tables
balancing, 385–387
dictionary classes, 656
Header files, C++ vs. C#, 174
Heaps, 62–63
Heater objects, 544–546
HelloWorld program
CIL output, 31–33
getting started, 2–3
Hexadecimal numbers
formatting numbers as, 43
notation, 42–43
specifying as values, 43
HideScheduler enum, 759
Hill climbing, 802–803
Hot tasks, 752
Hungarian notation, 7
Hyper-Threading, definition, 734
Hyphens (-), in identifier names, 7
IAngle.MoveTo interface, 367
IAsyncAction<T> interface, 876
ICloseable, 876
Icollection<T> interface, 646
IComparable interface, 649–650
IComparable<string>.CompareTo() method, 649–650
IComparable<T>, 477, 649–650
IComparer<T>, 649–651
Id property, 755
Identifiers
camelCase, 7
casing formats, 6–7
definition, 6
guidelines for, 7
keywords as, 8
naming conventions, 6–7
PascalCase, 6–7
syntax, 6–7
IDictionary<TKey, TValue>, 644–646, 653–657
IDisposable interface
cleaning up after iterating, 586
resource cleanup, 424–425, 426–427
Task support for, 774
in WinRT. See ICloseable.
IDistributedSettingsProvider interface, 346
IEnumerable interface
class diagram, 584
CopyTo() method, 646
Count() method, 646
customizing. See Iterators.
foreach loops, class collections, 587
.NET class collections, 582
IEnumerable<T> interface
class diagram, 584
customizing. See Iterators.
foreach loops, 583–585
.NET class collections, 582
in WinRT. See IIterable<T>.
IEnumerator<T> interface, customizing. See Iterators.
IEqualityComparer<T> interface,
658–659
#if preprocessor directive, 153, 154
if statements, 108, 111, 114, 150
if/else statements, examples, 111, 113
IFileCompression interface, 326–327
IFormattable interface, 362, 370, 479, 483–485
IIterable<T>, 876
IL. See CIL (Common Intermediate Language).
IL Disassembler. See ILDASM.
ILDASM, 30–31
IListable interface, 327–335, 342–343
IList<T>, 644–646
ILMerge.exe utility, 890
Immutability
delegates, 513
strings, 56–57
value types, 357
Immutable strings, syntax, 17
Implicit conversion, 65, 68–69
Implicit deterministic resource cleanup,
C++ vs. C#, 224
Implicit implementation interfaces, 334, 336–337
Implicit local variables, anonymous types, 572–576
Implicit member implementation, interfaces, 335–336
Implicit nondeterministic resource cleanup, C++ vs. C#, 224
Implicit overriding, Java vs. C#, 304
Implicitly typed local variables, 60–61
Importing types from namespaces. See using directive.
Imports directive, 176
in modifier, 495–497
In type parameter, 526
Increment() method, 829–830
Increment/decrement operators (++, --)
C++ vs. C#, 105
decrement, in a loop, 102–105
description, 101–102
lock statement, 105–106
postfix increment operator, 104–105
post-increment operator, 103
prefix increment operator, 104–105
pre-increment operator, 103–104
race conditions, 105–106
thread safety, 105–106
Indentation, flow control statement, 115
Indexers
defining, 665–666
specifying, 644–646
IndexOf() method, 651–652
Inextensible classes, 273–275
Infinite recursion error, 194
Information hiding. See Encapsulation.
Inheritance
base classes, 302
base type, 220
chaining, 292–293
child type, 220
constraint limitations, 482
constraints on type parameters, 480–482
vs. contracts, 340–341
derived types, 220–221
extension methods, 299
interfaces, 338–341
introduction, 219–220
“is a kind of” relationships, 290
multiple, C++ vs. C#, 299
Inheritance (continued)
   multiple, simulating, 299–301
   parent type, 220
   private members, 296–297
   purpose of, 290
   single, 299–301
   specializing types, 221
   subtypes, 220
   super types, 220
   value types, 361–362
Initialize() method, 244, 263
Initializing attributes with a constructor, 701–705
Inner classes, Java, 283
Inner join, 604, 607–610
InnerExceptions property, 558, 764, 780, 804
Insert method, 58
Inserting
   elements in dictionary classes, 654
   new elements in collections, 651–652
Instance fields. See also Static, fields.
   accessing, 226–227
   declaring, 225–226
   definition, 225
Instance methods
   adding to a class, 276
   definition, 51
   introduction, 227–228
Instantiating
   arrays, 74–76
   classes, 221–224
   delegates, 510–512
   interfaces, 334
Instantiation, 10, 222–223
int type, 14, 36, 202–203
Integer literals, determining type of, 41–42
Integers, type for, 36–37
Integral types, 96
Interface type constraints, 476–477
Interfaces
   vs. abstract classes, 325–326, 338
   aggregation, 343–344
   vs. attributes, 349
   vs. classes, 347–348
   contracts vs. inheritance, 340–341
   converting between interfaces and implementing types, 338
   data, 327
   defining, 327
   deriving one from another, 338-341, 346-347
   extension methods, 341-343
   inheritance, 338-341
   instantiating, 334
   introduction, 326–327
   method declarations in, 327
   naming conventions, 327
   polymorphism, 327–332
   post-release changes, 346
   purpose of, 326–327
   value types, 361–362
   versioning, 346-347
Interfaces, generic types
   description, 466–467
   finalizers, declaring, 468–469
   implementing multiple versions, 467-468
Interfaces, implementing and using
   accessing methods by name, 336
   code example, 328–332, 332–333
   explicit implementation, 334, 336-337
   explicit member implementation,
      334-335
   guidelines, 336-337
   implicit implementation, 334, 336-337
   implicit member implementation,
      335-336
   overview, 327
Interfaces, multiple inheritance
   aggregation, 343-344
   diagramming, 345
   implementing, 341, 343-344
   working around single inheritance,
      343-344
   internal access modifiers on type declarations, 407-408
   internal accessibility modifier, 409
Interpolating strings, 50-53
Intersect() method, 618
into keyword, 637
InvalidAddressException, 446
InvalidCastException, 366
Invoke() method, 548–549
I/O-bound latency, definition, 732
IObsolete interface, 349
IOrderedEnumerable<T> interface, 602
IProducerConsumerCollection<T>, 840–841
IQueryable<T> interface, 619
IReadableSettingsProvider interface, 338-341
IReadOnlyPair<T> interface, 492–494
“Is a kind of” relationships, 290
“Is a” relationships, 293–294, 317
Is operator, verifying underlying types, 321–322
IsAlive property, 744
IsBackground property, 743
IsCancellationRequested property, monitoring, 770–771
IsCompleted property, 754
ISerializable interface, 449, 715–716
ISettingsProvider interface, 339–341, 346–347
IsInvalid, 858
IsKeyword() method, 628
Items property, 468
Iterating over
class collections using
IEnumerable<T>. See
defining, 668
class collections using while(), 584
properties of objects in a collection. See
Reflection.
Iterators
canceling iteration, 677–678
contextual keywords, 679–680
creating a language construct from, 668
defining, 668
functional description, 678–679
guidelines, 676
multiple in one class, 680–681
nested, 676
origin of, 667–668
recursive, 676
reserved keywords, 679–680
returning values from, 669–671, 673–674
state, 671–673
struct vs. class, 677
syntax, 668–669
yield break statement, 677–678
yield return statement, 670–671,
673–676, 681
ITrace interface, 337
IWriteableSettingsProvider interface, 341
J
Jagged arrays, 78, 79, 81
Java, similarities to C#, 2
Java vs. C#
classes spanning multiple files, 4
exception specifiers, 440
filename matching class name, 4
generics, 500–501
implicit overriding, 304
importing namespaces with
wildcards, 176
inner classes, 283
main() method, 10
nested classes, 283
partial class, 4
virtual methods by default, 303
JavaScript vs. C#
var keyword, 575
Variant, 575
void*, 575
JIT (just-in-time) compiler, 881
Jitting, 881. See also Compilers; Compiling;
Just-in-time compilation.
Join() method, 607–610, 743–744
Join operations. See Standard query
operators, join operations.
Jump statements
break statement, 146–147
continue statement, 148–150
goto statement, 150–152
if statement, 150
Just-in-time compilation, 26
K
KeyNotFoundException, 656
Keywords. See also specific keywords.
contextual, 6
definition, 4
as identifiers, 8
incompatibilities, 6
list of, 5
placement, 4
reserved, 6, 8
syntax, 4–6, 8
Kill() method, 789
L
Lambda calculus, 524
Lambda expressions
=> (equal sign, greater than) lambda
operator, 517, 524
captured variables, 528–530
capturing loop variables, 531–533
closed over variables, 528–530
closures, 531
as data, 534–535
definition, 516
Lambda expressions (continued)
- explicitly declared parameter types, 518
- expression lambdas, 520
- `GetTypeInfo()` method, 520
- guidelines, 518
- internals, 527–528
- lifetime of captured variables, 530
- name origin, 523–524
- notes and examples, 521–522
- outer variable CIL implementation, 530–531
- outer variables, 528–530
- predicate, definition, 520, 591
- returning a `bool`, 520
- sequence of operations, 599
- statement lambdas, 517–519
- `typeof()` operator, 520

Lambdas, asynchronous, 786–787

Language contrast. See specific languages.

Language interoperability, 27

LastIndexOf() method, 651–652

Latency
- asynchronous high-latency operations with the TPL, 777–781
- definition, 732
- synchronous high-latency operations, 775–777

Lazy loading, 429–431

LazyCancellation enum, 759

Left outer join, definition, 604

Left-associative operators, 93

Length
- arrays, 80–81
- strings, 55–56

Length member, 55–56, 80

Less than, equal sign (`<=`) less than or equal operator, 120, 396–397

Less than sign (`<`) less than operator, 120, 396–397

Less than signs, equal (`<==`) shift left assignment operator, 130

Less than signs (`<<`) shift left operator, 130, 397–399

let clause, 633–634

Libraries
- BCL (Base Class Library), 893
- class, 404–405
- definition, 3–4, 404
- FCL (Framework Class Library), 893
- file extension, 4
- PCLs (portable class libraries), 406

TPL (Task Parallel Library), 733

Library implementation. See CLS
(Common Language Specification).

Lifetime of captured variables in lambda expressions, 530

#line preprocessor directive, 153, 157–158

Line-based statements, Visual Basic vs. C#, 11

LinkedListNode<T>, 663

LinkedList<T>, 663

LINQ queries
- building with expression trees, 537–538
- with query expressions. See Query expressions with LINQ.
- running in parallel, 594–595

Liskov, Barbara, 667

List collections, 646–649

Lists. See also Collections.
- vs. dictionaries, 644–645
- indexers, defining, 665–666
- sorting, 649–650

List<T>
- covariance, 492
- description, 647–649

List<T>.Sort() method, 649–650

Literal values
- case sensitivity, suffixes, 42
- definition, 39
- exponential notation, 42
- specifying, 39–40
- strings, 48

Loading
- files, 232–235
- lazy, 429–431

Local variable scope, C++ vs. C#, 118

Local variables
- assigning values to, 14, 16–17
- casing, 15
- changing values, 16–17
- declaring, 13–14
- guidelines for naming, 15
- implicitly typed, 60–61
- naming conventions, 15, 172
- unsynchronized, 820

lock keyword, 823–825

lock objects, 825

Lock performance, 825

lock statement, 366–368, 740

Lock synchronization, 823–825

Locking
- guidelines, 827
on this, typeof, and string, 826–827
threading problems, 740
LockTaken parameter, 823–825
Logical operators, 131–133. See also Boolean expressions, logical operators.
long type, 36
LongRunning enum, 758
Long-running tasks, 773–774
Loop variables, 137, 531–533
LoopExpression, 535
Loops
break statement, 110
continue statements, 109
with decrement operator, 102–105
do while, 108, 134–135
escaping, 146–147
for, 109, 137–140
foreach, 109, 140–142
goto, 110
guidelines, 139
if, 108, 111, 114
if/else, examples, 111, 113
iterations, definition, 136
nested if, 112–113
switch, 110
while, 108, 134–135, 136–137
Loops, jump statements
break, 146–147
continue, 148–150
goto, 150–152
if, 150
lpflOldProtect, 853
main() method, Java vs. C#, 10
__ makeref keyword, 8
MakeValue() method, 480
Managed code, definition, 26
Managed execution
BCL (Base Class Library), 27, 28
CIL (Common Intermediate Language), 26–28
CLI (Common Language Infrastructure) specification, 26–27
CLS (Common Language Specification), 27
code access security, 27
CTS (Common Type System), 27
definition, 26
execution time, 27
garbage collection, 28
just-in-time compilation, 26
language interoperability, 27
managed code, definition, 26
native code, definition, 26
platform portability, 28
runtime, definition, 26, 27
type safety, 27
unmanaged code, definition, 26
VES (Virtual Execution System), 26
Many-to-many relationships, definition, 604
Masks, 132
Max() method, 273–275, 618
MaxDegreeOfParallelism property, 807
Max<T> method, 487–489
mcs.exe compiler, 3
Me keyword, 230
Member invocation, reflection, 687–692
Member names, retrieving, 729
Member variables, 225
MemberInfo, 691–692
Memory, deallocating, 423. See also Garbage collector.
Metadata
about assemblies, adding, 697–698
within an assembly, examining. See Reflection.
CIL (Common Intermediate Language), 894–895
definition, 25
for types, accessing with System.Type, 685. See also Reflection.
XML, 25
Metadata tables, setting bits or fields in. See Pseudoattribute.
Method calls
avoiding boxing, 369–370
during construction, C++ vs. C#, 307
as ref or out parameter values, 252
vs. statements, 169
translating query expressions to, 640–641
Method group conversion, delegates, 512
Method names
as arguments, delegates, 511–512
calling, 167
Method resolution, 201–202
Method returns, multicast delegates, 558
MethodCallExpression, 535
MethodImplAttribute, 827
MethodImplOptions.Synchronized() method, 827
Methods. See also specific methods.
accessing by name, on interfaces, 336
class association, 163
declaring in interfaces, 327
definition, 9, 162–163
derived from System.Object, 320–321
global, C++ vs. C#, 171
guidelines for naming, 163
identifying support for generics, 692–693
instance, 227–228
naming conventions, 163
operational polymorphism, 195
overloading, 194–197, 199
overriding, 304
partial, 285–288
return type declaration, 172–174
return values, 168–169
syntax, 9–11
uniqueness, 195
void, 173–174
Methods, calling
applicable calls, 201
arguments, 163, 167–168
caller, 163
compatible calls, 201
method call example, 164
method name, 167
method resolution, 201–202
method return values, 168–169
named arguments, 199
namespaces, 164–166. See also specific namespaces.
recursively. See Recursion.
return values, 163
scope, 167
statements vs. method calls, 169
type name qualifier, 166–167
Methods, declaring
equivalent, 169–170
expression bodied methods, 174
formal parameter declaration, 171–172
formal parameter list, 172
method return type declaration, 172–174
refactoring, 171
return statement, example, 173
type parameter list, 172
void as a return type, 173–174
Methods, extension
derivation, 299
on interfaces, 341–343
overview, 275–277
Microsoft IL (MSIL). See CIL (Common Intermediate Language).
Microsoft Silverlight compiler, 880
Miller, J., 26
Min() method, 273–275, 618
Min(<T>) method, 487–489
Minus sign (-)
arithmetic subtraction operator, 91–92
delegate operator, 551–552
precedence, 92
subtraction operator, overloading, 397–399
unary operator, 90–91
Minus sign, equal (=)
binary/assignment operator, 399
delegate operator, 550–552
Minus signs (--) decrement operator
C++ vs. C#, 105
decrement, in a loop, 102–105
description, 101–102
guidelines, 105
lock statement, 105–106
postfix increment operator, 104–105
post-increment operator, 103
prefix increment operator, 104–105
pre-increment operator, 103–104
race conditions, 105–106
thread safety, 105–106
Modules, referencing, 405
Monads, definition, 591
Monitor, 821–823
Mono compiler, 3, 880
Montoya, Inigo, 2
Move() method, 276, 357
MoveNext() method, 672–673
MSIL (Microsoft IL). See CIL (Common Intermediate Language).
MTA (Multithreaded Apartment), 846
Multicast delegates
   adding methods to, 554
   chaining, 555
   definition, 543, 547
   error handling, 554–556
   internals, 554
   new delegate instances, 550
   passing by reference, 558
   removing delegates from a chain, 550–552
Multidimensional arrays, 74, 77–79
Multiple inheritance
   C++ vs. C#, 299
   simulating, 299–301
Multiple inheritance, interfaces
   aggregation, 343–344
   diagramming, 345
   implementing, 341, 343–344
   working around single inheritance, 343–344
Multiple Main() methods, disambiguating, 182
Multiplication with bit shifting, 130
Multithreaded Apartment (MTA), 846
Multithreaded programming
   complexities, 749–750
   single-threaded programs,
      definition, 734
   TAP (Task-based Asynchronous Pattern), 733
   task, definition, 735
   thread, definition, 734
   thread pool, definition, 735
   thread safe code, definition, 734
   threading model, definition, 734
   time slice, definition, 736
   time slicing, definition, 736
   TPL (Task Parallel Library), 733
Multithreading, asynchronous tasks
   antecedent tasks, 757
   associating data with tasks, 755
   asynchronous continuations, 756–762
   chaining tasks, 757
   cold tasks, 752
   composing large tasks from smaller one, 756–758
   continuation tasks, 757
   control flow, 755
   creating threads and tasks, 750–751
   hot tasks, 752
   Id property, 755
   introduction, 751–755
   invoking, 751–753
   multithreaded programming
      complexities, 749–750
   observing unhandled exceptions, 764–765
   polling a Task<T>, 753–754
   registering for notification of task behavior, 760–761
   registering for unhandled exceptions, 766–768
   synchronous delegates, 751
   task continuation, 755–762
   task identification, 755
   task scheduler, 750–751
   task status, getting, 754
   tasks, definition, 750
   tasks vs. delegates, 751
   unhandled exception handling with
      AggregateException, 762–765
   unhandled exceptions on a thread, 765–768
Multithreading, canceling tasks
   cooperative cancellation, definition, 769
   disposable tasks, 774
   long-running tasks, 773–774
   obtaining a task, 772–773
Multithreading, guidelines
aborting threads, 747
long-running tasks, 773–774
parallel loops, 801
performance, 737, 740
thread pooling, 749
Thread.Sleep() method, 745
unhandled exceptions, 768
Multithreading, parallel loop iterations
breaking, 808
canceling, 805–806
exception handling with
AggregateException, 803–804
hill climbing, 802–803
introduction, 798–802
options, 806–807
TPL performance tuning, 802–803
work stealing, 802–803
Multithreading, performance
context switching, 737
overview, 736–737
switching overhead, 737
time slicing costs, 737
Multithreading, PLINQ queries
canceling, 811–813
introduction, 809–811
Multithreading, task-based asynchronous pattern
with async and await, 781–786
async and await with the Windows UI, 795–798
async keyword, purpose of, 786
asynchronous high-latency operations with the TPL, 777–781
asynchronous lambdas, 786–787
await keyword, 791–792
await operators, 797–798
awaiting non-Task<>T values, 791–792
control flow misconceptions, 784
control flow within tasks, 784–786
custom asynchronous methods, 787–791
handling exceptions, 779–780
progress update, 789–791
synchronization context, 793–795
synchronous high-latency operations, 775–777
task drawbacks, overview, 775
task schedulers, 793–795
Multithreading, threading problems
atomic operations, 738
complex memory models, 739
deadlocks, 740
lock statement, 740
locking leading to deadlocks, 740
race conditions, 738
Multithreading, with System.Threading.
Thread
Abort() method, 745–746
aborting threads, 745–746
asynchronous delays, 745
asynchronous operations, 741–743
await operator, 745
checking threads for life, 744
foreground threads vs. background, 743
IsAlive property, 744
IsBackground property, 743
Join() method, 743–744
Priority property, 743
putting threads to sleep, 744
reprioritizing threads, 743
TaskDelay() method, 745
thread management, 743–744
thread pooling, 747–749
ThreadAbortException, 745–746
Thread.Sleep() method, putting threads to sleep, 744
ThreadState property, 744
waiting for threads, 743

\n, newline character, 46, 48
Name property, 249–250, 292, 297, 302–303
Named arguments, calling methods, 199
Named parameters, attributes, 707
nameof operator
properties, 246–247
throwing exceptions, 435, 436
Namespaces
alias qualifier, 412–413
aliasing, 179–180. See also using directive.
calling methods, 164–166
in the CLR (Common Language Runtime), 410
common, list of, 165–166
defining, 409–410
definition, 164, 175
dropping. See using directive.
eliminating. See using directive.
extern alias directive, 413
Index

guidelines, 166, 412
importing types from. See using directive.
introduction, 409–410
namespaces alias qualifier, 412–413
naming conventions, 166, 410
nested, 176, 411
Naming conventions
class definition, 8
identifiers, 6–7
interface guidelines for, 327
local variables, 15
methods, 163
namespaces, 166, 410
parameters, 172, 200
properties, 243
type parameter, 465–466
Native code, definition, 26
NDoc tool, 418
Nested
classes, 281–283
delegates, 510
generic types, 472–473
if statements, 112–113
iterators, 676
namespaces, 176, 411
types, 283
using directives, 177–178
.NET
definition, 895
garbage collection, 418–419, 884–885
.NET Compact Framework compiler, 880
.NET Micro Framework compiler, 880
.NET versions, mapped to C# releases, 29
New line, starting
\n (newline character), 46, 48, 55
strings, 46, 48
verbatim string literals, 49
WriteLine() method, 55
new modifier, 307–311
new operator
constructors, 256
value types, 359–360
NewExpression, 535
NextId initialization, 271–273
Non-nullable value types, 478–479
No-oping a call, 710–711
Normalized data, 607–608
NOT operator, 122
NotImplementedException, 210
NotOnCanceled enum, 759
NotOnFaulted enum, 758
NotOnRanToCompletion enum, 758
nowarn option, 157
Nowarn:<warn list> option, 157
null, checking for
empty arrays or collections, 666–667
guidelines, 549
multicast delegates, 548–549
null type
assigning value types to, 64–65
description, 58–59
use for, 58
NullReferenceException
invoking delegates, 562
throwing exceptions, 434, 436
Numbers, formatting as hexadecimal, 43
Numeric conversion, exception handling, 215–216
O
Object graphs, expression trees as, 535–536
Object initializers
calling, 257–259
constructors, 257–259
declaration, 257
object members, overriding, 388–395
Object-oriented programming, definition, 218–219
Objects. See also Constructors.
associations, 269
CTS types, 892
definition, 222–223
destroying, 259
identity vs. equal object values, 388–392
instantiation, 222–223
Observer pattern. See Coding the observer pattern with multicast delegates.
Obsolete APIs. See Deprecated APIs.
Obtaining a task, 772–773
OfType<T>() method, 618
One-to-many relationships, 604, 611–613
OnFirstNameChanging() method, 287
OnLastNameChanging() method, 287
OnlyOnCanceled enum, 758
OnlyOnFaulted enum, 759
OnlyOnRanToCompletion enum, 759
OnTemperatureChange event, 566
OnTemperatureChange() method, 567–568
OnTemperatureChanged() method, 545–546
Operands, 90
Operational polymorphism, 195
Index

OperationCanceledException, 772, 806, 811–813
Operator constraints, constraint limitations, 482–483
Operator keyword, 402
Operator order of precedence, C++ vs. C#, 105
Operator-only statements, C++ vs. C#, 91
Operators. See also specific operators:
  arithmetic binary (+, -, *, /, %), 91–92, 95
  associativity, 92, 93–94
  characters in arithmetic operations, 96
  comparison, 396–397
  compound assignment (+=, -=, *=, /=, %=), 100–101
  const keyword, 106-107
  constant expressions, 106–107
  constant locals, 106–107
  definition, 89
  left-associative, 93
  operands, 90
  operator-only statements, C++ vs. C#, 91
  order of operations, 92, 94
  plus and minus unary (+, -, ), 90–91
  precedence, 92, 93–94, 160
  results, 90
  right-associative, 93
Operators, increment/decrement (++, --) C++ vs. C#, 105
  decrement, in a loop, 102–105
  description, 101–102
  guidelines, 105
  lock statement, 105-106
  postfix increment operator, 104-105
  post-increment operator, 103
  prefix increment operator, 104-105
  pre-increment operator, 103-104
  race conditions, 105-106
  thread safety, 105-106
Optional parameters, 197–198
OR criteria, constraint limitations, 483
Order of operations, 92, 94. See also Precedence.
orderby clause, 632–633
OrderBy() method, 601–603
OrderByDescending() method, 602
out, passing parameters, 186–188
out modifier, 492–494
out parameter, properties as values, 252
out property, 700
out vs. pointers, P/Invoke, 853–854
Outer joins, 613–616
Outer variables, lambda expressions, 528–530
OutOfMemoryException, 435, 444
Output, passing parameters, 186–188
Overloading. See also Overriding.
  == (equal signs) equality operator, on value types, 362
  != (exclamation point, equal sign) inequality operator, on value types, 362
  cast operator, 402
  constructors, 259-261
  equality operators on value types, 362
  Equals() method, 362
  methods, 194–197
  type definitions, 471–472
Overloading, object members
  Equals() equality operator, 388-395
  GetHashCode() method, 385–387
  ToString() method, 384–385
Overloading, operators
  binary operators, 397–399
  binary operators combined with assignment operators, 397–399
  cast operator, 402
  conditional logical operators, 400
  conversion operators, 401, 403
  unary operators, 400–401
override keyword, 304, 313
Overriding. See also Overloading.
  abstract members, 317
  base classes. See Base classes, overriding.
  base classes, virtual methods, 302–307
  implicit, C++ vs. C#, 304
  methods, 304
  properties, 303
P
PairInitializer<T> interface, 497
Pair<T>, 492
Palindromes, 84–86
Parallel LINQ (PLINQ) queries, multithreading
  canceling, 811–813
  introduction, 809–811
Parallel loop iterations. See Multithreading, parallel loop iterations.
Parallel programming, definition, 736
Parallel.ForEach() loops, 807
Parallel.ForEach() method, 808
Parallel.ForEach() loops, 807
Parallel.ForEach() method, 808
ParallelOptions parameter, 807
ParallelQuery<T>, 810, 813
Parameter arrays, 189–191
Parameter types
determining, 692
explicitly declared, 518
P/Invoke, 851–853
ParameterExpression, 535
Parameterized types, 461
Parameters
calling methods, 163, 167–168
guidelines, 172, 199–200
on the Main() method, 180–183
matching caller variables with
parameter names, 184
method overloads, 199
names, generating. See Nameof
operator.
names, obtaining, 696
naming conventions, 172, 200
optional, 197–198
reference types vs. value types, 184–185
specifying by name, example, 199–200
Parameters, passing
out, 186–188
output, 186–188
ref type, 185–186
by reference, 185–186
by value, 183–184
params keyword, 189–191
Parent type, 220
Parentheses ( )
for code readability, 93–94
grouping operands and operators, 93–94
guidelines, 94
Parse() method, 69–70, 215–216, 376, 709
Partial classes, 4, 284–285
Partial methods, C++ vs. C#, 174
PascalCase, 6–7
Passing
anonymous methods, 522–523
command-line arguments to Main()
method, 180–183
delegates with expression lambdas, 520
by reference, multicast delegates, 558
Passing, parameters
out, 186–188
output, 186–188
ref type, 185–186
by reference, 185–186
by value, 183–184
PCLs (portable class libraries), 406
Peek() method, 661–662
Percent sign, equal (%=) binary /
assignment operator, 399
Percent sign (%) modulo, 91–92, 397–399
Performance
boxing, 365
effects of boxing, 365
locks, 825
multithreading, 736–737, 740
runtime, 887–888
TPL (Task Parallel Library), 802–803
Periods (....), download progress
indicator, 779
PiCalculator.Calculate() method, 753
PingButton_Click() method, 796
Ping.Send() method, 796
P/Invoke. See also WinRT.
allocating virtual memory, 852
calling external functions, 858–860
declaring external functions, 850–851
declaring types from unmanaged
structs, 854–855
description, 850
error handling, 855–857
function pointers map to delegates,
861–862
guidelines, 857, 862
out vs. pointers, 853–854
parameter types, 851–853
ref vs. pointers, 853–854
SafeHandle, 857–858
sequential layout, 854–855
Win32 error handling, 855–857
wrappers for API calls, 860–861
Platform interoperability. See P/Invoke;
Unsafe code; WinRT.
Platform portability, managed
execution, 28
PLINQ (parallel LINQ) queries,
multithreading
canceling, 811–813
introduction, 809–811
Plus sign (+)
addition operator, overloading, 397–399
arithmetic binary operator, 91–92
with char type data, 96
concatenating strings, 95
delegate operator, 551–552
Plus sign (+) (continued)
  determining distance between two
  characters, 96
  with non-numeric operands, 95
  precedence, 92
  unary operator, 90-91
Plus sign, equal (+=)
  binary/assignment operator, 399
  delegate operator, 550-552
Plus signs (++) increment operator
  C++ vs. C#, 105
  decrement, in a loop, 102-105
  description, 101-102
  guidelines, 105
  lock statement, 105-106
  postfix increment operator, 104-105
  post-increment operator, 103
  prefix increment operator, 104-105
  pre-increment operator, 103-104
  race conditions, 105-106
  thread safety, 105-106
Pointers and addresses
  accessing members of a referent
  type, 871
  allocating data on the call stack,
  868-869
  assigning pointers, 866-869
  dereferencing pointers, 869-871
  fixing (pinning) data, 866-868
  pointer declaration, 864-869
  referent types, 864
  unmanaged types, 865
  unsafe code, 862-864
Polling a Task<T>, 753-754
Polymorphism. See also Inheritance;
  Interfaces.
  abstract classes, 318-320
  description, 221
  interfaces, 327-332
Pop() method, 456-459, 661-662
Portable class libraries (PCLs), 406
#pragma preprocessor directive, 153, 156-157
Precedence, 92, 93-94, 160. See also Order
  of operations.
Precision
  binary floating-point types, 97
  double type, 97
  float type, 97-100
Predefined attributes, 709
Predefined types, 35-36
Predicates
  definition, 520, 591
  filtering class collections, 591
  lambda expressions, 520, 631
PreferFairness enum, 758
Preprocessing, C++ vs. C#, 152
Preprocessor directives. See also Control
  flow; Flow control.
  as comments, 154
  as debugging tool, 154
  defining preprocessor symbols, 155
  disabling/restoring warning
  messages, 156-157
  emitting errors and warnings, 155-156
  excluding/including code, 154
  handling differences among
  platforms, 154
  specifying line numbers, 157
  summary of, 152. See also specific
  directives.
  visual code editors, 158-159
Preprocessor symbols, defining with
  preprocessor directives, 155
The Princess Bride, 2
Print() method, 318
Priority property, 743
private access modifier, 235-237, 296-297
private accessibility modifier, 409
Private fields, 237-238
Private keyword, 237
Private members
  accessing, 296-297
  definition, 236
  inheritance, 296-297
Process, definition, 734
Process.Kill() method, 789
Processor-bound latency, definition, 732
Program
  accessing fields, 226-227
  accessing static fields, 267-268
  code example, 398-399, 405
  defining properties, 240
Programming, object-oriented definition,
  218-219
Programming with dynamic objects
  dynamic binding, 724-725
  dynamic directive, 721
  dynamic member invocation, 722
  dynamic principles and behaviors,
  721-723
  dynamic System.Object, 723-724
implementing a custom dynamic object, 726–728
introduction, 719
invoking reflection with dynamic, 719–720
reflection, support for extension methods, 723
retrieving member names, 729
signature verification, 722
vs. static compilation, 725–726
type conversion, 721–722
type safety, 720–721, 726
Progress update display, 789–791
Projecting collections
definition, 622
FileInfo collections, 633–634
with query expressions, 624–627
with Select(), 592–594
Properties
automatically implemented, 240–242, 248, 254, 273
automatically implemented, read-only, 280
declaring, 238–240
decorating with attributes, 696–697
definition, 238
guidelines, 242–244, 248, 258
internal CIL code, 253–254
introduction, 237–238
nameof operator, 246–247
naming conventions, 243
overriding, 303
read-only, 247–248
read-only automatically implemented, 280
as ref or out parameter values, 252
static, 273
validation, 244–246
as virtual fields, 249–250
write-only, 247–248
protected access modifier, 297–298
protected accessibility modifier, 409
protected internal accessibility modifier, 409
protected internal type modifier, 408
Protected members, accessing, 297–298
Pseudoattributes, 719
public access modifier, 235–237, 407–409
Publishing code, checking for null, 548
Pulse() method, 823
Pure virtual functions, C++ vs. C#, 317
Push() method, 456–459, 661–662
Q
Quantum, definition, 736
Query continuation clauses, 637–638
Query expressions with LINQ
code example, 622–623
continuation clauses, 637–638
deferred execution, 627–631
definition, 621
discarding duplicate members, 639–640
filtering collections, 631–632
flattening a sequence of sequences, 638–639
from clause, 623–624
group by clause, 635–637
group clause, 623
grouping query results, 634–637
into keyword, 637
introduction, 622–624
let clause, 633–634
projecting collections, 624–627
range variables, 622–623
returning distinct members, 639–640
select clause, 623–624
sorting collections, 632–633
translating to method calls, 640–641
where clause, 623–624
Query operators. See Standard query operators.
Question mark, colon (?:) conditional operator, 123–124
Question mark, dot (.) null-conditional operator, 125–128
Question mark (?) nullable modifier, 64–65, 459
Question marks (??) null-coalescing operator, 124–125, 126
Queue collections, 662
Queue<T>, 662
R
Race conditions. See also Thread synchronization.
class collections, 595
threading problems, 738
Ragsdale, S., 26
Range variables, 622–623
Rank, arrays
declaring, 72
getting the size of, 83–84
Referent types
accessing members of, 871
definition, 864
Reflection
accessing metadata, 894-895
accessing using System.Type
class, 685
CIL (Common Intermediate
Language), 894-895
definition, 684
GetProperties() method, 685-686
getting an object’s public properties,
685-686
GetType() method, 685-686
invoking, with dynamic, 719-720
invoking with dynamic, 719-720
member invocation, 687-692
retrieving Type objects, 686-687
support for extension methods, 723
TryParse() method, 690-692
typeof() method, 686-687
uses for, 684
Reflection, on generic types
classes, identifying support for
generics, 692-693
determining parameter types, 692
methods, identifying support for
generics, 692-693
type parameters for generic classes or
methods, 693-694
typeof operator, 692
__ reftype keyword, 8
__ refvalue keyword, 8
#region preprocessor directive, 153,
158-159
Registering for
notification of task behavior, 760-761
unhandled exceptions, 766-768
Relational operators, 119-120
ReleaseHandle() method, 858
Remove() method
event internals, 568
removing delegates from chains, 552
removing dictionary elements, 656
System.Delegate, 649
System.Text.StringBuilder, 58
RemoveAt() method, 649
Remove_OnTemperatureChange() method,
567-568
Removing
deleges from a chain, 550-552
dictionary elements, 656

elements from collections, 649
Replace, 58
ReRegisterFinalize() method, 429
Reserved keywords, 6, 8, 679–680
Reset events, 836–839
Reset() method, 584
Resource cleanup. See also Garbage collection.

class collections, 587
exception propagation from constructors, 428
finalization, 421–427
finalization queue, 426
garbage collection, 426–427
guidelines, 428
with IDisposable, 424–425, 426–427
introduction, 421
invoking the using statement, 425–426
resurrecting objects, 429
Result property, 754
Results, 90
Resurrecting objects, 429
Rethrowing
existing exceptions, 438–439
wrapped exceptions, 449–453
Retrieving
attributes, 700–702
member names, 729
Type objects, reflection, 686–687
Return attributes, 698–699
return statement, 173
Return values
calling methods, 163
from iterators, 669–671, 673–674
from Main() method, 180–183
methods, 168–169
from the ReadLine() method, 18
Reverse() method, 85–86, 618
Reversing
arrays, 81–82
strings, 84–86
Right outer join, definition, 604
Right-associative operators, 93
Root references, 418
Round-trip formatting, 44–45
Run() method, 772–773
RunContinuationAsynchronously enum, 759
Running applications, 3
RunProcessAsync() method, 788–789

Runtime. See also VES (Virtual Execution System).
circumnavigation encapsulation and access modifiers, 885
CLR (Common Language Runtime), 896
code access security, 886
definition, 26, 27
garbage collection, 883–885
managed code, definition, 883
managed data, definition, 883
managed execution, definition, 883
performance, 887–888
platform portability, 886–887
reflection, 885
type checking, 885
type safety, 885
RuntimeBinderException, 722

S
SafeHandle, 857–858
Save() method, 231–232
sbyte type, 36
Scope
calling methods, 167
flow control statements, 116–118
Sealed classes, 301–302
sealed modifier, 311–312
Sealed types constraint limitations, 484
Sealing virtual members, 311–312
Search element not found, 651–652
Searching
arrays, 81–83, 651–652
collections, 651–653
lists, 651–652
select clause, 623–624
Select() method
projecting class collection data, 592–594
vs. SelectMany(), 615–616
SelectMany() method
calling, 614–616
creating outer joins, 613–616
vs. Select(), 615–616
Semaphore, 839–840
SemaphoreSlim, 839–840
SemaphoreSlim.WaitAsync() method, 840
Semicolon (;), ending statements, 6–7, 11
Send() method, 796
SendTaskAsync() method, 796
SequenceEquals() method, 618
Sequential invocation, multicast delegates, 552, 554
Serializable objects, 449
Serialization() method, 714
Serialization-related attributes, 713–714
Serializing
  business objects into a database. See Reflection.
documents, 716–718
set keyword, 240
Set() method, 836–837
SetName() method, 228–234
SetResult() method, 789
Setter methods. See Getter/setter methods.
Shared Source CLI compiler, 880
Short circuiting, with the null-coalescing operator, 124–125
short types, 36, 37
SignalAndWait() method, 835
Signature verification, 722
Simultaneous multithreading,
definition, 734
Single backslash character (\), escape sequence, 46
Single inheritance, 299–301, 343–344
Single-line comments, 24
Single-threaded programs, definition, 734
Slash, equal (/=) binary/assignment operator, 399
Sleep() method, 744–745
Smiley face, displaying, 47–48
Sort() method, 649–650
SortedDictionary<T>, 660–661
SortedList<T>, 660–661
Sorting
  arrays, 81–82, 82–83
  class collections, 601–603. See also Standard query operators, sorting.
collections, 649–651, 660–661
lists, 649–650
Sorting, collections. See also Standard query operators, sorting.
  by file size, 633–634
  by key, 660–661
  with query expressions, 632–633
  by value, 660–661
Specializing types, 221
SqlException() method, 448
Square brackets ([]), array declaration, 72–74
Stack
  allocating data on, 868–869
definition, 353
  unwinding, 182–183
  values, accessing, 661–662
Stack, 461
Stack collections, 661–662
Stackalloc data, 868
StackOverflowException, 210
Stack<T>, 500–501, 661
Standard query operators
  AsParallel(), 595
caching data, 600
Concat(), 618
counting elements with Count(), 595–596
defered execution, 597–598, 600–601
definition, 588
Distinct(), 618
filtering with Where(), 591–592, 597–598
guidelines, 596, 641
Intersect(), 618
OfType(), 618
projecting with Select(), 592–594
queryable extensions, 619
race conditions, 595
Reverse(), 618
running LINQ queries in parallel, 594–595
sample classes, 588–591
sequence diagram, 599
SequenceEquals(), 618
sorting, 601–603
System.Linq.Enumerable method calls, 616–617
table of, 618
Union(), 618
Standard query operators, join operations
  Cartesian products, 608
DefaultIfEmpty(), 613–614
full outer join, 604
grouping results with GroupBy(), 610–611
inner join, 604, 607–610
introduction, 603–604
left outer join, 604
many-to-many relationships, 604
normalized data, 607–608
one-to-many relationships, 604
one-to-many relationships, with
  GroupJoin(), 611–613
outer joins, with GroupJoin(), 613–614
outer joins, with SelectMany(), 613–616
right outer join, 604
Start() method, 305–306, 762–763
StartNew() method, 772–773
State
  iterators, 671–673
  sharing, class collections, 585
Statement delimiters, 11–12
Statement lambdas, 517–519
Statements
  vs. method calls, 169
  multiple, on the same line, 11–12, 13
  splitting across multiple lines, 11, 12
  syntax, 11–12
STAThreadAttribute, 846
Static
  classes, 273–275
  compilation vs. programming with dynamic objects, 725–726
  constructors, 271–272
  fields, 267–268. See also Instance fields; static keyword.
  methods, 51–52, 269–271
  properties, 273
static keyword, 266. See also Static, fields.
Status property, 754
Stop() method, 305–306, 808
Storage
  disk, 662
  files, 232–235
  reclaiming. See Finalizers; Garbage collection.
Store() method, 232–235
String interpolation, formatting with, 20
string keyword, 48, 826–827
string() method, 48
String methods
  instance methods, 51
  static methods, 51–52. See also using directive; using static directive.
string type, 48, 202–203
string.Format method, 51
string.join statement, 801
Strings
  “” (double quotes), coding string literals, 48
  @ (at sign), coding verbatim strings, 48
  $ (dollar sign), string interpolation, 48
  @@ (dollar sign, at sign), string interpolation, 50
  as arrays, 84–86
  changing, 56–57
concatenation at compile time, C++ language, vs. C#, 50
concatenation at compile time, vs. C++, 50
determining length of, 55–56
formatting, 54
having no value vs. empty, 59
immutability, 56–57
interpolation, 50–53
length, 55–56
Length member, 55–56
literals, 48
\n, newline character, 46, 48
read-only properties, 55
representing a binary display, 132
starting a new line, 46, 48
type for, 48
verbatim string literals, 49
void type, 59–60
Strong references, garbage collection, 420–421
struct keyword
  vs. class, 677
  constraints, 478–479
  declaring a struct, 356
StructLayoutAttribute, 854–855
Structs
  declaring, 356
  default value for, 360–361
  definition, 356
  finalizer support, 360
  generic types, 466–467
  initializing, 357–359
  referential identity, 360
Structural equality, delegates, 526–527
subscriber methods, 544–545
Subtypes, 220
Sum() method, 618
Super types, 220
SuppressFinalize() method, 426
Surrogate pairs, 46
Swap() method, 186
Swapping array data elements, 79
switch statements
  catching exceptions, 436
  code example, 142–146
  fall-through, C++ vs. C#, 145
  syntax, 110
Switching overhead, 737
Synchronization. See Thread synchronization.
Synchronization context, 793–795
Synchronization types. See Thread synchronization, synchronization types.
Synchronized() method, 827
Synchronizing
code, boxing, 366–368
local variables, 819–820
multiple threads with Monitor class, 821
Synchronous delegates, 751
Synchronous high-latency operations, 775–777
Syntax
class definition, 8–9
identifiers, 6–7
immutable strings, 17
keywords, 4–6, 8
methods, 9–11
statement delimiters, 11–12
statements, 11–12
type definition, 8–9
variables, 13–17
System, 165
System.Action delegates, 524–525
System.ApplicationException, 210, 435, 436
System.ArgumentException, 210, 433, 436
System.ArithmeticException, 210
System.Array, 484
System.Array.Reverse() method, 85–86
System.ArrayTypeMismatchException, 210
System.Attribute, 699, 711
System.AttributeUsageAttribute, 706–709
System.Collections, 165
IEnumerate<T>. See IEnumerable<T> interface.
System.Collections.Generic, 165
System.Collections.IEnumerate. See IEnumerable interface.
System.ComponentModel.Win32Exception method, 855–856, 862
System.Console.Clear() method, 154
System.Console.Read() method, 19
System.Console.ReadKey() method, 19
System.Console.ReadLine() method
calling, 163–164
reading from the console, 18–19
return values, 168
System.Console.Write() method
calling, 163–164
return values, 168
starting a new line, 48, 55
writing to the console, 19–21
System.Console.WriteLine() method
calling, 163–164
outputting a blank line, 55
overriding ToString() method, 384–385
return values, 168
round-trip formatting, 44–45
starting a new line, 48, 55
writing to the console, 19–21
System.Convert, 69–70
System.Data, 165
SqlException(), 448
System.Delegate
constraint limitations, 484
delegate internals, 513–516
multicast delegate internals, 554
System.Delegate.Combine() method
combining delegates, 552
constraint limitations, 484
event internals, 568
System.Delegate.Remove() method
event internals, 568
removing delegates from chains, 552
removing list elements, 649
System.Diagnostics.
ConditionalAttribute, 710–711
System.Diagnostics.Processor, 851
System.Diagnostics.Trace.Write() method, 384
System.Drawing, 165
System.Dynamic.DynamicObject, 728
System.Dynamic.IDynamicMetaObjectProvider interface, 726–729
System.Enum, 484
System.Enum.IsDefined() method, 379
System.Enum.Parse() method, 376
System.Environment.CommandLine, 10
System.EventArgs, 563–565
System.EventHandler<T>, 566
System.Exception, 209–210, 435, 436
System.ExecutionEngineException, 435
System.FormatException, 209–210, 435, 436
System.Func delegates, 524–525
System.GC, 419
System.GC.SuppressFinalize() method, 426
System.IndexOutOfRangeException, 210
System.IntPtr, 852
System.InvalidCastException, 210
System.InvalidOperationException, 210, 438
System.IO, 165
System.IO.FileAttributes, 377
System.Linq.Enumerable aggregate functions, 618
Average() method, 618
Count() method, 618
GroupBy() method, grouping results, 610–611
GroupJoin() method, 611–613
Join() method, 607–610
Max() method, 618
method calls, 616–617
Min() method, 618
Select() method, 592–594
Sum() method, 618
Where() method, 591–592
System.Linq.Enumerable methods, 828–829
System.Runtime.InteropServices.COMException, 435
System.Runtime.InteropServices.SafeHandle, 858
System.Runtime.InteropServices.SEHException, 435
System.Runtime.Serialization.ISerializable, 715–716
System.Runtime.Serialization.OptionalFieldAttribute, 718
System.SerializableAttribute, 713–714, 718–719
System.Text.RegularExpressions, 165
System.Text.StringBuilder, 58, 477
System.Text.StringBuilder.Remove() method, 58
System.Threading, 165
System.Threading.Interlocked
Add() method, 829
CompareExchange() method, 828–829
CompareExchange<T> method, 829
Decrement() method, 829
Exchange<T> method, 829
Increment() method, 829
Read() method, 829
System.Threading.Interlocked methods, 828–829
System.Threading.Monitor
Enter() method, 366, 821, 823, 825–826
System.Threading.Monitor.Pulse() method, 823
System.Threading.Mutex, 833–835
System.Threading.Tasks, 165
System.Threading.Thread, 734. See also
   Multithreading, with System.Threading.Thread.
System.Threading.Timer, 846
System.Threading.WaitHandle, 835
System.Timers.Timer, 846
System.Type class, accessing metadata, 685
System.UnauthorizedAccessException, 449
System.ValueType, 361–362
System.Web, 165
System.Windows, 166
System.Windows.Forms, 166
System.Windows.Forms.Timer, 846
   DispatcherTimer, 846
System.Xml, 166

T
TAP (Task-based Asynchronous Pattern).
   See Multithreading, task-based asynchronous pattern.
Target property, 513
Task Parallel Library (TPL). See TPL (Task Parallel Library).
Task schedulers, 750–751, 793–795
Task-based asynchronous pattern.
   See Multithreading, task-based asynchronous pattern.
Task-based Asynchronous Pattern (TAP).
   See Multithreading, task-based asynchronous pattern.
TaskCanceledException, 772
TaskContinuationSource.SetResult() method, 789
TaskContinuationSource<T> object, 788–789
TaskCreationOptions enums, 758–759
Task.ContinueWith() method, 756, 793–794, 798
TaskCreationOptions.LongRunning option, 773–774
Task.Run() method, 772–773
Tasks
   antecedent, 757
   associating data with, 755
   asynchronous. See Multithreading, asynchronous tasks.
canceling. See Multithreading, canceling tasks.
   chaining, 757
cold, 752
composing large from smaller, 756–758
continuation, 755–762
control flow within, 784–786
creating, 750–751
definition, 735, 750
delegates, 751
disposable, canceling, 774
drawbacks, 775
hot, 752
identification, 755
long-running, canceling, 773–774
registering for notification of behavior, 760–761
status, getting, 754
TaskScheduler, 793
Task<T>, 753–754
Temporary storage pool. See Stack.
Ternary operators, definition, 123
TextNumberParser.Parse() method, 434
TextToUpper() method, 57
ThenBy() method, 601–603
ThenByDescending() method, 602
Thermostat, 545–546
this keyword
   avoiding ambiguity, 229–232
definition, 228
   identifying field owner, 228–229
   locking, 826–827
   with a method, 230–231
   passing in a method call, 231–232
   in static methods, 270–271
Thread management, 743–744
Thread pool, definition, 735
Thread pooling, definition, 747–749
Thread safe code, definition, 734
Thread safe delegate invocation, 550
Thread safety
   definition, 819
delegates, 550
Thread synchronization. See also
   Deadlocks; Race conditions.
   best practices, 831–833
   monitors, 821–823
   multiple threads, 821–823
   with no await operator, 820
timers, 845–846
Thread synchronization, synchronization types
concurrent collection classes, 840–841
reset events, 836–839
Thread synchronization, thread local storage
definition, 841
ThreadLocal<T>, 841–842
ThreadStaticAttribute, 843–845
Thread synchronization, uses for atomicity of reading and writing to variables, 819
declaring fields as volatile, 828
event notification with multiple threads, 830
lock keyword, 823–825
lock objects, 825
lock performance, 825
lock synchronization, 823–825
locking guidelines, 827
locking on this, typeof, and string, 826–827
withMethodImplAttribute, 827
withMethodImplOptions.
Synchronized() method, 827
multiple threads and local variables, 819–820
sample pseudocode execution, 818
synchronizing local variables, 819–820
synchronizing multiple threads with Monitor class, 821
with System.Threading.Interlocked
methods, 828–829
thread safety, definition, 819
thread-safe event notification, 831
torn read, definition, 819
unsynchronized local variables, 820
unsynchronized state, 817
volatile keyword, 828
ThreadAbortException, 745–746
Threading model, definition, 734
ThreadLocal<T>, 841–842
Threads
aborting, 745–746
checking for life, 744
creating, 750–751
definition, 734
foreground vs. background, 743
putting to sleep, 744
reprioritizing, 743
unhandled exceptions, 765–768
waiting for, 743
Thread-safe event notification, 831
Thread.Sleep() method, putting threads to sleep, 744
ThreadState property, 744
ThreadStaticAttribute, 843–845
Three-dimensional arrays, 77–78
Throw() method, 439–440
throw statements, 212–215
ThrowIfCancellationRequested() method, 772
Throwing exceptions. See also Catching exceptions; Exception handling.
ArgumentNullException, 436
ArgumentOutOfRangeException, 436
checked and unchecked conversions, 451–453
code sample, 434
description, 203–204
guidelines, 450
identifying the parameter name, 435
nameof operator, 435, 436
NullReferenceException, 436
rethrowing, 438–440
rethrowing wrapped exceptions, 449–453
throw statement, 212–215
without replacing stack information, 439
TicTacToe game. See also Arrays.
checking player input, 142–143
conditional operators, 123
determining remaining moves, 141
#region preprocessor directive, example, 158–159
escaping out of, 146–147
if/else example, 111
initializing, 73
nested if statements, 112
#endregion preprocessor directive, example, 158–159
source code, 901–905
tracking player moves, 147–148
Tilde (~) bitwise complement operator, 134
Time slice, definition, 736
Time slicing, definition, 736
Time slicing costs, 737
Timers. See Thread synchronization, timers.
TKey parameter, 479
ToArray() method, 600
ToCharArray() method, 85–86
Index

ToDictionary() method, 600
ToList() method, 600
ToLookup() method, 600
Torn read, definition, 819
ToString() method, 69–70, 384–385, 709
TPL (Task Parallel Library). See also Multi-threading, parallel loop iterations.
  asynchronous high-latency operations, 777–781
  performance tuning, 802–803
Trapping errors, 203–209. See also Exception handling.
TrimToSize() method, 662
True/false evaluation. See Boolean expressions.
Try blocks, 205–206
TryGetMember() method, 728
TryGetPhoneNumber() method, 186–188
TryParse() method, 70–71, 215–216, 690–692
TryParse<T>() method, 376
TrySetMember() method, 728
Tuple, 471–472
Tuple.Create() method, 471–472
TValue parameter, 479
Two-dimensional arrays, 74, 79
Type
  array defaults, 73
  compatibility, enums, 374–375
  conversion, programming with dynamic objects, 721–722
  incompatibilities, anonymous types, 576–577
  inference, generic methods, 487–489
  name qualifier, calling methods, 166–167
Type categories, reference types
  definition, 61
  description, 62–64
  heaps, 62–63
  memory area of the referenced data, 63
Type categories, value types
  ? (question mark), nullable modifier, 64–65
  assigning to null, 64–65
  definition, 61
  description, 62
Type definition
  casing, 8
  naming conventions, 8
  overloading, 471–472
  syntax, 8–9
Type objects, retrieving, 686–687
Type parameter list, declaring methods, 172
Type parameters
  constraints on. See Constraints on type parameters.
  generic classes, 464
  for generic classes or methods, 693–694
  naming guidelines, 465–466
Type parameters, 472, 489
Type safety
  anonymous types, 576–577
  covariance, 498
  managed execution, 27
  programming with dynamic objects, 720–721, 726
Type.ContainsGenericParameters
  property, 693
typeof keyword, locking, 826–827
typeof() method, 686–687
typeof operator, 520, 692
Types
  aliasing, 179–180. See also using directive.
  anonymous, 61, 263–265
  bool (Boolean), 45
  char (character), 14, 45
  data conversion with the as operator, 322–323
  declaring on the fly. See Anonymous types.
  definition, 15
  encapsulating, 407–408
  extending. See Inheritance.
  implicitly typed local variables, 60–61
  integral, 96
  null, 58–59
  predefined, 35–36
  string, 48
  for strings, 48
  underlying, determining, 321–322
  Unicode standard, 46–48
  void, 58, 59–60
  well formed. See Well-formed types.
Types, conversions between
  cast operator, 65–66
  casting, 65
  checked block example, 67
  checked conversions, 66–68
  defining custom conversions, 295–296
  explicit cast, 65–66
  implicit conversion, 65, 68–69
numeric to Boolean, 68
overflowing an integer value, 66–68
Parse() method, 69–70
System.Convert class, 69–70
ToString() method, 69–70
TryParse() method, 70–71
unchecked block example, 67–68
unchecked conversions, 66–68
without casting, 69–70
Types, fundamental numeric. See also
  Literal values.
  byte, 36
  C# vs. C++ short type, 37
defaults, 40–42
floating-point types. See Floating-point
types.
  formatting numbers as hexadecimal, 43
hardcoding values, 40–42
hexadecimal notation, 42–43
  int, 36
  int (integer), 14
integer literals, determining type of, 41–42
integers, 36–37
  keywords associated with, 36
  long, 36
  sbyte, 36
  short, 36
  uint, 36
  ulong, 36
  ushort, 36

U
  uint type, 36
  ulong type, 36
  UML (Unified Modeling Language), 345
Unary operators
  definition, 122
  overloading, 400–401
UnaryExpression, 535
UnauthorizedAccessException, 449, 804
Unboxing, 363–364, 366
Unchecked block example, 67–68
Unchecked conversions, 66–68
Uncompress() method, 326–327
#undef preprocessor directive, 153, 155
Underscore (_)
  in identifier names, 7
  line continuation character, 11
  in variable names, 15
Underscores (_), in keyword names, 8
Undo, with a generic Stack, 461
Undo() method, 458
Unexpected inequality, float type, 97–100
Unhandled exceptions
  error messages, 203–204
  handling with AggregateException, 762–765
  observing, 764–765
  registering for, 766–768
  on a thread, 765–768
UnhandledException event, 766
Unicode standard, 46–48
Unified Modeling Language (UML), 345
Union() method, 618
Unmanaged code, definition, 26
Unmanaged types, 865
Unmodifiable. See Immutable.
Unsafe code. See also P/Invoke; Pointers
  and addresses; WinRT.
description, 862–864
  executing by delegate, 872–873. See also
  P/Invoke; WinRT.
unsafe code blocks, 863–864
unsafe modifier, 863
unsafe statement, 863
/unsafe switch, 864
Unsynchronized local variables, 820
Unsynchronized state, 817
Unwrap() method, 779
ushort type, 36
using directives
  dropping namespaces, 53–54, 175–177
  example, 53–54, 175
  importing types from namespaces,
    175–177
  nested namespaces, 176
  nesting, 177–178
  wildcards, Java vs. C#, 176
using statement
  deterministic finalization, 423–426
resource cleanup, 425–426
using static directive
  abbreviating a type name, 178–179
dropping namespaces, 53–54
  example, 53–54, 178–179
V
Validating properties, 244–246
Value, passing parameters by, 183–184
value keyword, 240
Value type conversion
to an implemented interface. See Boxing.
to its root base class. See Boxing.
Value types
custom types. See Enums; Structs.
default operator, 360-361
guidelines, 353, 362
immutability, 357
inheritance, 361-362
interfaces, 361-362
introduction, 352-353
new operator, 359-360
vs. reference types, 184-185, 353-355
temporary storage pool. See stack.
Values
CTS types, 892
hardcoding, 40-42
var keyword
anonymous types, 572-576
C++ vs. C#, 575
implicitly typed local variables, 60-61
JavaScript vs. C#, 575
Visual Basic vs. C#, 575
Variables. See also Local variables.
setting to null, 58
syntax, 13-17
type, 14
using, 17
Variables, global, C++ vs. C#, 266
Variant
C++ vs. C#, 575
JavaScript vs. C#, 575
Visual Basic vs. C#, 575
VerifyCredentials() method, 344
Versioning, 716-718
Versioning interfaces, 346-347
Vertical bar, equal sign (|=) compound
assignment operator, 133-134
Vertical bar (|) OR operator, 131, 132, 397-399
Vertical bars (||) OR operator, 121
VES (Virtual Execution System). See also CIL (Common Intermediate
Language); CLI (Common Language Infrastructure); Runtime.
definition, 878, 896
managed execution, 26
Virtual abstract members, 317
Virtual computer, 872-873
Virtual Execution System (VES). See VES
(Virtual Execution System).
Virtual fields, properties as, 249-250
Virtual functions, pure, 317
Virtual members, sealing, 311-312
Virtual memory, allocating with
P/Invoke, 852
Virtual methods
custom dynamic objects, 728
Java vs. C#, 440
overriding base classes, 302-307
virtual modifier, 302-307
VirtualAllocEx() API, 851-853
VirtualMemoryManager, 851
VirtualMemoryPtr, 858
Visual Basic vs. C#
importing namespaces, 176
line-based statements, 11
this keyword, 230
var keyword, 575
Variant, 575
void*, 575
void type, 59
Visual code editors, 158-159
void*
C++ vs. C#, 575
JavaScript vs. C#, 575
Visual Basic vs. C#, 575
Void methods, 173-174
void type
C++ vs. C#, 59
description, 59-60
no value vs. empty string, 59
in partial methods, 287
as a return, 173-174
strings, 59-60
use for, 58
volatile keyword, 828
W
Wait() method, 836-837
WaitAll() method, 835
WaitAny() method, 835
WaitAsync() method, 840
WaitForExit() method, 788
WaitHandle, 774
WaitOne() method, 835, 837
Warning messages, disabling/restoring, 156-157
#warning preprocessor directive, 153,
155-156
Weak references, garbage collection, 420-421
WebRequest.GetResponseAsync() method, 779
Well-formed types
determining whether two objects are equal, 392–395
implementing Equals() equality operator, 392–395
lazy initialization, 429–431
object identity vs. equal object values, 388–392
overriding Equals() equality operator, 392–395
Well-formed types, garbage collection. See also Resource cleanup.
Collect() method, 419
introduction, 418
in .NET, 418–419
root references, 418
strong references, 420–421
weak references, 420–421
Well-formed types, namespaces in the CLR (Common Language Runtime), 410
extern alias directive, 413
guidelines, 412
introduction, 409–410
namespaces alias qualifier, 412–413
naming conventions, 410
nesting, 411
Well-formed types, overloading object members
Equals() equality operator, 388–395
GetHashCode() method, 385–387
ToString() method, 384–385
Well-formed types, overloading operators
binary operators, 397–399
binary operators combined with assignment operators, 397–399
cast operator, 402
conditional logical operators, 400
conversion operators, 401, 403
unary operators, 400–401
Well-formed types, referencing other assemblies
changing the assembly target, 404
class libraries, 404–405
console executables, 404
encapsulation of types, 407–408
internal access modifiers on type declarations, 407–408
modules, 405
PCLs (portable class libraries), 406
protected internal type modifier, 408
public access modifiers on type declarations, 407–408
referencing assemblies, 405–406
referencing assemblies on Mac and Linux, 406
type member accessibility modifiers, 409. See also specific modifiers.
Windows executables, 405
Well-formed types, resource cleanup
exception propagation from constructors, 428
finalization, 421–427
finalization queue, 426
garbage collection, 426–427
guidelines, 428
with IDisposable, 424–425, 426–427
introduction, 421
invoking the using statement, 425–426
resurrecting objects, 429
Well-formed types, XML comments
associating with programming constructs, 414–416
generating an XML documentation file, 416–418
guidelines, 418
introduction, 413–414
when clauses, catching exceptions, 438
where clauses, 623–624, 631–632
Where() method, 591–592, 597–598
while loops, 108, 134–135, 136–137
While() method, 584
while statement, 108, 134–135, 136–137
Whitespace, 12, 13
Win32, error handling in P/Invoke, 855–857
Windows Desktop CLR compiler, 880
Windows Runtime. See WinRT.
WinRT. See also P/Invoke.
automatically shimmed interfaces, 875–876
definition, 896
description, 873–874
task-based asynchrony, 876
WinRT events, 874–875
Work stealing, 802–803
Wrappers for API calls from P/Invoke, 860–861
Write() method
starting a new line, 48, 55
writing to the console, 19–21
WriteLine() method
  - round-trip formatting, 44–45
  - starting a new line, 48, 55
  - writing to the console, 19–21
Write-only properties, 247–248
WriteWebRequestSizeAsync() method, 780, 783

X
XML (Extensible Markup Language), 25, 416–418
XML comments
  - associating with programming constructs, 414–416
  - delimited comments, 24
  - generating an XML documentation file, 416–418

Y
yield break statement, 677–678
yield keyword, 6
yield return statement in foreach loops, 674–676
  - implementing BinaryTree<T>, 673–676
  - requirements, 681
  - returning iterator values, 670–671
yield return statements, 6

Z
ZipCompression, 337