“The Essential C# series is a classic, and teaming up with famous C# blogger Eric Lippert on the new edition is another masterstroke!...[The authors] share a great gift of providing clarity and elucidation, and by combining their ‘inside’ and ‘outside’ perspectives on C#, this book reaches a new level of completeness. Welcome to one of the greatest collaborations you could dream of in the world of C# books!”

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

Essential C# 5.0

Mark Michaelis

with Eric Lippert

FREE SAMPLE CHAPTER

SHARE WITH OTHERS
Essential C# 5.0
Microsoft Windows Development Series

Visit informit.com/mswinseries for a complete list of available publications.

The Windows Development Series grew out of the award-winning Microsoft .NET Development Series established in 2002 to provide professional developers with the most comprehensive and practical coverage of the latest Windows developer technologies. The original series has been expanded to include not just .NET, but all major Windows platform technologies and tools. It is supported and developed by the leaders and experts of Microsoft development technologies, including Microsoft architects, MVPs and RDs, and leading industry luminaries. Titles and resources in this series provide a core resource of information and understanding every developer needs to write effective applications for Windows and related Microsoft developer technologies.

“This is a great resource for developers targeting Microsoft platforms. It covers all bases, from expert perspective to reference and how-to. Books in this series are essential reading for those who want to judiciously expand their knowledge and expertise.”

– JOHN MONTGOMERY, Principal Director of Program Management, Microsoft

“This series is always where I go first for the best way to get up to speed on new technologies. With its expanded charter to go beyond .NET into the entire Windows platform, this series just keeps getting better and more relevant to the modern Windows developer.”

– CHRIS SELLS, Vice President, Developer Tools Division, Telerik

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

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

1 Introducing C#  1
2 Data Types  33
3 Operators and Control Flow  85
4 Methods and Parameters  155
5 Classes  209
6 Inheritance  277
7 Interfaces  313
8 Value Types  339
9 Well-Formed Types  371
10 Exception Handling  423
11 Generics  443
12 Delegates and Lambda Expressions  495
Contents at a Glance

13 Events 533
14 Collection Interfaces with Standard Query Operators 561
15 LINQ with Query Expressions 613
16 Building Custom Collections 635
17 Reflection, Attributes, and Dynamic Programming 677
18 Multithreading 727
19 Thread Synchronization 811
20 Platform Interoperability and Unsafe Code 845
21 The Common Language Infrastructure 875
A Downloading and Installing the C# Compiler and CLI Platform 897
B Tic-Tac-Toe Source Code Listing 901
C Interfacing with Mutithreading Patterns Prior to the TPL and C# 5.0 907
D Timers Prior to the Async/Await Pattern of C# 5.0 937

Index 943
Index of C# 5.0 Topics 974
Index of C# 4.0 Topics 975
Index of C# 3.0 Topics 984
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 17

2 Data Types 33
   Fundamental Numeric Types 34
   More Fundamental Types 43
   null and void 53
   Categories of Types 57
   Nullable Modifier 60
   Conversions between Data Types 60
   Arrays 67

3 Operators and Control Flow 85
   Operators 86
   Introducing Flow Control 103
   Code Blocks ({}) 110
## Contents

- Code Blocks, Scopes, and Declaration Spaces  
  112
- Boolean Expressions  
  114
- Bitwise Operators (\(<\), \(\rangle\), |, &, ^, ~)  
  121
- Control Flow Statements, Continued  
  127
- Jump Statements  
  139
- C# Preprocessor Directives  
  145

### 4 Methods and Parameters  
155

- Calling a Method  
  156
- Declaring a Method  
  163
- The using Directive  
  168
- Returns and Parameters on \(\text{Main}()\)  
  172
- Advanced Method Parameters  
  175
- Recursion  
  184
- Method Overloading  
  186
- Optional Parameters  
  189
- Basic Error Handling with Exceptions  
  194

### 5 Classes  
209

- Declaring and Instantiating a Class  
  213
- Instance Fields  
  217
- Instance Methods  
  219
- Using the this Keyword  
  220
- Access Modifiers  
  227
- Properties  
  229
- Constructors  
  244
- Static Members  
  255
- Extension Methods  
  265
- Encapsulating the Data  
  267
- Nested Classes  
  269
- Partial Classes  
  272

### 6 Inheritance  
277

- Derivation  
  278
Contents

Overriding the Base Class 290
Abstract Classes 302
All Classes Derive from System.Object 308
Verifying the Underlying Type with the is Operator 309
Conversion Using the as Operator 310

7 Interfaces 313
Introducing Interfaces 314
Polymorphism through Interfaces 315
Interface Implementation 320
Converting between the Implementing Class and Its Interfaces 326
Interface Inheritance 326
Multiple Interface Inheritance 329
Extension Methods on Interfaces 330
Implementing Multiple Inheritance via Interfaces 331
Versioning 334
Interfaces Compared with Classes 336
Interfaces Compared with Attributes 337

8 Value Types 339
Structs 340
Boxing 349
 Enums 358

9 Well-Formed Types 371
Overriding object Members 371
Operator Overloading 385
Referencing Other Assemblies 393
Defining Namespaces 398
XML Comments 402
Garbage Collection 407
Resource Cleanup 410
Lazy Initialization 419
Contents

10 Exception Handling 423
   Multiple Exception Types 424
   Catching Exceptions 426
   General Catch Block 430
   Guidelines for Exception Handling 432
   Defining Custom Exceptions 435
   Wrapping an Exception and Rethrowing 438

11 Generics 443
   C# without Generics 444
   Introducing Generic Types 449
   Constraints 462
   Generic Methods 476
   Covariance and Contravariance 481
   Generic Internals 489

12 Delegates and Lambda Expressions 495
   Introducing Delegates 496
   Lambda Expressions 506
   Anonymous Methods 512
   General-Purpose Delegates: System.Func and System.Action 514

13 Events 533
   Coding the Observer Pattern with Multicast Delegates 534
   Events 548

14 Collection Interfaces with Standard Query Operators 561
   Anonymous Types and Implicitly Typed Local Variables 562
   Collection Initializers 568
   What Makes a Class a Collection: IEnumerable<T> 571
   Standard Query Operators 577

15 LINQ with Query Expressions 613
   Introducing Query Expressions 614
   Query Expressions Are Just Method Invocations 632
16 Building Custom Collections  635
   More Collection Interfaces  636
   Primary Collection Classes  638
   Providing an Indexer  655
   Returning null or an Empty Collection  659
   Iterators  660

17 Reflection, Attributes, and Dynamic Programming  677
   Reflection  678
   Attributes  688
   Programming with Dynamic Objects  714

18 Multithreading  727
   Multithreading Basics  730
   Working with System.Threading  737
   Asynchronous Tasks  745
   Canceling a Task  764
   The Task-Based Asynchronous Pattern in C# 5.0  770
   Executing Loop Iterations in Parallel  794
   Running LINQ Queries in Parallel  804

19 Thread Synchronization  811
   Why Synchronization?  813
   Timers  841

20 Platform Interoperability and Unsafe Code  845
   Using the Windows Runtime Libraries from C#  846
   Platform Invoke  849
   Pointers and Addresses  862
   Executing Unsafe Code via a Delegate  872

21 The Common Language Infrastructure  875
   Defining the Common Language Infrastructure (CLI)  876
   CLI Implementations  877
   C# Compilation to Machine Code  879
Contents

Runtime 881
Application Domains 887
Assemblies, Manifests, and Modules 887
Common Intermediate Language (CIL) 890
Common Type System (CTS) 891
Common Language Specification (CLS) 891
Base Class Library (BCL) 892
Metadata 892

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

B Tic-Tac-Toe Source Code Listing 901

C Interfacing with Mutithreading Patterns Prior to the TPL and C# 5.0 907
Asynchronous Programming Model 908
Asynchronous Delegate Invocation 921
The Event-Based Asynchronous Pattern (EAP) 924
Background Worker Pattern 928
Dispatching to the Windows UI 932

D Timers Prior to the Async/Await Pattern of C# 5.0 937

Index 943
Index of C# 5.0 Topics 974
Index of C# 4.0 Topics 975
Index of C# 3.0 Topics 984
Figures

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

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

**Figure 4.1:** Exception-Handling Control Flow  

**Figure 5.1:** Class Hierarchy  

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

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

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

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

**Figure 12.1:** Delegate Types Object Model  
**Figure 12.2:** Anonymous Function Terminology  
**Figure 12.3:** The Lambda Expression Tree Type  
**Figure 12.4:** Unary and Binary Expression Tree Types
Figures

**Figure 13.1:** Delegate Invocation Sequence Diagram  543
**Figure 13.2:** Multicast Delegates Chained Together  544
**Figure 13.3:** Delegate Invocation with Exception Sequence Diagram  545

**Figure 14.1:** IEnumerable<T> and IEnumerator Interfaces  573
**Figure 14.2:** Sequence of Operations Invoking Lambda Expressions  589
**Figure 14.3:** Venn Diagram of Inventor and Patent Collections  593

**Figure 16.1:** Generic Collection Interface Hierarchy  637
**Figure 16.2:** List<T> Class Diagrams  639
**Figure 16.3:** Dictionary Class Diagrams  646
**Figure 16.4:** SortedList<T> and SortedDictionary<T> Class Diagrams  653
**Figure 16.5:** Stack<T> Class Diagram  654
**Figure 16.6:** Queue<T> Class Diagram  654
**Figure 16.7:** LinkedList<T> and LinkedListNode<T> Class Diagrams  655
**Figure 16.8:** Sequence Diagram with yield return  665

**Figure 17.1:** MemberInfo Derived Classes  685
**Figure 17.2:** BinaryFormatter Does Not Encrypt Data  708

**Figure 18.1:** Clock Speeds over Time  728
**Figure 18.2:** CancellationTokenSource and CancellationToken Class Diagrams  767

**Figure 20.1:** Pointers Contain the Address of the Data  865
**Figure 21.1:** Compiling C# to Machine Code  880
**Figure 21.2:** Assemblies with the Modules and Files They Reference  889

**Figure C.1:** APM Parameter Distribution  911
**Figure C.2:** Delegate Parameter Distribution to BeginInvoke() and EndInvoke()  924
Tables

**Table 1.1:** C# Keywords 5
**Table 1.2:** C# Comment Types 22
**Table 1.3:** C# and .NET Versions 27

**Table 2.1:** Integer Types 34
**Table 2.2:** Floating-Point Types 36
**Table 2.3:** decimal Type 36
**Table 2.4:** Escape Characters 45
**Table 2.5:** string Static Methods 49
**Table 2.6:** string Methods 50
**Table 2.7:** Array Highlights 68
**Table 2.8:** Common Array Coding Errors 82

**Table 3.1:** Control Flow Statements 104
**Table 3.2:** Relational and Equality Operators 116
**Table 3.3:** Conditional Values for the XOR Operator 118
**Table 3.4:** Preprocessor Directives 146
**Table 3.5:** Operator Order of Precedence 153

**Table 4.1:** Common Namespaces 159
**Table 4.2:** Common Exception Types 202

**Table 6.1:** Why the New Modifier? 296
**Table 6.2:** Members of System.Object 308
Tables

**Table 7.1:** Comparing Abstract Classes and Interfaces  337

**Table 8.1:** Boxing Code in CIL  350

**Table 9.1:** Accessibility Modifiers  398

**Table 12.1:** Lambda Expression Notes and Examples  511

**Table 14.1:** Simpler Standard Query Operators  608

**Table 14.2:** Aggregate Functions on `System.Linq.Enumerable`  609

**Table 17.1:** Deserialization of a New Version Throws an Exception  711

**Table 18.1:** List of Available `TaskContinuationOptions` Enums  754

**Table 18.2:** Control Flow within Each Task  780

**Table 19.1:** Sample Pseudocode Execution  814

**Table 19.2:** `Interlocked`’s Synchronization-Related Methods  825

**Table 19.3:** Execution Path with `ManualResetEvent` Synchronization  833

**Table 19.4:** Concurrent Collection Classes  836

**Table 21.1:** Primary C# Compilers  878

**Table 21.2:** Common C#-Related Acronyms  894

**Table D.1:** Overview of the Various Timer Characteristics  938
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 is already a classic, and teaming up with famous C# blogger Eric Lippert on the new edition is another 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, 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 that I have since formed great relationships with. But to be honest, I don’t remember it. 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 technol-
ogy. 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 colleague of seven years on the C# team. He’s been
here much longer than I have, 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 is grappling with a complex issue and is sharing his thoughts
about it with the team, his emails about it are just as meticulous and every bit
as hilarious. You fundamentally can’t ignore an issue raised by Eric because
you can’t wait to read his prose about it. They’re even purple, too! So I es-
sentially get 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 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#, this book reaches a new level of completeness. No one will help you get C# 5.0 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 takes you through these same paradigm shifts.

The beginning chapters take you through sequential programming structure in which statements are executed in the order in which they are written. The problem with this model is that complexity increases exponentially as the requirements increase. To reduce this complexity, code blocks are moved into methods, creating a structured programming model. This allows you to call the same code block from multiple locations within a program, without duplicating code. Even with this construct, however, programs quickly become unwieldy and require further abstraction. Object-oriented programming, discussed in Chapter 5, was the response. In subsequent chapters, you will learn about additional methodologies, such as interface-based programming, LINQ (and the transformation it makes to the collection API), and eventually rudimentary forms of declarative programming (in Chapter 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#, this book provides insight into some of the more complex programming paradigms and provides
in-depth coverage of the features introduced in the latest version of the language, C# 5.0 and .NET Framework 4.5.

- 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, ADO.NET, smart client development, 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 Microsoft Windows Development Series provides a wealth of writing on these topics. Essential C# 5.0 focuses on C# and the types within the Base Class Library. Reading this book will prepare you to focus on and develop expertise in any of the areas covered by the rest of the series.

**Target Audience for This Book**

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

- **Beginners**: If you are new to programming, this book serves as a resource to help transition you from an entry-level programmer to a C# developer, comfortable with any C# programming task that’s thrown your way. This book not only teaches you syntax, but also trains you in good programming practices that will serve you throughout your programming career.
Structured programmers: Just as it’s best to learn a foreign language through immersion, learning a computer language is most effective when you begin using it before you know all the intricacies. In this vein, this book begins with a tutorial that will be comfortable for those familiar with structured programming, and by the end of Chapter 4, developers in this category should feel at home writing basic control flow programs. However, the key to excellence for C# developers is not memorizing syntax. To transition from simple programs to enterprise development, the C# developer must think natively in terms of objects and their relationships. To this end, Chapter 5’s Beginner Topics introduce classes and object-oriented development. The role of historically structured programming languages such as C, COBOL, and FORTRAN is still significant but shrinking, so it behooves software engineers to become familiar with object-oriented development. C# is an ideal language for making this transition because it was designed with object-oriented development as one of its core tenets.

Object-based and object-oriented developers: C++ 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++ style languages that you already know.

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

- 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)
Preface

– 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. Even experienced C# developers often do not understand this topic well.

Features of This Book

Essential C# 5.0 is a language book that adheres to the core C# Language 5.0 Specification. To help you understand the various C# constructs, it provides numerous examples demonstrating each feature. Accompanying each concept are guidelines and best practices, ensuring that code compiles, avoids likely pitfalls, and achieves maximum maintainability.

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

C# Coding Guidelines

One of the more significant enhancements added to Essential C# 5.0, and not explicitly called out in previous editions, was the addition 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 enable maintenance well into the future. The coding guidelines highlight some of the key principles that readers will want to be sure to incorporate into their development.

**Code Samples**

The code snippets in most of this text can run on any implementation of the Common Language Infrastructure (CLI), including the Mono, Rotor, 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 indexes at the end of the book.

Here is a sample code listing.

**Listing 1.9: Declaring and Assigning a Variable**

```csharp
class MiracleMax
{
    static void Main()
    {
        string max;
        max = "Have fun storming the castle!";
        System.Console.WriteLine(max);
    }
}
```

The formatting is as follows.

- Comments are shown in italics.

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

- Keywords are shown in bold.
Preface

static void Main()

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

    System.Console.Write /* No new 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 you from 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/mswinseries.
Mind Maps
Each chapter’s introduction includes a mind map, which serves as an outline that provides 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 will help you navigate through the text.

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

At a high level, software engineering is about managing complexity, and it is toward this end that I have organized *Essential C# 5.0*. 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** 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. It also touches on the context of a C# program’s execution and its intermediate language.

- **Chapter 2—Data Types:** Functioning programs manipulate data, and this chapter introduces the primitive data types of C#. 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 an out parameter. In C# 4.0 default parameter support was added and this chapter explains how to use them.

- **Chapter 5—Classes:** Given the basic building blocks of a class, this chapter combines these constructs together 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 not as prevalent as defining reference types, it is sometimes necessary to define value types that behave in a fashion similar to the primitive types built into C#. This chapter describes how to define structures, while exposing the idiosyncrasies they may introduce.

- **Chapter 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 defining namespaces and 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 enables creating 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 virtually eliminates the need for writing routines that poll. Lambda expressions are the key concept that make C# 3.0’s LINQ possible. This chapter explains how lambda expressions build on the delegate construct by providing a more elegant and succinct syntax. This chapter forms the foundation for the new collection API discussed next.

• **Chapter 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 and yet elegantly powerful changes introduced in C# 3.0 begin to shine in this chapter as we take a look at the extension methods of the new Enumerable class. This class makes available an entirely new collection API known as the standard query operators and discussed in detail here.

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

• **Chapter 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. This 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 active response to simultaneous events. As programs become more sophisticated, they must take additional precautions to protect data in these advanced environments. Programming multithreaded applications is complex. This chapter discusses how to work with threads and provides best practices to avoid the problems that plague multithreaded applications.

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

Chapter 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 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 source code displayed in parts within Chapter 3 and Chapter 4.

Appendix C—Interfacing with Multithreading Patterns prior to the TPL and C# 5.0: This appendix provides details on multithreading patterns for development prior to C# 5.0 and/or the Task Parallel Library.

Appendix D—Timers prior to the Async/Await Pattern of C# 5.0: This appendix describes three different types of timers for use when .NET 4.5/C# 5.0 is not available.
Preface

- C# 3.0, 4.0, 5.0 Index: These indexes provide a quick reference for the features added in C# 3.0–5.0. They are specifically designed to help programmers quickly update their language skills to a more recent version.

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

—Mark Michaelis
IntelliTect.com/mark
Twitter: @Intellitect, @MarkMichaelis
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 that come first. By far, my family has made the biggest sacrifice to allow me to complete this. 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 virtually 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 still managed to catch: Paul Bramsman, Kody Brown, Ian Davis, Doug Dechow, Gerard Frantz, Thomas Heavey, Anson Horton, Brian Jones, Shane Kercheval, Angelika Langer, Eric Lippert, John Michaelis, Jason Morse, Nicholas Paldino, Jon Skeet, Michael Stokesbary, Robert Stokesbary, John Timney, and Stephen Toub. Thanks also to Mandy Frei who diligently kept notes of changes needed for reprints.

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 is no longer. Eric painstakingly reviewed every Essential C# 5.0 chapter with amazing
detail and precision. I am extremely grateful for his contribution to making this book even better than the earlier 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.

Like Eric and C#, there are fewer than 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 asynchronous support in C# 5.0. Thanks, Stephen!

Thanks to everyone at Addison-Wesley for their patience in working with me in spite of my frequent focus on everything else except the manuscript. Thanks to Elizabeth Ryan, Audrey Doyle, Vicki Rowland, Curt Johnson, and Joan Murray. Joan deserves a special medal of patience for the number of times I delayed not only with deliverables but even responding to emails.
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, and 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. He speaks at developer conferences and has written numerous articles and other books. He holds a bachelor of arts degree 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, he is busy with his family or training for another triathlon (having completed his first Ironman in 2008). He lives in Spokane, Washington, with his wife Elisabeth and three children, Benjamin, Hanna, and Abigail.

Eric Lippert is a principal developer on the C# compiler team at Microsoft. He has worked on the design and implementation of the Visual Basic, VBScript, Jscript, and C# languages and on Visual Studio Tools For Office, and is a member of the C# language design team. When not writing or editing books about C#, he does his best to keep his tiny sailboat upright. He lives in Seattle 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 will not cast (even
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 in order to perform calculations. For example, you can make calculations on variables that you have declared.

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 broken down into three categories: unary, binary, and ternary, corresponding to the number of operands 1, 2, and 3, respectively. This section covers some of the most basic unary and binary operators. Introduction to the ternary operator appears 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.
Operators

Listing 3.2: Specifying Negative Values

```csharp
// National Debt to the Penny
decimal debt = -1523633233848.35M;
```

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

The subtraction example in Listing 3.3 is an example 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.

Listing 3.3: Using Binary Operators

```csharp
class Division
{
    static void Main()
```

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.


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 above, 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.
**BEGINNER TOPIC**

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

When an expression contains multiple operators it can be unclear what precisely 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 what 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.
Clearly, operators of higher precedence must execute before adjoining operators of lower precedence: in \( x + y \times z \) 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 left-to-right in C#. In an expression with three method calls such as \( A() + B() \times C() \), first \( A() \) is evaluated, then \( B() \), then \( C() \), then the multiplication operator determines the product, and then 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 above, the C++ specification allows an implementation broad latitude to decide the evaluation order of operands. When given an expression such as \( A() + B() \times C() \), 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 then 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

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 require localization. Composite formatting is preferred (refer to Chapter 1).

Guidelines

DO favor composite formatting over the addition operator for concatenating strings.

Using Characters in Arithmetic Operations

When introducing the char type in the preceding chapter, 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, contains a Unicode value of 0x33 (hexadecimal), which in base 10 is 51. The digit 4,
on the other hand, contains a Unicode value of \text{0x34}, or 52 in base 10. Adding 3 and 4 in Listing 3.5 results in a hexadecimal value of \text{0x167}, or 103 in base 10, which is equivalent to 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 results 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 \text{f} is three characters away from the letter \text{c}. You can determine this value by subtracting the letter \text{c} from the letter \text{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 results of Listing 3.6.

**Output 3.5**

| 3 |

**Special Floating-Point Characteristics**

The binary floating-point types, \text{float} and \text{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 \text{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 \text{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 upcoming Advanced Topic, Unexpected Inequality with Floating-Point Types.)

Note that internally the binary floating-point types actually store a binary fraction, not a decimal fraction. This means that “representation error” inaccuracies can occur with a simple assignment, such as double number = 140.6f. The exact value of 140.6 is the fraction 703/5, but the denominator of that fraction is not a power of two, and therefore, it cannot be represented exactly by a binary floating-point number. The value actually represented is the closest fraction with a power of two 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.
Chapter: Operators and Control Flow

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);
// Displays: 4.2 != 4.20000006258488
System.Console.WriteLine("(0) != {1}", decimalNumber, (decimal)doubleNumber1);

Trace.Assert((double)decimalNumber != doubleNumber1);
// Displays: 4.2 != 4.20000006258488
System.Console.WriteLine("(0) != {1}", (double)decimalNumber, doubleNumber1);

Trace.Assert((float)decimalNumber != floatNumber);
// Displays: (float)4.2M != 4.2F
System.Console.WriteLine("(float){0}M != {1}F", (float)decimalNumber, floatNumber);

Trace.Assert(doubleNumber1 != (double)floatNumber);
// Displays: 4.20000006258488 != 4.20000028610229
System.Console.WriteLine("(0) != {1}", doubleNumber1, (double)floatNumber);

Trace.Assert(doubleNumber1 != doubleNumber2);
// Displays: 4.20000006258488 != 4.2
System.Console.WriteLine("(0) != {1}", doubleNumber1, doubleNumber2);

Trace.Assert(floatNumber != doubleNumber2);
// Displays: 4.2F != 4.2D
System.Console.WriteLine("(0)F != {1}D", floatNumber, doubleNumber2);

Trace.Assert((double)4.2F != 4.2D);
// Display: 4.1999999999999996 != 4.2
System.Console.WriteLine("(0) != {1}", (double)4.2F, 4.2D);

Trace.Assert(4.2F != 4.2D);
// Display: 4.2F != 4.2D
System.Console.WriteLine("(0)F != {1}D", 4.2F, 4.2D);
```

Output 3.6 shows the results of Listing 3.7.
The `Assert()` methods alert the developer whenever their argument evaluates to `false`. However, of all the `Assert()` calls in this code listing, only half have arguments that evaluate to `true`. In spite of the apparent equality of the values in the code listing, they are in fact not equivalent due to the inaccuracies of a `float`.

**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 instead 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. Take Listing 3.10, for example.
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 relatively frequent, 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.

Listing 3.11: Using the \(+=\) Operator

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

This code, therefore, is equivalent to Listing 3.10.

Numerous other combination assignment operators exist to provide similar functionality. You can also use the assignment operator with subtraction, multiplication, division and the remainder operators (Listing 3.12 demonstrates).

Listing 3.12: Other Assignment Operator Examples

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

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

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 3.13: Increment Operator

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

**LISTING 3.14: Decrement Operator**

```csharp
lines = lines - 1;
lines -= 1;
lines--; 
```

---

**BEGINNER TOPIC**

**A Decrement Example in a Loop**

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

**LISTING 3.15: Displaying Each Character's Unicode Value in Descending Order**

```csharp
char current;
int unicodeValue;

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

do
{
    // Retrieve the Unicode value of current.
    unicodeValue = current;
    System.Console.Write("{0}={1}	", current, unicodeValue);
    // Proceed to the previous letter in the alphabet;
    current--;

} while(current>='a');
```

Output 3.9 shows the results of Listing 3.15.

**OUTPUT 3.9**

<table>
<thead>
<tr>
<th>char</th>
<th>Unicode Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>122</td>
</tr>
<tr>
<td>y</td>
<td>121</td>
</tr>
<tr>
<td>x</td>
<td>120</td>
</tr>
<tr>
<td>w</td>
<td>119</td>
</tr>
<tr>
<td>v</td>
<td>118</td>
</tr>
<tr>
<td>u</td>
<td>117</td>
</tr>
<tr>
<td>t</td>
<td>116</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
</tr>
<tr>
<td>q</td>
<td>113</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
</tr>
<tr>
<td>n</td>
<td>110</td>
</tr>
<tr>
<td>m</td>
<td>109</td>
</tr>
<tr>
<td>l</td>
<td>108</td>
</tr>
<tr>
<td>k</td>
<td>107</td>
</tr>
<tr>
<td>j</td>
<td>106</td>
</tr>
<tr>
<td>i</td>
<td>105</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
</tr>
<tr>
<td>g</td>
<td>103</td>
</tr>
<tr>
<td>f</td>
<td>102</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
</tr>
<tr>
<td>d</td>
<td>100</td>
</tr>
<tr>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>a</td>
<td>97</td>
</tr>
</tbody>
</table>
The increment and decrement operators are used to control how many times a particular operation is performed. Notice also that in this example, the increment operator is 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 causes 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 = {0} and count = {1}", result, 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 = {0} and count = {1}", result, count);
```
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 appears 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
{
    public static void Main()
    {
        int x = 123;
        // Displays 123, 124, 125.
        System.Console.WriteLine("{0}, {1}, {2}", x++, x++, x);
        // x now contains the value 125.
        // Displays 126, 127, 128
        System.Console.WriteLine("{0}, {1}, {2}", ++x, ++x, x);
        // x now contains the value 128.
    }
}
```

**Output 3.12**

| 123, 124, 125 |
| 126, 127, 128 |

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 in C++, the operands in an expression can be evaluated in any order, whereas in C# they are always evaluated left to right. 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 \( M(1, 2) \) or \( M(2, 1) \) at the whim of the compiler; C# will always call \( M(1, 2) \) because C# makes two guarantees: first, that the arguments to a call are always computed left to right, and second, that 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 usages 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 calculating 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` 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 3.43) 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 the table 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>
</table>
| if statement | `if(boolean-expression) embedded-statement` | if (input == "quit")
{   System.Console.WriteLine("Game end");
    return;
}

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| while statement | `while(boolean-expression) embedded-statement` | while(count < total)
{   System.Console.WriteLine("count = {0}", count);
    count++;
} |

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| do while statement | `do embedded-statement while(boolean-expression);` | do
{   System.Console.WriteLine("Enter name:");
    input = System.Console.ReadLine();
} while(input != "exit"); |

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Table 3.1: Control Flow Statements, Continued**

<table>
<thead>
<tr>
<th>Statement</th>
<th>General Syntax Structure</th>
<th>Example</th>
</tr>
</thead>
</table>
| for statement       | `for(for-initializer; boolean-expression; for-iterator) embedded-statement` | `for (int count = 1; count <= 10; count++)` {  
System.Console.WriteLine(  
    "count = {0}", 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;`                                                          |                                                                        |
### Table 3.1: Control Flow Statements, Continued

<table>
<thead>
<tr>
<th>Statement</th>
<th>General Syntax Structure</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch statement</td>
<td><code>switch</code>(governing-type-expression)</td>
<td><code>switch</code>(input)</td>
</tr>
<tr>
<td></td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>case &quot;exit&quot;:</td>
</tr>
<tr>
<td></td>
<td>case const-expression:</td>
<td>case &quot;quit&quot;:</td>
</tr>
<tr>
<td></td>
<td>statement-list</td>
<td>System.Console.WriteLine(</td>
</tr>
<tr>
<td></td>
<td>jump-statement</td>
<td>&quot;Exiting app....&quot;);</td>
</tr>
<tr>
<td></td>
<td>default:</td>
<td>break;</td>
</tr>
<tr>
<td></td>
<td>statement-list</td>
<td>case &quot;restart&quot;:</td>
</tr>
<tr>
<td></td>
<td>jump-statement</td>
<td>Reset();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>goto case &quot;start&quot;;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>case &quot;start&quot;:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetMove();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>break;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>default:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System.Console.WriteLine(</td>
</tr>
<tr>
<td></td>
<td></td>
<td>input);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>break;</td>
</tr>
<tr>
<td>break statement</td>
<td><code>break</code>;</td>
<td></td>
</tr>
<tr>
<td>goto statement</td>
<td><code>goto</code> identifier;</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>goto</code> case const-expression;</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>goto</code> default;</td>
<td></td>
</tr>
</tbody>
</table>
Introducing Flow Control

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 the table 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.

**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.
7.             "What is the maximum number " +
8.             "of turns in tic-tac-toe?" +
9.             "(Enter 0 to exit.): ");
10.        // int.Parse() converts the ReadLine() // return to an int data type.
11.        input = int.Parse(System.Console.ReadLine());
12.        if (input <= 0)  // Input is less than or equal to 0.
14.        else
15.            if (input < 9)  // Input is less than 9.
16.                System.Console.WriteLine("Tic-tac-toe has more than {0}" +
17.                    " maximum turns.", input);
18.            else
19.                if (input > 9)  // Input is greater than 9.
20.                    System.Console.WriteLine("Tic-tac-toe has fewer than {0}" +
21.                        " maximum turns.", input);
24.                        " has a max. of 9 turns.");
25.        }
26.     }
27. }
```
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 max. 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. Without indenting and without 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 \{0\}" +
        " maximum turns.", input);
else if (input < 9)
    System.Console.WriteLine(
        "Tic-tac-toe has less than \{0\}" +
        " maximum turns.", input);
else
    System.Console.WriteLine(
        "Correct, tic-tac-toe has a maximum " +
        " of 9 turns.");
```

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

Each if statement listing above omits the use of 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.
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 = 3.14*radius*radius;
            System.Console.WriteLine("The area of the circle is: {0}", area);
        }
        else
        {
            System.Console.WriteLine("{0} is not a valid radius.", radius);
        }
    }
}
```
Output 3.14 shows the results of Listing 3.24.

**OUTPUT 3.14**

```
Enter the radius of the circle: 3
The area of the circle is: 28.26
```

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 = 3.14 * radius * radius;
System.Console.WriteLine(  
    "The area of the circle is: {0}", area);
```

In C#, indentation is for code readability only. The compiler ignores it, and therefore, the previous code is semantically equivalent to Listing 3.26.

**Listing 3.26: Semantically Equivalent to Listing 3.25**

```csharp
if (radius >= 0)
{
    area = 3.14 * radius * radius;
}
System.Console.WriteLine(  
    "The area of the circle is: {0}", area);
```

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. In fact, the 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.

**Guidelines**

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

---

**ADVANCED TOPIC**

**Math Constants**

In Listing 3.25 and Listing 3.26, the value of pi as 3.14 was hardcoded—a crude approximation at best. There are much more accurate definitions for pi and E in the `System.Math` class. Instead of hardcoding a value, code should use `System.Math.PI` and `System.Math.E`.

**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 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 not only defines a scope, it also defines 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. (Though the rule is relaxed somewhat for methods; two methods may have the same name in a declaration space provided that they have
Code Blocks, Scopes, and Declaration Spaces

A code block not only defines a scope, it also defines a local variable declaration space. That is to say, within a block a local 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 by its name; the local is said to be “out of scope” outside the block.

In short: 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 is “out of scope” when its name is used later on in the method. To avoid the error, you must declare the variable outside the if statement.

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): ");
        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

```
...\Program.cs(18,26): error CS0103: The name 'message' does not exist in the current context
```
The declaration space throughout which a local’s name must be unique includes 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 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; an attempt to refer to the local variable before its declaration will fail to find the local because the local 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 is in scope throughout the entire block in which it is declared, but it is illegal to refer to the local before its declaration. That is, the attempt to find the local succeeds and the usage is then 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.
```
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 it 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 common in C and C++. In C++, Listing 3.29 is allowed.

```csharp
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 a glance this 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.
## Operators and Control Flow

### Table 3.2: Relational and Equality Operators

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

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, `!=`.

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 {0} quit!!", currentPlayer);
    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 only rarely used for Boolean logic, for reasons which we discuss below.

**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 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((hourOfTheDay > 23) || (hourOfTheDay < 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 above, if `hourOfTheDay` has the value 33, `(hourOfTheDay > 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. 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.

---

4. The typical hours that programmers work.
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.");
```

**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 results.

Listing 3.34: Using the Logical Negation Operator

```csharp
bool valid = false;
bool result = !valid;
// Displays "result = True".
System.Console.WriteLine("result = \{0\}", result);
```

Output 3.16

result = True
To begin, 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:

```
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 is an example of how to use the conditional operator. The full listing of this program appears in Appendix B.

**Listing 3.35: Conditional Operator**

```csharp
public class TicTacToe
{
    public 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 this, 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 consistently typed, 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 still 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.” Its form is:

\n
expression1 ?? expression2;

\n
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 is an example of how to use 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 coalescing operator “chains” nicely; 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; the null coalescing operator works on both operands of nullable value types and reference types.

**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, to $2^7$ on the left, as shown in Figure 3.1.

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^7$</td>
<td>$2^6$</td>
<td>$2^5$</td>
<td>$2^4$</td>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
</tbody>
</table>

**Figure 3.1: Corresponding Placeholder Values**

In many scenarios, particularly when dealing with low-level or system services, information is retrieved as binary data. In order to manipulate these devices and services, you need to perform manipulations of binary data.
As shown 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.

\[
\begin{array}{ccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

\[7 = 4 + 2 + 1\]

**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 “twos complement” notation. This is so 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 than 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.

\[
\begin{array}{ccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

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

**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. 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 ones and not zeroes. The shift operators are >> and <<, 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 3.37, you right-shift the binary representation of the number -7 by two locations.

Listing 3.37: Using the Right-Shift Operator

```csharp
int x;
x = (-7 >> 2); // 11111111111111111111111111111001 becomes
                // 11111111111111111111111111111110
// Write out "x is -2."
System.Console.WriteLine("x = {0}.", x);
```

Output 3.17 shows the results of Listing 3.37.

**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, do not use bit shift operators for multiplication or division. This might have been true in 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.
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.”

Therefore, the bitwise AND of the two values in Figure 3.4 would be 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 3.38 demonstrates how to use these bitwise operators. The results of Listing 3.38 appear in Output 3.18.

**Listing 3.38: 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 = {0} \nor = {1}\nxor = {2}",
    and, or, xor);
```

**Output 3.18**

```
and = 4
or = 15
xor = 11
```
In Listing 3.38, 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.

In order to convert a number to its binary representation, you need to iterate across each bit in a number. Listing 3.39 is an example of a program that converts an integer to a string of its binary representation. The results of Listing 3.39 appear in Output 3.19.

LISTING 3.39: Getting a String Representation of a Binary Display

```csharp
public class BinaryConverter
{
    public 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;
        }
        System.Console.WriteLine();
    }
}
```

OUTPUT 3.19

Enter an integer: 42
0000000000000000000000000000000000000000000000000000000000101010
Notice that within each iteration of the `for` loop (discussed later in this chapter), you 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, you can determine whether the bit is set. If the mask test produces a nonzero result, you write `1` to the console; otherwise, `0` is written. In this way, you 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 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`.

**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 3.40 demonstrates.

**Listing 3.40: 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 = {0} or = {1} xor = {2}",
    and, or, xor);
```

The results of Listing 3.40 appear in Output 3.20.

**Output 3.20**

```
and = 4
or = 15
xor = 11
```

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. ~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. In order 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. It isn’t until the condition is reevaluated “at the top of the loop” that the loop exits. The Fibonacci calculator shown in Listing 3.41 demonstrates the while loop.

Listing 3.41: while Loop Example

```csharp
class FibonacciCalculator
{
    static void Main()
    {
```
decimal current;
declaration previous;
declaration temp;
declaration 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("The Fibonacci number following this is {0}",
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 3.41, 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 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 3.42 is taken from the tic-tac-toe program.

**Listing 3.42: do/while Loop Example**

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

    // Request a move from the current player.
    System.Console.Write(
        "\nPlayer {0}: Enter move:", currentPlayer);
    input = System.Console.ReadLine();

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

In Listing 3.42, you always 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:

```csharp
do
    statement
while (condition);
```

As with all the control flow statements, a code block is generally used as the single statement in order 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 3.43 shows the for loop used to display an integer in binary form. The results of this listing appear in Output 3.21.

Listing 3.43: Using the for Loop

```csharp
public class BinaryConverter
{
    public 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;
        }
    }
}
```

Output 3.21

Enter an integer: -42
1111111111111111111111111111111111111111111111111111111111010110

Listing 3.43 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:

```c
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 3.43, 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 one.
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. for(;;){ ... } is perfectly valid; although there still needs to be a means to escape from the loop to avoid executing infinitely. (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 3.44.

Listing 3.44: for Loop Using Multiple Expressions

```csharp
for(int x=0, y=5; ((x<=5) && (y>=0)); y--, x++)
{
    System.Console.Write("{0}{1}{2}\t",
        x, (x>y? ' ':''), y);
}
```

The results of Listing 3.44 appear in Output 3.22.

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

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)
{
```
Control Flow Statements, Continued

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 3.45.

**Listing 3.45: Determining Remaining Moves Using the foreach Loop**

```csharp
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("{0} ", cell);
            }
        }
    }
}
```

Output 3.23 shows the results of Listing 3.45.

**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.

It is important to 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 than it did in previous versions; the difference is only apparent 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 3.46.

**Listing 3.46: Checking the Player’s Input with an if Statement**

```
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(
        "\nERROR: Enter a value from 1-9."
        + "Push ENTER to quit");
```
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 above has two switch sections; Listing 3.47 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, earlier the guideline was to avoid omitting braces in general. One exception is to omit braces for case and break statements because they serve to indicate the beginning and end of a block.

Listing 3.47, with a switch statement, is semantically equivalent to the series of if statements in Listing 3.46.

Listing 3.47: Replacing the if Statement with a switch Statement

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;
Chapter 3: Operators and Control Flow

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 3.47, 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 unintended 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.
Jump Statements

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, 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 3.48 examines the foreach loop from the tic-tac-toe program.

Listing 3.48: 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
        };
    }
}
Chapter 3: Operators and Control Flow

// 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 {0} was the winner", winner);

Output 3.24 shows the results of Listing 3.48.

**OUTPUT 3.24**

Player 1 was the winner

Listing 3.48 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...
Jump Statements

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 0000000000000000 << 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 0000000000000000100. Listing 3.49 uses |= so that previous moves are combined with the current move.

**Listing 3.49: Setting the Bit That Corresponds to Each Player’s Move**

```csharp
int shifter; // The number of places to shift
  // over in order to set a bit.
int position; // The bit which 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 one to use currentPlayer as an
// index in a 0-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 3.48.

**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 3.50 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 3.50.

**Listing 3.50: 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.Write("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 3.50, 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, and this compromises readability. In Listing 3.51, 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 3.51: Replacing a continue 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 3.52, if the /out option is set, code execution jumps to the default case using the goto statement; similarly for /f.

**Listing 3.52: Demonstrating a switch with goto Statements**

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

    foreach (string option in args)
    {
```
Chapter : Operators and Control Flow

```csharp
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 3.52.

**Output 3.26**

```bash
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();` and then the statement `goto myLabel;` would branch to the labeled statement. Fortunately, C# prevents using `goto` to branch into a code block; it may only be used to branch within a code
block or to an enclosing code block. By making these restrictions, C# avoids most of the serious goto abuses possible in other languages.

In spite of the improvements, using 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.

### Guidelines

**AVOID** using goto.

---

**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.

### Language Contrast: C++—Preprocessing

Languages such as C and C++ use a 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 3.4: Preprocessor Directives

<table>
<thead>
<tr>
<th>Statement or Expression</th>
<th>General Syntax Structure</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>#if directive</td>
<td>#if preprocessor-expression code #endif</td>
<td>#if CSHARP2 Console.Clear(); #endif</td>
</tr>
<tr>
<td>#elif directive</td>
<td>#if preprocessor-expression1 code #elif preprocessor-expression2 code #endif</td>
<td>#if LINUX ... #elif WINDOWS ... #endif</td>
</tr>
<tr>
<td>#else directive</td>
<td>#if code #else code #endif</td>
<td>#if CSHARP1 ... #else ... #endif</td>
</tr>
<tr>
<td>#define directive</td>
<td>#define conditional-symbol</td>
<td>#define CSHARP2</td>
</tr>
<tr>
<td>#undef directive</td>
<td>#undef conditional-symbol</td>
<td>#undef CSHARP2</td>
</tr>
<tr>
<td>#error directive</td>
<td>#error preproc-message</td>
<td>#error Buggy Implementation</td>
</tr>
<tr>
<td>#warning directive</td>
<td>#warning preproc-message</td>
<td>#warning Needs code review</td>
</tr>
<tr>
<td>#pragma directive</td>
<td>#pragma warning</td>
<td>#pragma warning disable 1030</td>
</tr>
<tr>
<td>#line directive</td>
<td>#line org-line new-line #line default</td>
<td>#line 467 &quot;TicTacToe.cs&quot; ... #line default</td>
</tr>
<tr>
<td>#region directive</td>
<td>#region pre- proc-message code #endregion</td>
<td>#region Methods ... #endregion</td>
</tr>
</tbody>
</table>

**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 1.2 compilers, you use a preprocessor directive to exclude C# 2.0-specific code.
when compiling with a 1.2 compiler. You can see this in the tic-tac-toe example and in Listing 3.53.

**Listing 3.53: Excluding C# 2.0 Code from a C# 1.x Compiler**

```csharp
#if CSHARP2
    System.Console.Clear();
#endif
```

In this case, you call the `System.Console.Clear()` method, which is available only in 2.0 CLI and later versions. Using the `#if` and `#endif` preprocessor directives, this line of code will be compiled only if the preprocessor symbol `CSHARP2` 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 3.54.

**Listing 3.54: Using `#if`, `#elif`, and `#endif` Directives**

```csharp
#if LINUX
    ...
#elseif 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 3.55.

**Listing 3.55: A `#define` Example**

```csharp
#define CSHARP2
```
The second method uses the `define` option when compiling for .NET, as shown in Output 3.27.

**OUTPUT 3.27**

```
> csc.exe /define:CSHARP2 TicTacToe.cs
```

Output 3.28 shows the same functionality using the Mono compiler.

**OUTPUT 3.28**

```
>mcs.exe -define:CSHARP2 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 warning, respectively. Listing 3.56 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 3.56 appear in Output 3.29.

**LISTING 3.56: Defining a Warning with #warning**

```
#define 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 3.57).

**Listing 3.57: Using the Preprocessor `#pragma Directive to Disable the #warning Directive**

```
#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 3.58.

**Listing 3.58: Using the Preprocessor `#pragma Directive to Restore a Warning**

```
#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, as this appears when you elect to generate XML documentation using the `/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. In addition, the `nowarn` option affects the entire
compilation, and 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**

```
> 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 3.59, the actual line numbers within the file appear on the left.

**Listing 3.59: The #line Preprocessor Directive**

```csharp
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 shown in Output 3.31.

**Output 3.31**

```
Performing main compilation...
...\tictactoe.cs(113,18): warning CS1030: #warning: '"Same move allowed multiple times."
Build complete -- 0 errors, 1 warnings
```

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 the one in the Microsoft Visual Studio .NET IDE, 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. Again, Listing 3.60 shows the tic-tac-toe program as an example.

**Listing 3.60: A `#region` and `#endregion` Preprocessor Directive**

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

#if CSHARP2
    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(" {0} {1}", 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
...  
```  

One example of how these preprocessor directives are used is with Microsoft Visual Studio .NET. Visual Studio .NET 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).
Chapter : Operators and Control Flow

Figure 3.5: Collapsed Region in Microsoft Visual Studio .NET

SUMMARY

This chapter began with an introduction to the C# operators related to assignment and arithmetic. Next, you 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.

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 typeof(T) checked(x) unchecked(x) default(T)</td>
</tr>
<tr>
<td></td>
<td>delegate{} ()</td>
</tr>
<tr>
<td>Unary</td>
<td>+ - ! ~ ++x --x (T)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 the program, you can see one way in which you can combine all that you have learned into a complete program.
Index

! logical negation operator, 118–119
!=, <, <=, => relational operators, 386
&&, || logical Boolean operators, 116–118, 122–126
with flag enums, 365
() (cast/conversion) operators, 61–64, custom conversion operators, 391–393
%, *, +, -, / (arithmetic) operators, 87
+ (string) operator, 90–91
% *, +, -, / operator overloading 387–389
++/- (increment/decrement) operators, 97–102
+, - (unary plus/minus) operators, 86–87
+, +=, -= delegate operators, 540–542
= assignment operators, 15
?: (conditional) operators, 119–120
?? (null coalescing) operators, 120–121
@ characters, 8, 47
[ ] (square brackets), 67–75 attributes, 687–692
indexers, 655–657
\ (escape sequence), 44
\n (newline) characters, 46, 50
^ (exclusive OR) operators, 118, 122
} (curly braces), 2, 10, 110–114
| (OR) operators, 116, 117, 122
constraints, 473–474
with flag enums, 365
~ (bitwise complement) operators, 127, 64
Abort() method, 740
aborting threads, 741–743
abstract classes inheritance, 302–308
compared to interfaces, 337
abstract members, 302, 303–304, 305
access modifiers, 227–229, 397–398
internal, 396–398
private, 227–229, 397–398
protected, 285–286, 397–398
protected internal, 398
on property getters/setters, 239–240
on classes, 397
array accessors, 74
Active Template Library. See ATL
Add() method, 249, 352, 473, 568–569, 641
add/remove event handlers, customizing, 558
aliases qualifiers addition (+) operators, 87, 387, 541
guidelines, 91
strings, 90–91
overloading, 387–389
delegate operators, 540–542
addresses, 862–872
advanced parameters, methods, 175–184
AggregateException, 547, 757–764, 768, 779, 798–800
aggregation for multiple inheritance, 287–290
algorithms
hill climbing, 798
mark-and-compact, 407
mark-and-sweep-based, 882
work stealing, 798
aliasing with using, 171, 401–402
allocating
data on call stacks, 868
virtual memory, 851
AllocExecutionBlock() method, 855
AllowMultipleAttribute parameter, 700
alternative statements, 107
AND (&& operator, 117–118, 122–126
with flag enums, 365

NOTE: Page references marked with an n are footnotes
anonymous methods, 495, 512–514
internals, 517–518
anonymous types, 56–57, 562–564, 566–568
array initialization, 570–571
constructors, 253–255
projecting to, 583
antecedent tasks, 753
APIs (application programming interfaces), 27
APM (Asynchronous Programming Model), 908–921
AppDomain, 762
application programming interfaces.
See APIs
Appointment class, 278
ArgumentException, 424
ArgumentNullException, 424
ArgumentOutOfRangeException, 424
arguments
command-line, passing, 173
methods, 161–162
named, 191
arithmetic (binary) operators, 87–96, 387–389
overloading, 387–389
arity, 460–461
ArrayList method, 350, 352
arrays, 67–82
access, 68
applying, 74–79
assigning, 68, 70–74
command-line options, 80
covariance, support, 488–489
declaring, 68, 69–70
errors, 72, 81–82
foreach loops, 571–572
instantiation, 70–74
jagged, 73, 75
length, 75–76
literal values, 71
methods, 77–79
parameters, 181–184
redimensioning, 78
runtime, defining size at, 72
strings, 79–81, 80–81
three-dimensional, 73
two-dimensional, 69, 72, 74–75
as operator, conversions, 310–311
AsParallel() method, 584
assemblies, 4. See also libraries
CLI, 887–890
metadata, viewing, 678. See also reflection
targets, modifying, 394–395
versioning, 889
well-formed types, referencing, 393–398
Assert() method, 95
assigning
arrays, 68
indexes, 657–658
null to strings, 54
pointers, 866–869
text, 42
variables, 13, 15–16
assignment operators
applying, 548
binary operators, combining with, 389
compound, 96–103
events, 541
associating
classes, 259
data types, 57
relationships, 217
XML comments, 403–405
associativity, 88
async keyword, 777–781, 937–942
asynchronous delegate invocation, 921–924
asynchronous programming, 732
high-latency operations, 772–777
lambda expressions, 782–783
methods, customizing, 783–786
models, 908–921
System.Threading.Thread class, 737–739
Task-based Asynchronous Pattern (TAP), 770–794
task-based aspiration, 848–849
asynchronous tasks, 745–764
ATL (Active Template Library), 287
atomic operation, 734, 745, 829
attributes, 677, 688–714
backward compatibility, 712
constructors, initializing, 694–699
customizing, 692–693, 893
FlagsAttribute class, 367, 701–702
IndexerNameAttribute, 657
interfaces, comparing, 337–338
named parameters, 700–714
naming, 692
predefined, 703
searching, 693–694
serialization, 706–714
System.AttributeUsageAttribute class, 699–700
System.ConditionAttribute class, 703–705
System.ObsoleteAttribute class, 705–706
System.SerializableAttribute class, 438, 713–714
automatically shimmmed interfaces, 848
automatically implemented properties, 232–234
Average() method, 609
await keyword, 741, 777–781, 937–942
background worker patterns, 928–932
backing field declarations, 232, 244
backslash (\) escape sequence, 44
Base Class Library. See BCL
base classes
base member, 300–301
finalizers in, 412
inheritance, overriding, 290–302
new modifier, 295–299
refactoring, 279
sealed modifiers, 299–301
virtual modifiers, 290–295
base members, 300–301
base types, 212
BCL (Base Class Library), 25, 34, 743, 885, 892, 894
BeginGetResponse() method, 908
BeginX() method, 908
behaviors
boxing, 349–357
dynamic objects, 716–718
implementation-defined, 101
polymorphism, data types, 310
best practices, thread synchronization design, 827–829
binary floating-point types, 37, 92–96
binary (arithmetic) operators, 87–96, 387–389
Index

BinarySearch() method, 643, 644
BinaryTree<T> class, 462–463, 658
binding
dynamic objects, 719–720
late, 893
methods, 714
runtime, XML elements, 719–720
bits, 121–122
bitwise complement () operators, 127, 644
compound assignment operators, 121–127
operators, 121–127, 140
blocks
catch, 199–200, 203, 204, 428–432
code blocks ({}), 110–114
finally, 199–200
try, 197
unsafe, 863, 864
Boolean expressions, 107, 114–121
number conversions, 64
types, 43
boxing
avoiding, 356–357
value types, 349–357
break () method, 803
break statements, 139–141
breaking parallel loops, 803–804
brittle base classes, 295
BubbleSort() method, 496–497
buffers, overrun, 76, 883
bugs, runtime performance, 883
building custom collections, 635. See also collections, customizing
bytes, 121–122
C language, 1
pointers, declaring, 865
C++ language, 1, 890
arrays, declaring, 69
buffer overrun, 76
delete operator, 216
deterministic destruction, 418, 882
evaluation order of operands, 90
global methods, 164
global variables/functions, 256
header files, 168
implementation-defined behavior, 101
implicit overriding, 292
implicitly typed variables, 565
local variable scope, 114
Main method, 10
methods, calling, 295
multiple inheritance, 287
operator errors, 115
operator-only statements, 87
pointers, declaring, 865
preprocessors, 145
pure virtual functions, 305
short data types, 35
string concatenation at compile time, 48
structure, defining with public members, 348
switch statements, 138
templates, 466
void, 55
C# without generics, 444–449
caches, avoiding repeated, 590
calculating
compound assignment operators, 96–103
financial, 36
operators, 86. See also operators pi, 794–795
values, bytes, 122
callers, 157
variables, matching with parameter names, 176
calling
APM methods, 915–921
constructors, 245
methods, 11, 156–163, 295
avoiding boxing, 356–357
statements, 163
P/Invoke
APIs, 861–862
external functions, 858–861
SelectMany() method, 604–606
sites, 175
stacks, 175, 868
Task.ContinueWith method, 752, 789
CamelCase, 7, 15
Cancel() method, 766
CancellationToken class, 764, 767
CancellationTokenSource class, 767
cancelling
parallel loops, 800
PLINQ queries, 807
tasks, 764–770
Capacity() method, 639
capitalizing variables, 15
capturing
loop variables, 521–522
variables, 518
cartesian products, 598, 631
CAS (code access security), 884
permissions, 686
case-sensitivity, 2
cast (()) operators, 61, 391–392
casting
inheritance, chaining, 282–283
multicast delegates, 533
operators, defining, 283
types, 66–66, 281–282
catch blocks, 199–200, 203, 204, 428–432
catch clause, 779
catching exceptions, 196–197, 426–427
categories of types, 57–60, 340
centralizing initialization, 252–253
chaining
constructors, 251–253
inheritance
casting, 282–283
exceptions, 437
multicast delegates, 544
change() method, 942
char data types, 92
characteristics of parameter arrays, 183
characters
@, 47
escape, 45
newline (\n), 46, 50
operators, applying, 91–92
Unicode, 43–46, 98
checking
conversions, 62–64, 440
for null, 538–539
types, 883
child collections, formatting, 602
Church, Alonzo, 513
CIL (Common Intermediate Language), 24, 876, 877, 894
boxing, 350
CLI, 890. See also CLI
CIL (Common Intermediate Language) (cont’d)
dynamic objects, 718–719
empty catch blocks, 432
events, 556–557
extension methods, 266
generics, 490–491
  best practices, 452–453
HelloWorld output, 29–30
ILDASM, 28–29
indexers, 657
iterators, defining, 661
machine code, compilation to, 879–880
objects
  deriving, 309
  initializers, 248
outer variables, implementing, 520–521
properties, 243–244
runtime, 881–886
Stack<T> class, 490
System.SerializableAttribute class, 713–714
  circumventing encapsulation, 883
class
  keyword constraints, 468–469
classes, 209–210
  abstract, 302–308, 337
  access modifiers, 227–229, 397
  associating, 259
  base. See also base classes
    base member, 300–301
    new modifier, 295–299
    overriding, 290–302
    refactoring, 279
    sealed modifiers, 299–301
  constructors, 244–255
  anonymous types, 253–255
  chaining, 251–253
  common initializers, 249
  declaring, 245–246
  defaults, 247
  finalizers, 249–250
  object initializers, 247–248
  overloading, 250–251
  declaring, 8, 213–216
  definitions, 214
  encapsulation, 215, 267–269
  extension methods, 265–266
  finalizers, 416
generics, 450–452
  hierarchies, 212, 324
inheritance, 277. See also inheritance
  initializers, 247–248
  instances
    fields, 217–219
    methods, 219–220
  instantiation, 213–216
  interfaces
    comparing, 336–337
    conversions, 326
  Java inner classes, 272
  locations, 408
  members, 216
  memory, garbage collection, 408
  methods, 157. See also methods
    naming, 4, 8
    nested, 269–272
  object-oriented programming, 210–213
  partial, 272–273, 272–276
  properties, 229–244
    access modifiers on getters/setters, 239–240
    guidelines, 234–235
    parameter values, 242–244
    read-only/write-only, 237–239
    validation, 236–237
  virtual fields, 240–242
  sealed, inheritance, 290
  static members, 255–265
  this keyword, 220–227
  types, constraints, 467–468
  cleanup
    collection interfaces after iteration, 575
    resources
      APM, 914
      using statements, 575
    well-formed types, 410–419
  CLS (Common Language Specification), 25, 877, 891–892, 895
  code, xxiii
    access security, 25, 884
    blocks, 110–114
    machine, 24, 876, 879–880
    managed, 24, 881
    native, 24
    preprocessor directives, excluding/including, 146–147
    rebase, 394
  runtime performance, 885–886
  unsafe, 845–846, 863–864, 867, 872–873
  whitespace, formatting, 13
collections
  child, formatting, 602
classes, 638–655
  concurrence, 835–837
  customizing, 635
dictionaries, 636, 646–650, 696
  empty, 659–660
  filtering, 614
dictionaries, 637
  indexers, 655–659
  finalizers, 568–571
  interfaces, 249, 561–562, 636–638
  iterators, 660–674
compiling, 671–672
  creating multiple in single classes, 673–674
defining, 661
elements of, 666–667
  recursive, 669
  state, 664–666
  syntax, 661–662
  yield break statements, 670–671
  yield return statements, 674
  yielding values, 662–664
linked lists, 654–655
  lists, 639–641, 643–644
  multiple items, searching, 645
  projecting, 614
  queues, 654
  sorting, 641, 652
  stacks, 652–654
total ordering, 643
  yield return statements, placing in loops, 667–669
Index

collisions, type names, 158
COM (Component Object Model)
DLL registration, 890
STAThreadAttribute class, 842–843
combining assignment operators with binary operators, 389
command-line arguments, passing, 173
array options, 80
CommandLine class, 270
CommandLineAliasAttribute class, 694
CommandLineInfo class, 681, 684, 688, 689
CommandLineSwitchRequiredAttribute class, 692, 694
commands
preprocessor directives, 145
xcopy, 889
commas (,), 165
comments
code, 21–23
well-formed types, 402–407
XML, 678
Common Intermediate Language. See CIL
Common Language Infrastructure. See CLI
Common Language Runtime. See CLR
Common Language Specification. See CLS
Common Type System. See CTS
Compare() method, 495
CompareTo() method, 323, 641
generics, 463
comparison operators, 386–387
ComparisonHandler delegate, 499
comparisons
equality, requirements of, 651
interfaces
attributes, 337–338
classes, 336–337
compilers, 878
installing, 897–899
paths, configuring, 898
compiling
applications, 3–4
checked/unchecked conversions, 63
just-in-time compilation, 24, 466, 879
keywords, 4. See also keywords
machine code, 879–880
static compilation versus dynamic programming, 720–721
components, 1
composite Formatting, 20
compound assignment operators, 96–103
Concat() method, 608
concatenation, strings
addition (+) operators, 90–91
at compile time, 48
concrete classes, 302, 305
concurrency, 732, 835–837
conditional logical operators, 389
conditional (?:) operators, 119–120
consequence statements, 107
consistency, integers, 35
console executable assemblies, 394
ConsoleListControl class, 316–317, 320
consoles
input, 17–19
output, 19–21
synchronization, 920–921
Console.WriteLine() method, 340, 352, 871
const values, 267
constants
expressions, 102–103
fields, declaring, 267
math, 112
public, 268
constraints, 462–476
class keyword, 468–469
class types, 467–468
constructors, 470
defaults, 474–476
inheritance, 471–472
interfaces, 465–467
limitations, 472–473
multiple, 469
specifying, 479–480
struct keyword, 468–469
constructors
attributes, initializing, 694–699
base, specifying base, 301–302
classes, 244–255
anonymous types, 253–255
chaining, 251–253
common initializers, 249
declaring, 245–246
defaults, 247
finalizers, 249–250
object initializers, 247–248
overloading, 250–251
constraints, 470, 474–476
exceptions, 418, 437
generics, defining, 457
static, 261–262
Contains() method, 643
context
switches, 732
synchronization, 788–790
contextual keywords, 6, 672–673
cancellation
passing style. See CPS
query expressions, 629–630
tasks, 789
continue statement, 141–143
Continuously() method, 753, 775, 919–920
covariance
enabling, 485–488
generics, 481–489
control flow
asynchronous tasks, 751
await keyword, 792–794
statements, 85, 103–110, 127–139
do/while loops, 127–129
exception-handling, 198
foreach loops, 133–136
if statements, 107–110
for loops, 120–133
switch statements, 135–139
while loops, 127–129
tasks, 780
conversions
as operator, 310–311
Boolean types, numbers, 64
boxing, 349–357
checked, 62–64, 440
CIL, 879
classes, 326
covariant, 482. See also covariance
customizing, 283–283
data types, 60–67
enums, strings, 362–364
exception handling without, 207
explicit, 392
implicit, 64, 392
objects, deriving, 282
operators, 391, 393
strings, 65
conversions (cont’d)
  types  
  checking, 883
  without casting, 65–66
  unchecked, 62–64, 440
    cooperative cancellation, 764
    coordinates, 388–389, 395
  CopyTo() method, 638
  cores, 730n2
  Count() method, 585–586, 609,
  622
  Count property, 638
  CountdownEvent class, 835
  counting items, 585–586
  CountLines() method, 157
  covariances  
  enabling, 483–485
  generics, 481–489
  support, 488–489
  CPS (continuation passing style),
  911–913, 915
  CPUs (central processing units),
  730
    LINQ queries, running in paral-
  lel, 584
  CTS (Common Type System), 25,
  877, 891, 895
  curly braces ({}), 2, 10
  code blocks, 110–114
  customizing  
  add/remove handlers, 558
  attributes, 692–693, 893
  collections, 635. See also
  collections
    classes, 638-655
    dictionaries, 646–650
    empty, 659–660
    indexers, 655–659
    interfaces, 636–638
    iterators, 660–674
    linked lists, 654–655
    List<T> class, 639–641
    queues, 654
    searching items, 645
    searching List<T>, 643–644
    sorting, 641, 652
    stacks, 652–654
    total ordering, 643
    conversions, 283–283
    dictionaries, equality, 649–650
    dynamic objects, 721–724
    events, implementing, 558–559
    exceptions, defining, 435–438
    LINQ, providers, 609
    methods, asynchronous,
    783–786
    serialization, 708–710
    synchronization contexts, 790
    cycles, processors, 728, 741
  data  
    managed, 881
    persistence to files, 224
    retrieval from files, 225
    types, 14, 33–34
  DataStorage class, 403
  de-allocating objects, 882. See also
  allocating
  deadlocks, 736, 745, 827–828
  decimal types, 36–37
  declaration spaces, 112–114
  decrement (--) operators, 97–102
  default  
    keyword, 68
    operator, 348, 458
    default constructors, 247
    default constructor constraints,
    474–476
  deferred execution  
    implementing, 623
    query expressions, 619
    standard query operators,
    586–590
  #define preprocessor directive,
  147–148
  delegate keyword, 542
  delegates, 495
  asynchronous delegate invoca-
  tion, 921–924
  data types, 498–500
  events, 554–555
  operators, 540–542
  sequential invocation, 542
  expression trees, 527–528
  generics, declaring types, 552
  instances, returning, 539
  instantiation, 500–502
  internals, 503–506
  invoking  
    events, 537–538
    pass-by references, 547–548
    returning methods, 547–548
    thread-safe, 539–540
  multicast, 533
  internals, 542–544
  observer patterns, 534–548
  overview of, 496–506
  P/Invoke, 862
  passing, 510
  sequence diagrams, 545
  synchronous, 747
  syntax, 502
  System.Func/System.Action, 514–530
  types, declaring, 500
  unsafe code, executing via,
  872–873
  delete (C++) operator, 216
  delimited comments, 22
  denominators, 35
  deployment, xcopy, 889
  Dequeue() method, 654
  dereferencing pointers, 869–871
  deriving
    inheritance, 278–290
    casting between types,
    281–282
    customizing conversions,
    283–283
    private access modifiers,
    284–285
    protected access modifiers,
    285–286
  System.Object class, 308–309
  types, 212
  deserialization, 711. See also
  serialization
  deterministic destruction (C++),
  418, 882
  deterministic finalization, 412–415
  device drivers, 886
  dictionaries
    collections, 636, 646–650, 696
    equality, customizing, 649–650
  Dictionary<T> class, 646–650
  directives. See also commands
  preprocessor, 145–152
  code editors, 151–152
  errors/warnings, 148–150
  #pragma preprocessor direc-
  tive, 149
  specifying line numbers, 150
  symbols, 147–148
  using, 168–172
  DirectoryCountLines() method,
  185, 187, 191
  Directory.GetFiles() method,
  617
DirectoryInfoExtension.
Copy() methods, 260
DirectoryInfo.GetFiles() method, 597
disabling parallelism, 802
disambiguating multiple Main() methods, 174
Dispose() method, 6n6, 413, 416, 575
tasks, 770
distinct members, 631–632
Distinct() method, 608, 613
distribution, APM parameters, 911
division (/) operators, 87
dll (Dynamic Link Library), 4
COM registration, 890
do/while loops, control flow statements, 127–129
documents
saving, 706–708
XML, 402–407, 678
domains, applications, 887
dot (.) operator, 871
DotGNU, 878
dotPeek, 30
double quotes ("), 47
double.TryParse() method, 207
downloading .NET (Microsoft), 897–899
drivers, devices, 886
duck typing, 576
dump() method, 325
Dynamic Link Library. See DLL
dynamic objects
behaviors/principles, 716–718
binding, 719–720
CIL, 718–719
customizing, 721–724
programming, 714–724
reflection, invoking, 714–716
static compilation versus, 720–721
dynamic programming, 677
EAP (Event-based Asynchronous Pattern), 924–927
editors, code, 151–152
elements
deleting, 641
indexes, retrieving, 640
XML, runtime binding, 719–720
#endif preprocessor directive, 146–147
else clauses, 107
#else preprocessor directive, 146–147
empty catch blocks, 432. See also catch blocks
empty collections, 659–660
empty memory, retrieving, 246
Empty<T> method, 659
encapsulation, 211, 215
APIs, 859–860
circuitventing, 883
classes, 267–269
publishers, 549–550
subscribers, 548–549
of types, 396
encryption. See also security serialization, 708
strings, 804
EndGetResponse() method, 908
#endif preprocessor directive, 146–147
#endregion preprocessor directive, 151–152
EndX() method, 908
Enqueue() method, 654
Enter() method, 353
EntityBase, EntityBase<T> class, 468, 471
EntityDictionary, EntityDictionary<T> class, 469, 476
enumeration, values, 702
enums, 358–368
Enum.Parse() method, 363, 680
flags, 364–368
string conversions, 362–364
type compatibility, 361–362
equality, 93. See also inequality dictionaries, customizing, 649–650
operators, 115–116
structural, delegates, 516–517
Equals() method, 349, 651
overriding, 376–385
equals (==) operator, 386
#error preprocessor directive, 148–150
errors
arrays, 72, 81–82
buffer overrun, 76
handling
platform interoperability/unsafe code, 854–856
sequential notification, 544–547
using statements, 575
infinite recursion, 186
methods, 194–208
namespace alias qualifiers, 401
operators, 115
preprocessor directives, 148–150
reporting, 204–207
ranging, 37
trapping, 195–201
Windows Error Reporting, 425
escape sequences, 44
verbatim string literals, 47
Etch A Sketch, 444
evaluation, 89
order of operands, 90
Event-based Asynchronous Pattern. See EAP
event keyword, 550, 551
events, 533–534
CIL code, 556–557
code conventions, 552–554
declaring, 550–551
delegates, 554–555
invoking, 537–538
operators, 540–542
sequential invocation, 542
generics, 554–555
handlers
adding, 846–848
removing, 846–848
implementing, customizing, 558–559
internals, 556–558
multicast delegates, 534–548
notifications
firing, 553
multiple threads, 826–827
null, checking for, 538–539
publishers
connecting subscribers and, 536–537
defining, 536
resetting, 831–837
WinRT, 846–848
examples of iterators, 666–667
exceptions
AggregateException, 547, 757–764, 768
parallel loop exception handling, 798–800
ArgumentException, 424
exceptions (cont’d)
ArgumentNullException, 424
ArgumentOutOfRangeException, 424
asynchronous high-latency
operations, 775–776
catching, 196–197, 426–427
classes, inheritance, 201
common exception types, 202
constructors, 418, 437
defining custom exceptions,
435–438
defining multiple exception types,
424–425
deleting, 432
hiding, 432
InnerExceptions property, 760
InvalidAddressException, 435
InvalidOperationException, 424
NotImplementedException, 321
NullReferenceException, 424
OperationCanceledException, 427
FileNotFoundException, 427
OperationCanceledException, 427, 439
NotImplementedException, 321
HexStringException, 424, 538, 551, 659, 826
OutOfMemoryException, 433
reporting, 433
rethrowing, 206, 438–442
sequences, diagrams, 545
serializable, 438
specifiers, 427
SqlException, 437
StackOverflowException, 437
suffixes, 437
System.ComponentModel.Win32Exception, 854
System.FormatException, 198, 199
System.InvalidCastException, 393
System.Runtime.Serialization.SerializationException, 710
TaskCancelledException, 767, 768
ThreadAbortException, 741, 742
throwing, 195, 204–205, 321
arrays, 75
checked/unchecked conversions, 63
deserialization, 711
UnauthorizedAccessException, 438
unhandled, 195, 761–764
wrapping, 438–442
excluding code, preprocessor
directives, 146–147
exclusive OR (^) operator, 118, 122
execution
agents, 881
defered
implementing, 623
query expressions, 619
standard query operators,
586–590
delegates, unsafe code, 587–583
managed, 881
managing, 24–26
Exit() method, 353
explicit conversions, 392
explicit deterministic resource
cleanup, 216
explicit member implementation,
322–323
exponential notation, 40
expressions
Boolean, 107, 114–121
constants, 102–103
generics, 6n6
lambda, 495, 506–512
asynchronous programming,
782–783
expression trees, 524
internals, 517–518
lazy loading, 420
statements, 507–510
queries
continuation, 629–630
filtering, 623–624
flattening sequences, 630–622
grouping, 626–629
invoking methods, 632–634
1st clause, 625–626
LINQ, 561, 613
overview of, 614–632
projecting, 616–619
sorting, 624–625
trees, 496, 523–530
Extensible Markup Language.
See XML
externals methods, 265–266
IEnumerable<T> interface,
562
inheritance, 287
interfaces, 330–331
Reactive Extensions library, 729
external functions
declaring, 849–850
P/Invoke, calling, 858–861
expression trees, 496, 523–530
f-reachable queues, 416
factory interfaces, 475
factory methods, 461
FCL (Framework Class Library),
892, 895
features, adding, 278. See also
inheritance
Fibonacci numbers/series, 128,
351
files
backing, declaring, 232, 244
constants, declaring, 267
guidelines, 234–235
instances, 217–219
readonly modifiers, declaring,
268, 269
static, 256–258
virtual, properties, 240–242
volatile, declaring as, 823–824
FileInfo object, 625–626
files
data
persistence to, 224
retrieval from, 225
headers, 168
loading, 224
metadata, 892–893
references, assemblies, 889
storing, 224
XML, 23, 402–407. See also XML
FileSettingsProvider, 329
FileStream, 419
filtering
collections, 614
query expressions, 623–624
WHERE clause, 568–569
WHERE clause, 623–624
classes, 410–412
finally blocks, 199–200
FindAll() method, 645
firing events, 548, 553
**Index**

*GroupJoin()* method, 613  
guest computers, 872  
guidelines

- addition (+) operators, 91  
- anonymous methods, 513  
- attributes
  - assemblies, 692  
  - `AttributeUsageAttribute` class, 700  
- constructors, 699  
- custom, 693  
- catch blocks, 204  
- classes
  - access modifiers, 240  
  - naming, 8, 214  
- collections, 649  
- comments, 23  
- constants, fields, 267  
- constructors
  - defaults, 248  
  - naming, 251  
- conversion operators, 393  
- Count() method, 586  
- covariance, 489  
- curly braces ({}), 112  
- custom collections, 643  
- delegates, types, 515  
- empty collections, 660  
- `Equals()` method, 385  
- events, declaring, 554  
- exceptions
  - customizing, 437  
  - handling, 207, 432–435  
  - multiple exception types, 425  
  - reporting, 435  
  - throwing, 106, 201  
  - wrapping, 439  
- extension methods, 266  
- fields, 234–235  
- finalizers, 417  
- floating-point types, 93, 95  
- generics
  - implementing multiple interfaces, 457  
  - methods, 481  
  - type parameters, 462  
- `goto` statements, 145  
- identifiers, 7  
- `if/else` statements, 120  
- increment/decrement operators, 101  
- integers, 35  

- interfaces
  - adding members, 335  
  - attributes, 338  
  - comparing to classes, 337  
  - explicit/implicit implementations, 324–325  
  - implementations, 326  
  - multiple inheritance, 333  
  - naming, 315  
- lambda parameters, 509  
- literal suffixes, 40  
- local variables, 15  
- locking, avoiding, 823  
- long-running tasks, 770  
- for loops, 132, 133  
- managed wrappers/unmanaged methods, 856  
- methods, naming, 157  
- multiple type parameters, 460  
- multithreading, 733, 736  
- aborting threads, 743  
- thread pools, 745  
- `Thread.Sleep()` method, 741  
- unhandled exceptions, 764  
- namespaces, 161, 401  
- nested classes, 272  
- .NET (Microsoft), 6  
- `null`, invoking delegates, 539  
- `OrderBy()` / `ThenBy()` methods, 592  
- P/Invoke, 862  
- parallel loops, 797  
- parameters, 166, 191, 192  
- parentheses (()), 90  
- properties, 234–235  
  - get-only, 239  
  - validating, 237  
- query expressions, 634  
- static initialization, 262  
- `switch` statements, 137  
- synchronization, avoiding, 823  
  - `System.EventHandler<T>` class, 555  
- thread synchronization, 811–841  
  - best practices, 827–829  
  - `Monitor`, 817–819  
  - `ToString()` method, 373  
- types, naming parameters, 453–454  
- value types
  - avoiding mutable types, 355  
  - creating enums, 361  
- defaults, 347  
- defining structs, 369  
- direct enum/string conversions, 3656  
- enum underlying types, 360  
- flag enums, 366  
- immutable, 345  
- memory, 341  
- overloading equality operators, 349  

- XML comments, 407  
- `Handle()` method, 760  
- handlers, events
  - adding, 846–848  
  - removing, 846–848  
- handling
  - aliasing, 697–699  
  - errors
    - platform interoperability/unsafe code, 854–856  
    - sequential notification, 544–547  
  - using statements, 575  
- exceptions, 423  
  - asynchronous high-latency operations, 775–776  
  - avoiding, 206–207  
  - background worker patterns, 931–932  
  - catching, 426–427  
  - defining custom exceptions, 435–438  
  - general catch blocks, 428–432  
  - guidelines, 432–435  
  - multiple exception types, 424–425  
  - rethrowing, 438–442  
  - wrapping, 438–442  
- exceptions, parallel loops, 798–800  
- hard coding values, 38–40  
- hash codes, 651  
- headers, files, 168  
- heaps
  - memory, 349  
  - reference types, 59, 342  
- Heater objects, 534–535  
- `HelloWorld` program, 1, 2–4  
  - output, 29–30  
  - `static` keyword, 255  
- `Help` property, 689  
- hexadecimal notation, 40–41
hiding exceptions, 432
hierarchies
classes, 212, 324
interfaces, generic collections, 637
organizing, 161
high-latency operations, invoking, 771–772, 772–777
hill climbing, 798
hot tasks, 748. See also tasks
Hyper-Threading, 730
I/O-bound latency, 728, 732
IAngle.MoveTo interface, 354
IAsyncAction<T> interface, 848, 849
ICollection<T> interface, 638
IComparable interface, 323, 324
IComparable<T> interface, constraints, 465
IComparer<T> class, 495
identifiers, 6–7
keywords as, 8
namespaces, nesting, 400
IEnumerable interface, 636–638
IDisposable interface
finalization, 415–418
resource cleanup with, 413
tasks, 770
IDispose() method, 915
IDistributedSettingsProvider interface, 335
IEnumerable<T> interface, 571–577
IEnumerable interface, 331
IEnumerable<T> interface, 616n1
extension methods, 562
query expressions, 616
standard query operators, 577–610
if/else statements, guidelines, 120
#if preprocessor directive, 146–147
if statements, 80, 107–110
Boolean expressions, 114–121
IFileCompression interface, 314, 315
IFormatter interface, 357
ILDASM, CIL, 28–29
IListable interface, 331
IList<T> interface, 636–638
ILMerge utility, 889
ILSpy, 30
immutable strings, 17, 51, 52
modifying, 869–870
implementation-defined behavior, 101
implementing
CIL, outer variables, 520–521
CLI, 877–878
conversion operators, 392
defered execution, 623
dynamic objects, 714
Equals() method, 382
events, customizing, 558–559
GetHashCode() method, 375–376
members
explicit, 322–323
implicit, 323–326
multiple inheritance, interfaces, 331–334
new operator, 246
object-oriented programming, 210–213
one-to-many relationships, 601–604
outer joins, 603–604
properties, automating, 232–234
System.Runtime.Serialization.ISerializable class, 709–710
implicit base type casting, 281
implicit conversions, 64, 392
implicit deterministic resource cleanup, 216
implicit local variables, 562–568
implicit members, implementing, 323–326
implicit nondeterministic resource cleanup, 216
implicit overriding, Java, 292
implicitly typed local variables, 55–57
in type parameter, 485–488
including code, preprocessor directives, 146–147
Increment() method, 825
increment (+++) operators, 97–102
indenting
code blocks, 111
whitespace, 12
IndexerNameAttribute, 657
indexers assigning, 657–658
collections, 655–659
indexes, retrieving elements, 640
IndexOf() method, 643
inequality, floating-point types, 93–96
inferences, types, 478–479
infinity
negative, 96
recursion errors, 186
infrastructure, CLI, 875–877. See also CLI
inheritance, 211–212, 277, 310–311
abstract classes, 302–308
aggregation, 289
base classes
base member, 300–301
new modifier, 295–299
overriding, 290–302
sealed modifiers, 299–301
chaining
casting, 282–283
exceptions, 437
classes, exceptions, 201
constraints, 471–472
definitions, 277–278
derivation, 278–290
casting between types, 281–282
customizing conversions, 283–283
private access modifiers, 284–285
protected access modifiers, 285–286
extension methods, 287
interfaces, 326–329
multiple, 329–330, 331–334
value types, 348–349
is operator, verifying underlying type, 309–310
multiple, 287
polymorphism, 306
sealed classes, 290
single, 287–289
System.object, deriving classes, 308–309
virtual modifiers, 290–295
initialization
anonymous types, arrays, 570–571
centralizing, 252–253
initialization (cont’d)
classes, 132
collection initializers, 568–571
lazy, 419–421
NextId, 261
static, 262
static fields, 257
structs, 346–347
Initialize() methods, 236
initializers
common, 249
constructors, 251
objects, 247–248
initializing attributes, constructors, 694–699
inner classes, Java, 272
inner joins, 593
Join() method, 597–600
InnerExceptions property, 437, 439, 760
input, consoles, 17–19
inserting newline (\n) characters, 46
installing .NET (Microsoft), 897–899
instances
applications, formatting single, 829–830
delegates, returning, 539
fields, 217–219. See also static fields
methods, 48, 78–79, 219–220
polymorphism, 321
instantiation, 10
arrays, 70–74
classes, 213–216
delegates, 500–502
genrics
reference types, 492–493
value types, 491–492
ingegers, 34–35
42 as, 195
adding, 351
values, overflowing, 62, 440
integral types, 91
IntelliSense, enabling, 615
interfaces, 313–315
APIs, 27, 438, 439
attributes, comparing, 337–338
automatically shimmed, 848
classes
comparing, 336–337
conversions, 326
collection, 561–562, 636–638
anonymous types, 562–564
implicit local variables, 564–565, 565–566
constraints, 465–467
diagrams, 333–334
extension methods, 330–331
factory, 475
genrics, 454–455, 456–457
hierarchies, generic collections, 637
IAngle.MoveTo, 354
IAsyncAction<T>, 848, 849
ICollection<T>, 638
IComparable, 323, 324
IComparable<T>, 465
IDictionary<T>, 636–638
IDisposable
finalization, 415–418
resource cleanup with, 413
tasks, 770
IEnumerable<T>, 571–577
IEnumerable, 331
IEnumerable<T>
extension methods, 562
query expressions, 616
standard query operators, 577–610
IFileCompression, 314, 315
IFormattable, 357
IEnumerable<T>, 636–638
implementing, 316–319, 320–326
inheritance, 326–329
multiple, 329–330, 331–334
value types, 348–349
IObsolete, 338
IOrderedEnumerable<T>, 592
IQueryable<T>, 609–610
IQueryable<T>, 848
ISerializable, 709
ITrace, 325
multithreading, 492–493
ascending, prior to TPL
and C# 5.0, 907–936
naming, 315
PairInitializer<T>, 487
parameters, 192
polymorphism, 315–320
versioning, 334–336
Windows UIs, 790–792, 932–936
internal access modifier, 397
internals
anonymous methods, 517–518
delegates, 503–506
events, 556–558
genrics, 489–493
lambda expressions, 517–518
multicast delegates, 542–544
interoperability, 25
CIL, 890. See also CIL
languages, 25
platforms, 845–846, 862–872
Intersect() method, 608
into keyword, 629–630
InvalidOperationException, 427, 439
Invoke() method, 685, 932
InvokeRequired property, 932
invoking
asynchronous delegate invocation, 921–924
asynchronous tasks, 747–748
delegates
events, 537–538
exception sequence diagrams, 545
pass-by references, 547–548
returning methods, 547–548
sequences, 542
thread-safe, 539–540
finalizers, 412
high-latency operations, 771–777
members, reflection, 681–686
methods, query expressions, 632–634
reflection, dynamic objects, 714–716
type members, 678. See also
reflection
IObsolete interface, 338
IOrderedEnumerable<T> interface, 592
IProducerConsumer-Collection<T> class, 835
IReadOnlyPair<T> interface, 609–610
IReadableSettingsProvider interface, 327
IReadOnlyPair<T> interface, 484
IReadOnlyPair<T> interface, 484
is operator, verifying underlying types, 309–310
IsAlive property, 740
IsBackground property, 739
IsCancellationRequested property, 766, 767, 801
IsCompleted property, 750, 804
IsCompleted property, 709
IsCompleted property, 804
Index

ISerializable interface, 709
ISettingsProvider interface, 327
IsKeyword() method, 620
items
  collections, searching, 645
counting, 585–586
format, 20
grouping, 600–601
Items property, 456
iterations
  collection interfaces, 573
continue statements, 141
long-running loops, 802
loops, 129, 794–804
iterators
  collections, 660–674
  compiling, 671–672
contextual keywords, 672–673
creating multiple in single classes, 673–674
defining, 661
to examples of, 666–667
state, 664–666
struct versus class, 670
syntax, 661–662
yield break statements, 670–671
yield return statements, 674
yielding values, 662–664
recursion, 669
ITrace interface, 325
IWriteableSettingsProvider interface, 329
jagged arrays, 73, 75
Java, 1
to arrays, declaring, 69
classes, naming, 4
exceptions, specifying, 427
generics, 493
implicit overriding, 292
inner classes, 272
Main method, 10
virtual methods, 291
JavaScript, implicitly typed variables, 565
jitting, 879
Join() method, 739, 748
inner joins, 597–600
joining
collections, 593
data types, 57
jump statements, 139–145
break statements, 139–141
continue statement, 141–143
goto statement, 143–145
just-in-time compilation, 24, 466, 879
JustDecompile, 30
keys, 637
keywords, 2, 4–6
into, 629–630
async, 777–781, 937–942
await, 741, 777–781, 937–942
class, constraints, 468
contextual, iterators, 672–673
default, 68
delegate, 542
event, 550, 551
as identifiers, 8
integers, 34
lock
  applying, 819–821
  selecting objects, 821–822
new, 71, 205
null, 53–54
operator, 392
override, 292, 323
private, 229
properties, defining, 232
static, 264
string, avoiding locking, 822–823
struct, 343, 468
this, 220–227
avoiding locking, 822–823
chaining constructors, 251–253
try, 197
typeof, 363, 822–823
unchecked, 442
void, 53, 54–55
where, 465
lambda expressions, 495, 506–512
asynchronous programming, 782–783
expression trees, 524
internals, 517–518
lazy loading, 420
statements, 507–510
tables, 511–512
languages
  CIL, 876
  CLI. See CLI
  CLR, 881
  CLS, 25, 891–892
  CLU, 660
  COBOL, 890
  FORTRAN, 890
  overview of, 1–2
  Pascal, 7
  source, 890
  XML, 23–24. See also XML
  last in, first out (LIFO), 444, 652
  LastIndexOf() method, 463
  late binding, 893
  latency, 728
  invoking high-latency operations, 771–777
  lazy initialization, 419–421
  left outer joins, 593
  length
    arrays, 75–76
    strings, 51
Jet clause, 625–626
libraries
  assemblies, 394
  ATL, 287
  BCL, 25, 26, 34, 885, 892, 894
classes, 394
code, creating, 4
  DLL, 4
  FCL, 892, 895
  Reactive Extensions library, 729
  Task Parallel Library (TPL), 729, 790. See also TPL
  WinRT, 846–849
limitations, constraints, 472–473
line-based statements (Visual Basic), 11
#line preprocessor directive, 150
lines, specifying numbers, 150
linked lists, collections, 654–655
LinkedListNode<T> class, 655
LinkedList<T> class, 654–655
links, DLL, 4
LINQ
  expression trees, 527
  providers, customizing, 609
  queries. See also queries
  expressions, 561, 613
  running in parallel, 584–585, 804–808
support, 27
Linux, installing platforms, 898
Liskov, Barbara, 660
listings
  anonymous methods, passing, 512
APM patterns
  accessing user interfaces, 933–934
  asynchronous delegate invocation, 922–923
  background worker patterns, 928–929
  ContinueWith() method, 919
EAP, 926–927
invoking user interface objects, 935–936
invoking with callback/state, 911–912
passing state, 913–914
System.Net.WebRequest class, 908–909
using TPL to call, 915–918
arrays
  accessing, 74
  assigning, 70
  command-line options, 80
  declaring, 69
  defining size at runtime, 72
  errors, 72
  initializing two-dimensional arrays of integers, 72
  jagged, 73, 75
  length, 75, 76
  literal values, 71
  methods, 77
  new keyword, 71
  retrieving dimension size, 79
  reversing strings, 80–81
  swapping data, 74
  three-dimensional, 73
  throwing exceptions, 75
two-dimensional arrays, 69, 72, 74–75
attributes
  applying named parameters, 700
  assembles within Assembly-Info.cs, 690–691
  AttributeUsageAttribute, 699–700
  backward compatibility, 712
  constructors, 695
decorating properties with,
  689, 690
  defining custom, 693
  FlagsAttribute class, 701–702
  implementing System.Runtime.Serialization.ISerializable class, 709–710
  restricting constructs, 699
  retrieving custom, 693–694
  retrieving specific attributes, 695–696
  saving documents, 706–708
  specifying return attributes, 691
  System.AttributeUsageAttribute class, 699–700
  System.ConditionalAttribute class, 703–705
  System.ObsoleteAttribute class, 705–706
  System.SerializableAttribute class, 713
  updating CommandLineHandler.TryParse(), 697–699
  break statements, 139–140
  case-sensitivity of multiple strings, 43
  checked blocks, 62–63
classes
  access modifiers, 229
  accessing fields, 218, 219–220
  accessing static fields, 257–258
  assigning static fields at declaration, 257
  automatically implemented properties, 233–234
  avoiding ambiguity, 221–222
  calling constructors, 245–246, 251–252
  calling object initializers, 248, 249
  CIL code from properties, 243–244
data persistence to files, 224
data retrieval from files, 225
declaring, 8
declaring constant fields, 267
declaring fields, 256
declaring static classes, 260
declaring static constructors, 261
declaring static fields, 256
declaring static properties, 262
declaring variables of class types, 214
defining, 213
defining constructors, 245
defining nested classes, 270–271, 273, 274–275
defining partial classes, 272–273
defining properties, 231, 240–242
defining read-only properties, 238
defining static methods, 259–260
explicit construct properties, 244
implicit local variables, 254
initialization methods, 253
instantiation, 215
overloading constructors, 250
passing this keyword in method calls, 223
placing access modifiers on setters, 239–240
readonly modifiers, 268, 269
setting initial values of fields, 217
this keyword, 220–221, 222
validating properties, 236
code blocks
  if statements, 110
  indentation, 111
collection interfaces
  filtering with System.Linq.Enumerable.Where(), 568–569
  foreach with arrays, 572
implicit local variables with anonymous types, 562–563
initializing anonymous type arrays, 570–571
iterating, 573
Index

resource cleanup with using statements, 575
results of foreach, 575–576
separate enumerators during iteration, 574–575
type safety, 566–567
comments, 21
continue statement, 141–143
control flow statements
do/while loops, 129
foreach loops, 134
if/else formatted sequentially, 109
if/else statements, 107
if statements, 135
multiple expressions (for loops), 132
nested if statements, 108
while loops, 127
custom collections
adding items to Dictionary<T> class, 647
applying Pair<T>.GetEnumerator() method, 667
applying yield statements, 666
bitwise complement () operator, 644
compiling iterators, 671–672
defining index operators, 658–659
defining indexers, 655–657
FindAll() method, 645
implementing IComparer<T> interface, 642
implementing IEqualityComparer<T> interface, 650
inserting items to Dictionary<T> class using index operators, 647
iterating over Dictionary<T> class with foreach, 648–649
iterator interface patterns, 661–662
List<T> class, 640
modifying indexer default names, 658
yield break statements, 670–671
yield return statements, 673
yielding C# keywords sequentially, 663–664
delegates
applying method names as arguments, 501–502
applying variance, 516–517
BubbleSort() method, 496–497
BubbleSort() method, ascending/descending, 497–498
BubbleSort() method, parameters, 498–499
capturing loop variables, 521–522, 523
CIL code for outer variables, 520
CIL for lambda expressions, 517–518
declaring Comparison-Handler, 501
declaring Func/Action, 514–515
declaring nested types, 500
declaring types, 500
equity trees, 525
outer variables, 518–519
passing as parameters, 502
viewing expression trees, 528–530
dynamic objects
customizing, 721–723
overriding members, 722–724
runtime binding, 719–720
equality operators, overriding, 376–380
Equals() method, overriding, 383–384
escape sequences, 44
events
applying assignment operators, 548
CIL code, 556–557
connecting publishers/subscribers, 536–537
custom add/remove handlers, 558
custom delegate types, 554–555
declaring generic delegate types, 552
declaring OnTemperature-Change event, 556
defining Heater/Cooler objects, 534–535
defining publishers, 536
delegate operators, 540, 541
event keyword, 550–551
firing, 549
firing notifications, 553
handling exceptions from subscribers, 546–547
invoking delegates, 537–538
OnTemperatureChanged() method, 544
exceptions
catching, 426, 428–431
checked blocks, 440–441
customizing, 436
defining serializable, 438
overflowing integer values, 440
throwing, 424
unchecked blocks, 441–442
explicit casts, 61
generics
arity, 460
BinaryTree<T> class, 462–463, 480
CIL code for Stack<T> class, 490
CIL with Exclamation Point Notation, 491
combining constraints, 473–474
combining covariance and contravariance, 487
ComparisonHandler, 504–505
compile errors, 458
compiler validation of variance, 488
constraint expressions, 473
contravariance, 486
converting generics, 482
covariance, 482, 483–484
covariance using out type parameter modifier, 484
Create() method, 461
declaring class type constraints, 467
Index

listings, generics (cont’d)
   declaring constructors, 457
   declaring generic classes,
      Stack<T>, 452
   declaring interface constraints, 465
   declaring interfaces, 454
   declaring nullable types, 449
   declaring variables of type
      scatter<T>, 491
   declaring versions of value
types, 448
   default constructor constraints, 470, 475
   default operator, 458
   defining methods, 477
   defining specialized stack
classes, 447
   duplicating interface imple-
mentations, 456
   EntityDictionary<T> class,
      476
   factory interfaces, 475
   implementing interfaces, 455
   implementing Undo with
      Stack class, 450–451
   inferring type arguments, 478
   inheritance constraints, 474
   inheritance constraints,
      specified explicitly, 471
   interface support, 463
   multiple constraints, 469
   multiple type parameters, 459, 460
   nested types, 462
   repeating inherited constraints, 472
   specifying constraints, 479–480
   specifying type parameters, 478
   struct/class keywords,
      468
   supporting Undo, 444–445
   System.Collections.Stack
      method signatures, 444
   System.Delegate class, 504
   type parameter support, 464
   GetHashCode() method, over-
riding, 375
   goto statements, 143–144
   HelloWorld, 2
   breaking apart, 9
   output, 29–30
   hexadecimal notation, 41
   implicit conversions, 64
   implicit local variables, 56–57
   inheritance
      accessing base members, 300
      accessing private members,
         284
      accessing protected members,
         285–286
      applying methods, 280
      applying polymorphism, 306–307
      defining abstract classes, 303
      defining abstract members,
         303–304
      defining cast operators, 283
      deriving classes, 279, 280
      implicit base type casting,
         281
      is operator determining
      underlying type, 309
      new modifier, 297–298
      as operator conversions,
         310–311
      overriding properties, 291
      preventing derivation, 290
      run() method, 294
      sealing members, 299
      single using aggregation, 288
      specifying base constructors,
         301–302
      System.Object derivation,
         309
   virtual methods, 292, 293
   integers, overflowing values, 62
   interfaces
      applying base interfaces in
      class declarations, 328
      calling explicit member
      implementations, 322
      declaring explicit members,
         327
      defining, 315
      deriving, 326–327, 335
      explicit interface implemen-
tation, 322–323
      implementing, 316–319,
         320–321
      multiple inheritance, 329
      single inheritance using
      aggregation, 331–332
   lambda expressions
      omitting parameter types,
         508
      parameterless statements,
         509
      passing delegates, 510
      single input parameters, 509
      statements, 507
   literal values, 38, 39
   for loops, 130
   Main() method, 10
   methods
      aliasing, 171–172
      calling, 158
      catching exceptions, 196–197
      converting a string to an
      int, 194
      counting lines, 184–185,
         187–189
      declaring, 163–164
      finally blocks without catch
      blocks, 199–200
      general catch blocks, 203
      grouping statements, 156–157
      optional parameters, 189–191
      passing command-line
      arguments, 173
      passing return values, 162
      passing variable parameter
      lists, 181–182
      passing variables by refer-
ence, 177–178
      passing variables by values,
         175–176
      passing variables out only,
         179–180
      rethrowing exceptions, 206
      return statements, 167
      specifying parameters by
      names, 191–192
      throwing exceptions,
         204–205
      using directive, 170–171
   multiple statements on one
      line, 12
   multithreading
      applying Task.Factory.
      StartNew() method, 769
   asynchronous Web requests,
      773–774, 777–778
   await keyword, 787–788
calling Task.ContinueWith method, 752, 789
cancelling parallel loops, 800–801
cancelling PLINQ queries, 807–808
cancelling tasks, 765–766
customizing asynchronous methods, 784–785, 785–786
handling tasks, unhandled exceptions, 758–759
invoking asynchronous tasks, 747–748
iterating over await operations, 792–793
lambda expressions, 782–783
LINQ Select() method, 804
long-running tasks, 769–770
for loops, 794–795, 796
observing unhandled exceptions, 760–761
parallel execution of foreach loops, 797
PLINQ Select() method, 805
PLINQ with query expressions, 806
polling Task<T> classes, 749
registering for notifications, 756
registering for unhandled exceptions, 762–763
starting methods, 737–738
synchronous high-latency invocation with WPF, 791–792
synchronous Web requests, 772–773
ThreadStart, 743–744
unhandled exception handling, parallel iterations, 799
\n character, 46
no indentation formatting, 12
nullable modifiers, 60
operators, 86
AND (&&), 118
binary, 87–88
bitwise, 124
char data types, 92
cancel difference, 92
common increment calculations, 97
conditional, 119
constants, 103
decrement (–), 98
dividing a float by zero, 95
equality, 116
examples of assignment, 97
increment (+), 97
inequality with floating-point types, 94–95
logical assignment, 126
negative values, 87
non-numeric types, 91
NOT, 118
null coalescing, 120
overflowing bounds of float, 96
post-increment, 99
pre-increment, 99
prefix/postfix, 100
relational, 116
Unicode values in descending order, 98
Parse() method, 65
placeholders, 20
platform interoperability/unsafe code
accessing referent type members, 871–872
allocating data on call stacks, 868
applying ref/out rather than pointers, 852
declaring external methods, 849–850
declaring types, 853
designating unsafe code, 863, 872–873
encapsulating APIs, 859–860
fixed statements, 867, 868
invalid referent types, 866
managed resources, 857–858
modifying immutable strings, 869–870
SafeHandle, 856–857
Win32 error handling, 854–855
WinRT patterns, 847–848
wrapping APIs, 861
#pragma preprocessor directive, 149
preprocessor directives, 147
#define, 147
excluding/including code, 147
#region/#endregion, 151
#warning, 148
query expressions, 614–615, 632–633
anonymous types, 618
continuation, 630
deferred execution, 619–622
distinct members, 631–632
filtering, 623–624
grouping, 626–627
multiple selection, 630
ordering results, 616
projection using, 617
selecting anonymous types, 628–629
sorting, 624
sorting by file size, 625
standard query operator syntax, 633
reflection
declaring Stack<T> class, 686
dynamic programming using, 715
dynamically invoking members, 681–684
generics, 687, 688
using Type.GetProperties() method, 679–680
using Type.GetType() method, 680
round-trip formatting, 42
single statements, splitting, 12
standard query operators
calling SelectMany() method, 604–605
classes, 578–580
counting items, 585
creating child collections, 602
executing LINQ queries in parallel, 584
grouping items, 600–601
inner joins, 597–598, 599
ordering, 590–591
outer joins, 603–604
listings, standard query operators (cont’d)
projection to anonymous
 types, 583
projection with System.
Linq.Enumerable.Select() method, 582
sample employee/department data, 594–596
strings
applying, 52–53
assigning null to, 54
binary displays, 125
immutable, 52
implicitly typed local vari-
ables, 55–56
length, 51
switch statements, 137–138
System.Console.ReadLine() method, 17–18
System.Convert class, 65
System.Threading.Timer class, 941–942
System.Timers.Timer class, 939–940
thread synchronization, 811–841
best practices, 827–829
creating single instance
applications, 829–830
firing event notifications,
826
lock keyword, 820
ManualResetEventSlim, 832–833
Monitor class, 817–818
System.Threading.Inter-
locked class, 824–825
Task.Delay() method, 842
thread-safe event notifica-
tion, 826
ThreadLocal<T> class, 838
ThreadStaticAttribute
 class, 839–840
unsynchronized local vari-
ables, 815–816
unsynchronized state, 813
Tic-Tac-Toe source code,
901–905
timers, 939–940
ToString() method, 65, 372–373
unchecked blocks, 63
Unicode characters (smiley
faces), 45–46
value types
accessing properties, 346
avoiding copying/unbox-
ing, 357
boxing idiosyncrasies, 353
boxing/unboxing instruc-
tions, 351
casting between arrays/
enums, 362
comparing integer switches/
enum switches, 358
converting strings to enums,
363
declaring enums, 359
declaring structs, 344–345
default operator, 348
defining enum values, 366
defining enums, 359
enums as flags, 364
FlagsAttribute, 367
initializing structs, 346
OR/AND operators with
flag enums, 365
referencing Equals() method, 381
unboxing to underlying
types, 353
variables
assigning, 16
declaring, 13
modifying values, 15
one statement, declaring
with, 15
scopes, 114
verbatim string literals, 47
well-formed types
adding operators, 387–388
applying alias directives, 402
calling binary operators, 388
calling comparison operators, 386
conversion operators, 392
defining finalizers, 411
defining namespaces, 399
invoking using statements,
415
lazy loading properties, 419,
420
making types available, 396
nesting namespaces, 400
overloading unary opera-
tors, 390, 391
resource cleanup, 413
weak references, 409
XML comments, 403–406
whitespace, removing, 12
lists
collections, 640
formal parameters, 165
linked, collections, 654–655
type parameters, 165
List<T> class, 482, 639–641
searching, 643–644
literals
strings, 46–48
values, 37–38
arrays, 71
readonly fields, 269
loading
files, 224
lazy. See lazy initialization
local negation (!) operator, 118–119
local storage, threads, 837–841
local variables, 14
anonymous types, 562–563
declaring, 165–166
implicit, 564–565, 566–568
implicitly typed, 55–57
multiple threads, 815–816
scope, 114
locations
keywords, 4
objects, 408
reference types, 341–345
lock keyword
applying, 819–821
objects, selecting, 821–822
Lock statements, 736
value types, 353–255
locking, 828–829
avoiding, 822–823
consoles, synchronization,
920–921
multithreading, 736
LockTaken parameter, 819
logical Boolean operators,
116–121, 122–126
Logon() method, 228
long-running
loops, 802
tasks, 769–770
loops
for, 120–133
decrement (--) operators, 98

do/while, control flow statements, 127–129
executing, iterations in parallel, 794–804
foreach, 133–136
arrays, 571–572
IEnumerable<T> interface, 572–577
iterations, 129
parallel
breaking, 803–804
executing iterations in, 794–804
options, 802–803
variables, capturing, 521–522
while, control flow statements, 127–129
yield return statements, placing in, 667–669
LowestBreakIteration property, 804

machine code, 24, 876
compilation to CIL, 879–880
macros, preprocessors, 145
Main() method, 8, 9, 156, 821
declaring, 10
methods, refactoring, 165
multiple, disambiguating, 174
returns and parameters, 172–175
_makeref keyword, 8
MakeValue() method, 470
managed code, 24, 881
runtime performance, 885–886
managed data, 881
managed execution, 881
managing
execution, 24–26
hierarchies, 161
memory, 412. See also garbage collection
object-oriented programming, 210–213
threading, 739–740
manifests, CLI, 887–890
many-to-many relationships, 594
mark-and-compact algorithms, 407
mark-and-sweep-based algorithms, 882
masks, 125

matching parameter names, 176
math constants, 112
Max() method, 609
Max<T> method, 479
MemberInfo class, 685
members
abstract, 302, 303–304, 305
classes, 216
distinct, 631–632
dynamic objects, overriding, 723–724
explicit, implementing, 322–323
implicit, implementing, 323–326
interfaces, adding, 335
object, overriding, 371–385
overloading, 292
referent types, accessing, 871–872
reflection, invoking, 681–686
sealing, 299
static, classes, 255–265
of System.Object, 308
types
access modifiers, 397–398
invoking, 678. See also reflection
variables, 217
memory
garbage collection, 408
heaps, 349
managing, 412. See also garbage collection
models, 735
retrieving, 246
virtual, allocating, 851
messages
exceptions, 426. See also exceptions
warnings, turning off, 149–150
metadata, 23, 25, 877
CLI, 892–893
reflection, 678
System.Type class, accessing
using, 679–680
MethodImplAttribute, avoiding with synchronization, 823
methods, 155, xxiii
Abort(), 740
Add(), 249, 352, 473, 568–569, 641
adding, 476, 544
AllocExecutionBlock(), 855
anonymous, 495, 512–514, 517–518
APM, calling, 915–921
arguments, 161–162
arrays, 77–79
AsParallel(), 584
Assert(), 95
asynchronous, customizing, 783–786
Average(), 609
BeginGetResponse(), 908
BeginX(), 908
BinarySearch(), 643, 644
binding, 714
boxing, avoiding, 356–357
Break(), 803
BubbleSort(), 496–497
calling, 11, 156–163, 295
Cancel(), 766
Capacity(), 639
clear(), 942
Close(), 78
Collect(), 408
Combine(), 175, 178, 182, 542
Compare(), 495
CompareTo(), 323, 463, 641
Compress(), 325
Concat(), 608
Console.WriteLine(), 340, 352
Contains(), 643
ContinueWith(), 753, 775, 919–920
CopyTo(), 638
Count(), 585–586, 609, 622
CountLines(), 157
Create(), 461
declaring, 163–168
Decrement(), 814, 821, 825
default(), 71
DefaultIfEmpty(), 603
definitions, 9
delegates, returning, 547–548
Dequeue(), 654
Directory.CountLines(), 185, 187, 191
Directory.GetFiles(), 617
DirectoryInfoExtension.Copy(), 260
DirectoryInfo.GetFiles(), 597
Dispose(), 6n6, 413, 416, 575, 770
methods (cont’d)
Distinct(), 608, 613, 631–632
DoStuffAsync(), 787
double.TryParse(), 207
Dump(), 325
EAP, 925
Empty<T>(), 659
capsulation, 215
EndGetResponse(), 908
EndX(), 908
Enqueue(), 654
Enter(), 353
Enum.Parse(), 363, 680
Equals(), 349, 376–385, 651
events, 194–208
Event(), 474
exceptions, 194–208
Exit(), 353
extension, 265–266
inheritance, 287
interfaces, 330–331
extension, IEnumerable<T>
interface, 562
factory, 461
FindAll(), 645
ForEach(), 801
Format(), 48
GetCustomAttributes(), 694
GetDynamicMemberNames(), 724
GetEnumerator(), 576, 577, 610, 661, 662, 664, 787n8
GetFiles(), 157
GetFirstName(), 232
GetFullName(), 166
GetGenericArguments(), 687
GetHashCode(), 349, 373–376, 651
GetInvocationList(), 547, 548
GetLength(), 79
GetName(), 219
GetResponse(), 772
GetResponseAsync(), 779
GetReverseEnumerator(), 673
GetSetting(), 327
GetSummary(), 305
GetSwitches(), 696
GetType(), 679–680
GetUserInput(), 166
GetValue(), 685
global, 164
GreaterThan, 507
GroupBy(), grouping results, 600–601
GroupJoin(), 601–604, 613
groups, 502
Handle(), 760
IBaseDispose(), 915
Increment(), 825
IndexOf(), 643
initialize(), 236
instances, 48, 219–220
Intersect(), 608
Invoke(), 685, 932
invoking, 632–634
IsKeyword(), 620
Join(), 597–600, 739, 748
LastIndexOf(), 643
Logon(), 228
Main(), 8, 9, 156, 821
declaring, 10
refactoring, 165
returns and parameters, 172–175
MakeValue(), 470
Max(), 609
Max<T>(), 479
Min(), 609
Monitor.Enter(), 818
Monitor.Exit(), 818
Move(), 345
MoveNext(), 573, 576, 664
NameChanging(), 276
namespaces, 158–160
naming, 161
OfType<T>(), 608
OnFirstNameChanging(), 276
OnLastNameChanging(), 276
OnTemperatureChanged(), 535, 544
OrderBy(), 590–596
overloading, 186–189
Parallel.For(), 795, 803
parameters, 161–162
advanced, 175–184
formal declaration, 165–166
optional, 189–193
Parse(), 65, 195
partial, 273–276
person.NonExistentMethod
callStillCompiles(), 717
PiCalculator.Calculate(), 749
Ping.Send(), 791
Pop(), 444, 652
Print(), 306
ProcessKill(), 785
Program.MethodB(), 704
Pulse(), 819
Push(), 444, 652
ReadToEnd(), 772
ReadToEndAsync(), 775
recursion, 184–186
refactoring, 165
ReferenceEquals(), 380, 387
Remove(), 542, 641
RemoveAt(), 641
Reset(), 573, 834
resolution, 193
Reverse(), 608
Run(), 293, 294, 748
Save(), 310
scope, 161
Select(), 582–584, 804
SelectMany(), 603, 604–606, 613
SendTaskAsync(), 792
SequenceEquals(), 608
SetName(), 221, 222
Sleep(), 740
standard query operators, 577–610
Start(), 293, 737
StartX(), 908
statements
calling, 163
grouping, 156–157
static, 259–261
Stop(), 293, 803
strings, 48–50
subscriber, defining, 534–535
Sum(), 609
SuppressFinalize(), 416
Swap(), 178
System.Console.Clear(), 147
System.Console.ReadLine(), 17, 18
System.Console.WriteLine(), 19–20, 30, 157
System.Diagnostics.TraceListener()
System.Enumeration.IsDefined(), 366
Index

   Where(), filtering with, 568-569
Task.ContinueWith(), 793
Task.ContinueWith, calling, 752, 789
Task.Delay(), 741
Task.Factory(), 768
Task.Factory.StartNew(), 769
Task.Run(), 768, 785
TextWriter.Parse(), 424
ThenBy(), 590-596
Thread.Sleep(), 740-741
ThrowIfCancellationRequested(), 768
ToArray(), 588
ToCharArray(), 80
ToLookup(), 588
ToString(), 65, 357
enum conversions, 362
   overriding, 372-373
TrimToSize(), 639
TryGetMember(), 723
TryGetPhoneNumber(), 179, 180
TryParse(), 66-67, 207-208, 684
TrySetMember(), 723
type of, 680
types
   inference, 478
   naming, 160-161
Undo(), 446
Union(), 608
using directive, 168-172
values, returns, 162
VerifyCredentials(), 333
virtual, defaults, 291
Wait(), 750
WaitAll(), 831
WaitAny(), 831
WaitForExit(), 784
WebRequest.GetResponseAsync(), 774
Where(), 509, 580-591
WriteLine(), 157
WriteWebRequestSizeAsync(), 779
Microsoft
   FCL, 892, 895
ILMerge utility, 889
.NET, 894, 897-899, xxvii
   compilers, 878
delegates, 503
garbage collection, 882-883
   guidelines, 6
   platform portability, 885, 886
   regions, 152
   versioning, 26-28
Silverlight, 878
XNA, 878
Min() method, 609
mod operator. See remainder (%) operators
models
   asynchronous programming, 908-921
COM, STAThreadAttribute class, 842-843
memory, 735
structured programming model definitions, xxiii
threading, 730
modifiers
   access, 227-229. See also access modifiers
classes, 397
runtime, 883
type members, 397-398
new, 295-299
nullable, 60
readonly, 268, 269
sealed, 299-301
virtual, 290-295
volatile, declaring fields as, 823-824
modifying
   access, 237-239
assemblies, targets, 394-395
collections, 577
immutable strings, 869-870
strings, 53
modules
   assemblies, 395
CLI, 887-890
Monitor class, synchronization, 817-819
Monitor, Enter() method, 818
Monitor, Exit() method, 818
monitoring asynchronous operation state for completion, 745
Mono Project, 3n4, 878, 898, xxvii
Move() method, 345
MoveNext() method, 573, 576, 664
moving objects, 408
multicast delegates, 533
   internals, 542-544
observer patterns, 534-548
multidimensional arrays, errors, 72
multiple constraints, 469
multiple definitions, adding, 148
multiple exception types, 424-425
multiple inheritance, 287
aggregation, 289
interfaces, 329-330, 331-334
multiple items, searching collections, 645
multiple iterators, creating single classes, 673-674
multiple Main() methods, disambiguating, 174
multiple selection, query expressions, 631
multiple threads
   event notification, 826-827
   local variables, 815-816
multiple type parameters, 459-460
multiplication (*) operators, 87
multithreading, 727-729
asynchronous tasks, 745-764
interfaces, prior to TPL and C# 5.0, 907-936
LINQ queries, running in parallel, 804-808
loops, executing iterations in parallel, 794-804
overview of, 730-736
   performance, 732-733
System.Threading class, 737-745
Task-based Asynchronous Pattern (TAP), 770-794
tasks
   AggregateException, 757-764
canceling, 764-770
continuation, 751-757
troubleshooting, 734-736
Name property, 240
NameChanging() method, 276
named arguments, 191
namespaces, 168
aliasing, 171, 401–402
defining, 398–402
methods, 158–160
System.Collections.Generic, 638
naming. See also aliasing; naming
_ FirstName, conventions, 234n3
attributes, 692
classes, 4, 8
IndexerNameAttribute, 657
indexers, 657–658
integer types, 34
interfaces, 315
methods, 157, 161
parameters, 166, 176, 453–454
PascalCasing, 235
types, 160–161
native code, 24
NDoc, 406n4
negative
infinity, 96
values, 87
nesting
classes, 269–272
if statements, 108
types, generics, 461–462
.NET (Microsoft), 894, 897-899, xxvii
 compilers, 878
delegates, 503
garbage collection, 407–408, 882–883
guidelines, 6
platform portability, 885, 886
regions, 152
versioning, 26–28
new keyword, 71
throwing exceptions, 205
new modifier, 295–299
new operator
implementing, 246
value types, 347–348
newline (\n) characters, 50
NextId initialization, 261
no-op, 703
non-numeric operands, 90–91
nonprimitive value types, 71
nonstatic fields, 257. See also static fields
normalization, 597
not equals (=) operator, 386
NOT (local negation) operators, 118–119
notation
Alonzo Church, 513
exponential, 40
hexadecimal, 40–41
notifications
events
firing, 553
multiple threads, 826–827
registering for, 756
sequential, error handling, 544–547
NotImplementedException, 321
nowarn:<warn list> option, 149–150
null, 53–54
events, checking for, 538–539
returning, 659–660
null coalescing (??) operators, 120–121
nullable
modifiers, 60
value types, 447–448
NullReferenceException, 424, 538, 551, 659, 826
numbers
Boolean type conversions, 64
Fibonacci, 128, 351
hexadecimal, formatting as, 41
lines, specifying, 150
numeric types, 34–43
object, overriding members, 371–385
object-oriented programming, 210–213
objects
associating, 259
COM, STAThreadAttribute class, 842–843
de-allocating, 882
deterministic destruction, 882
lock keyword, selecting, 821–822
observer patterns, 846
multicast delegates, 534–548
OfType<T>() method, 608
one-to-many relationships, 594
implementing, 601–604
OnFirstNameChanging() method, 276
OnLastNameChanging() method, 276
OnTemperatureChanged() method, 533, 544
operator keyword, 392
operators, 85, 86–103, 117–118, 122–126, 310–311, 365
+, 97, 540
–, 539, 540
adding, 387–388
addition (+), 87, 387, 541
guidelines, 91
strings, 90–91
arithmetic binary, 87–96
assignment
applying, 548
events, 541
await, 741
binary (arithmetic), 387–389
bitwise, 121–127, 140
bitwise complement (~), 127, 644
cast (()), 61, 283, 391–392
characters, applying, 91–92
comparison, 386–387
compound assignment, 96–103
conditional (?), 119–120
conditional logical, 389
constraints, 473
conversions, 391, 393
decrement (–), 97–102
default, 348, 458
delegates, 540–542
delete, 216
division (/), 87
dot (.), 87
equality, 115–116, 376–380
equals (==), 386
ers, 115
exclusive OR (^), 118
floating-point types, 92–96
increment (+), 97–102
index, defining, 658–659
is, verifying underlying types, 309–310
local negation (!), 118–119
logical Boolean, 116–121, 122–126
new
implementing, 246
value types, 347–348
not equals (!=), 386
null coalescing (??), 120–121
OR (||), 116, 117, 122
constraints, 473–474
with flag enums, 365
order of precedence, 153
postfix, 100
prefix, 100
queries, 561–562, 577–610
relational, 115–116
remainder (%), 87
shift, 122–123
simple assignment (=), 15
standard query. See also standard query operators
subtraction (-), 541
unary, 390
minus (-), 86–87, 387
plus (+), 86–87
well-formed types, overloading, 385–393
options
command-line, arrays, 80
methods, parameters, 189–193
newwarn:<warn list>, 149–150
parallel loops, 802–803
TaskCreationOptions.LongRunning, 798
OR (||) operators, 116, 117, 122
constraints, 473–474
with flag enums, 365
order of precedence, operators, 153
orderby clause, 624–625
OrderBy() method, 590–596
distribution, APM, 911
IListable, 331
lockTaken, 819
equals keyword, 292, 323
overriding
abstract members, 305
base classes, 290–302
equality operators, 376–380
Equals() method, 376–385
GetHashCode() method, 373–376
implicit, 292
members, dynamic objects, 723–724
object members, 371–385
properties, 291
ToString() method, 372–373
virtual modifiers, 290–295
overrun, buffers, 76, 883
P/Invoke (Platform Invoke), 849–862
API calls with wrappers, 861
e external functions, calling, 858–861
guidelines, 862
PairInitializer<T> interface, 487
Pair<T> class, 482
palindrome, 79
Parallel LINQ. See PLINQ
parallel programming, 732
Parallel.For() method, 795, 803
parallelism
disabling, 802
LINQ queries, running in, 804–808
loops
breaking, 803–804
executing iterations in, 794–804
options, 802–803
TPL. See TPL
ParallelOptions parameter, 802
ParallelOptions type, 801
overflowing
floating-points numbers, 95
integer values, 440
overloading
constructors, 250–251
members, 292
methods, 186–189
operators, well-formed types, 385–393
types, applying arity, 460
options
command-line, arrays, 80
methods, parameters, 189–193
newwarn:<warn list>, 149–150
parallel loops, 802–803
TaskCreationOptions.LongRunning, 798
OR (||) operators, 116, 117, 122
constraints, 473–474
with flag enums, 365
order of precedence, operators, 153
orderby clause, 624–625
OrderBy() method, 590–596
distribution, APM, 911
IListable, 331
lockTaken, 819
equals keyword, 292, 323
overriding
abstract members, 305
base classes, 290–302
equality operators, 376–380
Equals() method, 376–385
GetHashCode() method, 373–376
implicit, 292
members, dynamic objects, 723–724
object members, 371–385
properties, 291
ToString() method, 372–373
virtual modifiers, 290–295
overrun, buffers, 76, 883
P/Invoke (Platform Invoke), 849–862
API calls with wrappers, 861
e external functions, calling, 858–861
guidelines, 862
PairInitializer<T> interface, 487
Pair<T> class, 482
palindrome, 79
Parallel LINQ. See PLINQ
parallel programming, 732
Parallel.For() method, 795, 803
parallelism
disabling, 802
LINQ queries, running in, 804–808
loops
breaking, 803–804
executing iterations in, 794–804
options, 802–803
TPL. See TPL
ParallelOptions parameter, 802
ParallelOptions type, 801
ParallelQuery<T> class, 806
parameters, 155
advanced, methods, 175–184
AllowMultiple, 700
arrays, 181–184
data types, 850–852
distribution, APM, 911
IListable, 331
lockTaken, 819
Main() method, 172–175
out parameter values, 242–244
Out property, 689
out type parameter, 483–485
outer joins, 593, 603–604
outer variables, 518–519
CIL implementations, 520–521
OutOfMemoryException, 433
output
consoles, 19–21
HelloWorld, 29–30
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
parameters, 178–181
patterns (cont’d)
multicast delegates, 534–548
observer, 546
publish-subscribe, 533
TAP, 729, 770–794, 920
token cancellation, 801
PDAs (Personal Digital Assistants), 278
performance
multithreading, 732–733
runtime, 885–886
synchronization, 828
Task Parallel Library (TPL), 798
permissions, CAS, 686
persistence, 224
Person, 278
person.NonExistentMethodCallStillCompiles ()
method, 717
pi, calculating, 794–795
PiCalculator.Calculate() method, 749
Ping.Send() method, 791
placeholders
formatting, 20
values, 121
placing yield return statements
in loops, 667–669
platforms
addresses/pointers, 862–872
CLI, See CLI
installing, 897–899
interoperability, 845–846
.NET (Microsoft), 897–899
portability, 25, 884–885
WPF, 934–936
PLINQ (Parallel LINQ), 729, 804–808
pointers, 862–872
assigning, 866–869
declaring, 864–866
dereferencing, 869–871
functions, 862
polling
 cancellation tasks, 766
 Task<T> classes, 749
polymorphism, 213, 306
behavior, data types, 310
interfaces, 315–320
operational, 187
pools
temporary storage, 341
threading, 731, 743–745, 746
Peep() method, 444, 652
portability of platforms, 25,
  884–885
positions, bitwise operators for,
  140
post-increment operators, 99
#pragma preprocessor directive,
  149
pre-increment operators, 99
precedence, operators, 88, 89, 153
predefined
attributes, 703
types, 33
predicates, 510
query expressions, 623
Where() method, 580
prefixes
@ as, 8
hexadecimal notation, 41
operators, 150
preprocessor directives, 145–152
code
  editors, 151–152
  excluding/including,
    146–147
errors/warnings, 148–150
line numbers, specifying, 150
#pragma preprocessor directive,
  149
symbols, 147–148
primitives, 33
principles, dynamic objects,
  716–718
Print() method, 306
private access modifiers, 229,
  284–285
private keyword, 229
processes, 730
ProcessKilled() method, 785
processors
cycles, 741
processor-bound latency, 728
products, Cartesian, 598, 631
Program class, 264
Program.MethodB() method, 704
programming
asynchronous, 732, 908–921
comments, 21–23
constructs, associating XML
  comments, 403–405
dynamic, 677, 714–724
object-oriented, 210–213
parallel, 732
sequential programming struc-
  ture definitions, xxiii
syntax, overview of, 4–17
programs
HelloWorld, 1, 2–4
multithreading, 729. See also
  multithreading
Tic-Tac-Toe, 901–905
Project Wizard (Visual Studio),
  690
projecting
to anonymous types, 583
collections, 614
query expressions, 616–619
with Select() method, 582–584
propagating exceptions from
  constructors, 418
properties, 50–51
access modifiers, getters/setters,
  239–240
automating, implementing,
  232–234
classes, 229–244
Count, 638
declaring, 230–232
FileStream, 419
guidelines, 234–235
Help, 689
InnerExceptions, 437, 439,
  760
InvokeRequired, 932
IsAlive, 740
IsBackground, 739
IsCancellationTokenRequested,
  766, 767, 801
IsCompleted, 750, 804
Items, 456
lazy loading, 419
LowestBreakIteration, 804
Out, 689
overriding, 291
parameter values, 242–244
read-only, 51, 237–239
static, 262
System.Reflection.MethodInfo,
  503
Type.ContainsGenericParameters,
  687
validation, 236–237
virtual fields, 240–242
protected access modifiers,
  285–286
Index

returning (cont’d)
instances, delegates, 539
Main() method, 172–175
methods, delegates, 547–548
null, 659–660
types, declaring methods, 166–168
values, 157, 162
reuse
assemblies, 393
code, 394
Reverse() method, 608
right outer joins, 593
root references, 407
Rotor, 878, xxvii
round-trip formatting, 42–43
rounding
errors, 37
inequality with floating-point types, 93–96
rules, keywords, 4. See also guidelines
Run() method, 293, 294, 748
running
applications, 3–4
LINQ queries in parallel, 584–585, 804–808
runtime, 25. See also WinRT
arrays, defining size at, 72
buffer overrun, 76
CIL, 881–886
CLR, 894. See CLR
eXceptions, 200. See also exceptions
garbage collection, 881–882
Main() method, 173
members, overloading, 292
performance, 885–886
WinRT, 895
XML elements, binding, 719–720
SafeHandle class, 856–857
safety
thread-safe. See thread-safe types, 25, 883
unsafe code. See unsafe code
Save() method, 310
saving documents, 706–708
schedulers, tasks, 746, 788–790
scopes, 112–114, 161
sealed classes, inheritance, 290
sealed modifiers, 299–301
sealing members, 299
searching
attributes, 693–694
List<T> class, 643–644
multiple items, collections, 645
security
CAS permissions, 686
code access, 25, 884
SELECT clause, 615
Select() method, 804
projecting with, 582–584
selecting
multiple query expressions, 631
objects, lock keyword, 821–822
SelectMany() method, 603, 604–606, 613, 630
Semaphore class, 835
semaphores, System.Threading.
AutoResetEvent class, 834
SemaphoreSlim class, 835
semicolons (;), 2
statements without, 11
SendTaskAsync() method, 792
SequenceEquals() method, 608
sequences
delegates, invoking, 542
escape, 44, 47
exceptions, diagrams, 545
flattening, 630–632
if/else statements, formatting, 109
notification, error handling, 544–547
programming structures, definitions, xxiii
SelectMany(), 630
yield return statement, 665
serializable exceptions, 438
serialization, 678
attributes, 706–714
customizing, 708–710
System.SerializableAttribute class, 713–714
versioning, 710–713
series, Fibonacci, 128
ServiceStatus, 865
SetName() method, 221, 222
sets, keys, 637
setters, 51
access modifiers, 239–240
accessibility, modifying, 237–239
declaring, 230
shadowing type parameters, 462
sharing
assemblies, 395
state, collection interfaces, 574
shift operators, 122–123
short data types, 35
signatures, APM, 910–911
Silverlight (Microsoft), 878
simple assignment (=) operator, 15
simultaneous multithreading, 730
single classes, creating multiple iterators, 673–674
single inheritance, 287–289
single instance applications, formatting, 829–830
single-line comments, 22
single-threaded programs, 730.
See also programs
sites, calling, 175
Sleep() method, 740
slicing time, 732
Smalltalk, 890
software
multithreading, 729. See also multithreading
thread
872
SortedDictionary<T> class, 652, 653
SortedList class, 653
sorting
collections, 641, 652
OrderBy() method / ThenBy() method, 590–596
query expressions, 624–625
source languages, 890. See also languages
spaces
declaration, 112–114
dirty, 882
specialization, types, 213
specifiers
exceptions, 427
round-trip formatting, 42
specifying
constraints, 479–480
line numbers, 150
SqlException, 437
square brackets ([ ]), 68, 69, 367
attributes, 689
indexers, 655
Stack class, 450–451
StackAlloc data, 868
StackOverflowException, 433
stacks

calling, 175, 868
collections, 652–654
temporary storage pools, 341
unwinding, 175
Stack<T> class, 652–654
CIL, 490
standard query operators, 577–610
Count() method, counting items with, 585–586
deferred execution, 586–590
GroupBy() method, grouping results with, 600–601
GroupJoin() method, implementing one-to-many relationships, 601–604
IQueryable<T> interface, 609–610
Join() method, inner joins, 597–600
LINQ queries, running in parallel, 584–585
OrderBy() method / ThenBy() method, 590–596
Select() method, projecting with, 582–584
SelectMany() method, 604–606
Where() method, filtering with, 580–591
star (*), 756
Start() method, 293, 737
StartX() method, 908
state
APM, passing, 913–914
collection interfaces, 574
iterators, 664–666
statements, xxiii
alternative, 107
block, 110
break, 139–141
consequence, 107
Console.WriteLine(), 871
continue, 141–143
control flow, 85, 103–110, 127–139, 198. See also control flow statements
definitions, 11–12
fixed, 867, 868
if, 143–145
if, 80, 107–110, 114–121
if/else guidelines, 120
jump, 139–145. See also jump statements
lambda expressions, 507–510
line-based, Visual Basic, 11
lock, 353–355, 736
methods
calling, 163
grouping, 156–157
nested if, 108
operator-only, 87
Redim, 78
return, methods, 167
string, join, 79
switch, 135–139, 426
throw, reporting errors, 204–207
using, 606, 185n3, 412–415, 575
yield break, 670–671
yield return, 665, 667–669, 674
STAThreadAttribute class, 842–843
static
classes, 263–265
compilation versus dynamic programming, 720–721
constructors, 261–262
fields, 256–258
initialization, 262
members, classes, 255–265
methods, 48, 259–261
properties, 262
static keyword, 264
Stop() method, 293, 803
storing
files, 224
local threads, 837–841
reference types, 341
static fields, 257
temporary storage pools, 341
string keyword, avoiding locking, 822–823
string, join statement, 796
strings, 46–53
42 as a, 195
addition (+) operator, 90–91
applying, 52–53
arrays, 79–81, 80–81
concatenation at compile time, 48
conversions, 65
encryption, 804
default conversions, 362–364
format, 20
immutable, 17, 51, 52, 869–870
length, 51
methods, 48–50
round-trip formatting, 42
System.Text.StringBuilder, 53
strong references, 408
struct keyword, 434, 468–469
StructLayoutAttribute, 583
structs, 340–349
defining, 369
generics, 454–455
initialization, 346–347
structural equality, delegates, 516–517
structured programming model definitions, xxiii
structures
object-oriented programming, 210–213
sequential programming definitions, xxiii
styles
ambiguity, avoiding, 221
CPS, 911–913
subscriber methods, defining, 534–535
subscribers
encapsulation, 548–549
publishers, connecting, 536–537
subtraction (-) operators, 541
subtypes, 212
suffixes, 39
exceptions, 437
literals, 40
Sum() method, 609
super types, 212
support
covariance, 488–489
delegates, syntax, 502
finalizers, 347
generics, 491
LINQ, 27
OR ((|) operators, 473–474
SuppressFinalize() method, 416
Swap() method, 178
switch statements, 135–139
exceptions, 426
switches
context, 732
unsafe, 864
symbols, preprocessor directives, 147–148
new modifiers, 296
operators, order of precedence, 153
preprocessor directives, 146
sample pseudocode execution, 814
standard query operators, 608
strings
methods, 50
static methods, 49
System.Threading.ILocked class, 825
TaskContinuationOptions enums, 754–755
types of comments, 22
XOR operator values, 118
TAP (Task-based Asynchronous Pattern), 729
APM methods, 920
multithreading, 770–794
targets, modifying assemblies, 394–395
Task-based Asynchronous Pattern. See TAP
task-based asynchrony, 848–849
Task Parallel Library (TPL). See TPL
TaskCanceledException, 767, 768
TaskCompletionSource<T> class, 784
TaskContinuationOptions enums, 754–755
Task.CreatorWith() method, 752, 789, 793
TaskCreationOptions.LongRunning option, 798
TaskDelay() method, 741
Task.Delay() method, 843
Task.Factory() method, 768
Task.Factory.StartNew() method, applying, 769
Task.Run() method, 768, 785
tasks, 731
AggregateException, 757–764
antecedent, 753
asynchronous, 745–764
canceling, 764–770
continuation, 751–757, 789
control flow, 780
IDisposable interface, 770
long-running, 769–770
schedulers, 746, 788–790
TaskScheduler class, 788
Task<T> class, 731, 779
await keyword, 787–788
polling, 749
templates, 449
ATL, 287
C++ language, 466
temporary storage pools, 341
TemporaryFileStream, 412, 419
text
assigning, 42
XML comments, 402–407
TextNumberParser.Parse() method, 424
ThenBy() method, 590–596
Thermostat class, 535
this keyword, 220–227
constructors, chaining, 251–253
locking, avoiding, 822–823
delegates, invoking, 539–540
event notification, 826
incrementing/decrementing, 101–102
ThreadAbortException, 741, 742
threading, 727, 730. See also multithreading
ground threads, 739
managing, 739–740
multiple threads, local variables, 815–816
pools, 731, 743–745, 746
synchronization, 811–812
applying lock keywords, 819–821
avoiding locking, 822–823
avoiding with Method-ImplAttribute, 823
declaring fields as volatile, 823–824
design best practices, 827–829
event notification, 826–827
local storage, 837–841
Monitor class, 817–819
overview of, 813–841
resetting events, 831–837
selecting Lock objects, 821–822
System.Threading.ILocked class, 824–826
timers, 841–843
types, 829–837
troubleshooting, 734–736
unhandled exceptions, 761–764
ThreadLocal<T> class, 838
Thread.Sleep() method, 740–741
ThreadStaticAttribute class, 839
three-dimensional arrays, 73
three-forward-slash delimiters (/\), 405
throw statements, reporting errors, 204–207
ThrowIfCancellationRequested() method, 768
throwing exceptions, 195, 204–205, 321
arrays, 75
checked/unchecked conversions, 63
deserialization, 711
Tic-Tac-Toe source code, 901–905
time slicing, 732
timers
prior to async/await patterns, 937–942
threading, 841–843
ToArray() method, 588
ToCharArray() method, 80
token cancellation, 801
ToLookup() method, 588
ToString() method, 65, 357
enum conversions, 362
overriding, 372–373
time slicing, 732
trees, expressions, 496, 523–530
triggering events, 548
ToList() method, 639
troubleshooting. See also errors
arrays, 81–82
multithreading, 734–736
try blocks, 197
TryGetMember() method, 723
TryGetPhoneNumber() method, 179, 180
TryParse() method, 66–67, 207–208, 684
TrySetMember() method, 723
placeholders, 121
public constants, 268
returns, 157, 162
types, 57–58, 177, 339–340
avoiding boxing, 356–357
boxing, 349–357
default operator, 348
eenums, 358–368
instantiation generics, 491–492
interface inheritance, 348–349
lock statements, 353–355
new operator, 347–348
nullable, 447–448
structs, 340–349
Unicode characters, descending order, 98
XOR operators, 118
variables applying, 16–17
assigning, 15–16
callers, matching with parameter names, 176
declaring, 11, 13, 14–15
global, 256
instances, 217
local, 14. See also local variables
declaring, 165–166
implicitly typed, 55–57
multiple threads, 815–816
scope, 114
loops, 130, 521–522
outer, 518–519, 520–521
parameters, defining index operators, 658–659
ranges, 615
reference types, 341–345
Venn diagrams, join operations, 593
verbatim string literals, 47
VerifyCredentials() method, 333
verifying underlying types, 309–310
versioning assemblies, 889
.NET (Microsoft), 26–28
serialization, 710–713
VES (Virtual Execution System), 24, 876, 877, 881, 895
viewing assemblies, metadata, 678. See also reflection
Virtual Execution System. See VES
virtual fields, properties, 240–242
virtual memory, 851
virtual method defaults, 291
virtual modifiers, 290–295
virtual software, 872
VirtualAllocEx API, 850–852
Visual Basic arrays, redimensioning, 78
classes, accessing, 222
global methods, 164
global variables/functions, 256
implicitly typed variables, 565
line-based statements, 11
void, 55
Visual Studio Project Wizard, 690
void, 53, 54–55
volatile modifier, declaring fields as, 823–824
vulnerabilities, buffer overruns, 883
Wait() method, 750
WaitAll() method, 831
WaitAny() method, 831
WaitForExit() method, 784
#warning preprocessor directive, 148–150
warnings
nowarn:<warn list> option, 149–150
preprocessor directives, 148–150
WCF (Windows Communication Foundation), 27
weak references, 408–410
WebRequest.GetResponseAsync() method, 774
well-formed types, 371–385
assemblies, referencing, 393–398
garbage collection, 407–410
lazy initialization, 419–421
namespaces, defining, 398–402
object, overriding members, 371–385
operators, overloading, 385–393
resource cleanup, 410–419
XML comments, 402–407
WF (Windows Workflow), 27
WHERE clause, 615
filtering, 623–624
where keyword, 465
where() method, 509
filtering with, 580–591
while loops, control flow statements, 127–129
whitespace, definitions, 12–13
Win32, error handling, 854–855
Windows Communication Foundation. See WCF
Error Reporting, 425
executable assemblies, 394
Forms, 932–934
Presentation Foundation. See WPF
UIs (user interfaces), 790–792, 932–936
Workload. See WF
WinRT (Windows Runtime), 27, 845, 895
libraries, 846–849
wizards, Project Wizard (Visual Studio), 690
work stealing, 798
WPF (Windows Presentation Foundation), 27, 934–936
wrappers, APIs (P/Invoke), 861
wrapping exceptions, 438–442
write-only properties, 237–239
WriteLine() method, 157
WriteWebRequestSizeAsync() method, 779
xcopy deployment, 889
XML (Extensible Markup Language)
comments, 402–407, 678
delimited comments, 22
elements, runtime binding, 719–720
overview of, 23–24
single-line comments, 22
XNA (Microsoft), 878
XOR (exclusive OR) operators, 118, 122
yield break statements, 670–671
yield keyword, 615
yield return statements, 665, 667–669, 674
yielding values from iterators, 662–664
ZipCompression class, 325