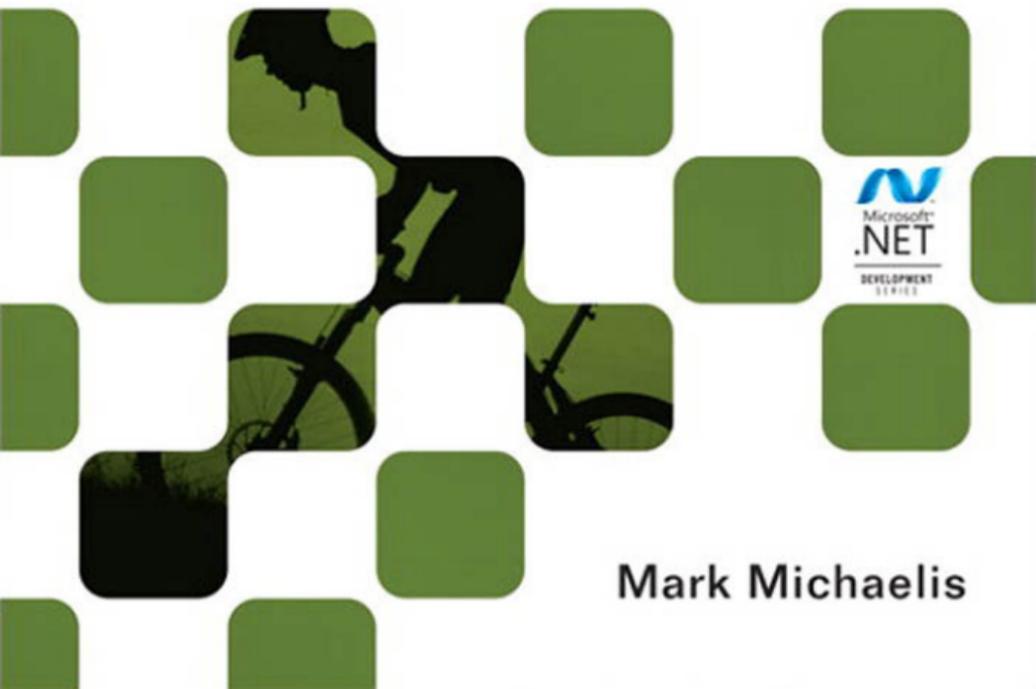


"If you want to be a C# developer, or if you want to enhance your C# programming skills, there is no more useful tool than a well-crafted book on the subject. You are holding such a book in your hands."



—From the Foreword by **Charlie Calvert**,
Community Program Manager, Visual C#, Microsoft

Essential C# 4.0



Mark Michaelis

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

The .NET logo is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

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

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

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Michaelis, Mark.

Essential C# 4.0 / Mark Michaelis.

p. cm.

Includes index.

ISBN 978-0-321-69469-0 (pbk. : alk. paper)

1. C# (Computer program language) I. Title.

QA76.73.C154M5237 2010

005.13'3—dc22

2009052592

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-69469-0

ISBN-10: 0-321-69469-4

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, March 2010



Foreword

MARK MICHAELIS'S OVERVIEW OF THE C# language has become a standard reference for developers. In this, its third edition, programmers will find a thoughtful, well-written guide to the intricacies of one of the world's most popular computer languages. Having laid a strong foundation in the earlier editions of this book, Mark adds new chapters that explain the latest features in both C# and the .NET Framework.

Two of the most important additions to the book cover the latest tools for parallel programming and the new dynamic features found in C# 4.0. The addition of dynamic features to the C# language will give developers access to late-bound languages such as Python and Ruby. Improved support for COM Interop will allow developers to access Microsoft Office with an intuitive and easy-to-use syntax that makes these great tools easy to use. Mark's coverage of these important topics, along with his explanation of the latest developments in concurrent development, make this an essential read for C# developers who want to hone their skills and master the best and most vital parts of the C# language.

As the community PM for the C# team, I work to stay attuned to the needs of our community. Again and again I hear the same message: "There is so much information coming out of Microsoft that I can't keep up. I need access to materials that explain the technology, and I need them presented in a way that I can understand." Mark Michaelis is a one-man solution to a C# developer's search for knowledge about Microsoft's most recent technologies.



I first met Mark at a breakfast held in Redmond, Washington, on a clear, sunny morning in the summer of 2006. It was an early breakfast, and I like to sleep in late. But I was told Mark was an active community member, and so I woke up early to meet him. I'm glad I did. The distinct impression he made on me that morning has remained unchanged over the years.

Mark is a tall, athletic man originally from South Africa, who speaks in a clear, firm, steady voice with a slight accent that most Americans would probably find unidentifiable. He competes in Ironman triathlons and has the lean, active look that one associates with that sport. Cheerful and optimistic, he nevertheless has a businesslike air about him; one has the sense that he is always trying to find the best way to fit too many activities into a limited time frame.

Mark makes frequent trips to the Microsoft campus to participate in reviews of upcoming technology or to consult on a team's plans for the future. Flying in from his home in Spokane, Washington, Mark has clearly defined agendas. He knows why he is on the campus, gives his all to the work, and looks forward to heading back home to his family in Spokane. Sometimes he finds time to fit in a quick meeting with me, and I always enjoy them. He is cheerful and energetic, and nearly always has something provocative to say about some new technology or program being developed by Microsoft.

This brief portrait of Mark tells you a good deal about what you can expect from this book. It is a focused book with a clear agenda written in a cheerful, no-nonsense manner. Mark works hard to discover the core parts of the language that need to be explained and then he writes about them in the same way that he speaks: with a lucid, muscular prose that is easy to understand and totally devoid of condescension. Mark knows what his audience needs to hear and he enjoys teaching.

Mark knows not only the C# language, but also the English language. He knows how to craft a sentence, how to divide his thoughts into paragraphs and subsections, and how to introduce and summarize a topic. He consistently finds clear, easy-to-understand ways to explain complex subjects.

I read the first edition of Mark's book cover to cover in just a few evenings of concentrated reading. Like the current volume, it is a delight to

read. Mark selects his topics with care, and explains them in the simplest possible terms. He knows what needs to be included, and what can be left out. If he wants to explore an advanced topic, he clearly sets it apart from the rest of the text. He never shows off by first parading his intellect at the expense of our desire to understand.

A centrally important part of this new edition of the book continues to be its coverage of LINQ. For many developers the declarative style of programming used by LINQ is a new technology that requires developing new habits and new ways of thinking.

C# 3.0 contained several new features that enable LINQ. A main goal of the book is to lay out these features in detail. Explaining LINQ and the technologies that enable it is no easy task, and Mark has rallied all his formidable skills as a writer and teacher to lay this technology out for the reader in clear and easy-to-understand terms.

All the key technologies that you need to know if you want to understand LINQ are carefully explained in this text. These include

- Partial methods
- Automatic properties
- Object initializers
- Collection initializers
- Anonymous types
- Implicit local variables (`var`)
- Lambdas
- Extension methods
- Expression trees
- `IEnumerable<T>` and `IQueryable<T>`
- LINQ query operators
- Query expressions

The march to an understanding of LINQ begins with Mark's explanations of important C# 2.0 technologies such as generics and delegates. He then walks you step by step through the transition from delegates to lambdas. He explains why lambdas are part of C# 3.0 and the key role they play

in LINQ. He also explains extension methods, and the role they play in implementation of the LINQ query operators.

His coverage of C# 3.0 features culminates in his detailed explanation of query expressions. He covers the key features of query expressions such as projections, filtering, ordering, grouping, and other concepts that are central to an understanding of LINQ. He winds up his chapter on query expressions by explaining how they can be converted to the LINQ query method syntax, which is actually executed by the compiler. By the time you are done reading about query expressions you will have all the knowledge you need to understand LINQ and to begin using this important technology in your own programs.

If you want to be a C# developer, or if you want to enhance your C# programming skills, there is no more useful tool than a well-crafted book on the subject. You are holding such a book in your hands. A text such as this can first teach you how the language works, and then live on as a reference that you use when you need to quickly find answers. For developers who are looking for ways to stay current on Microsoft's technologies, this book can serve as a guide through a fascinating and rapidly changing landscape. It represents the very best and latest thought on what is fast becoming the most advanced and most important contemporary programming language.

—*Charlie Calvert*
Community Program Manager,
Visual C#, Microsoft
January 2010



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 written in the order in which they are executed. The problem with this model is that complexity increases exponentially as the requirements increase. To reduce this complexity, code blocks 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.

1. 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.



2. 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# 4.0 and .NET Framework 4.
3. 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 .NET Development Series provides a wealth of writing on these topics. *Essential C# 4.0* 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 and C# 4.0, 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)
- Standard query operators (see Chapter 14)
- Query expressions (see Chapter 15)
- Dynamic programming (Chapter 17)
- Multithreaded programming with the Task Programming Library (Chapter 18)
- Parallel query processing with PLINQ
- 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# 4.0 is a language book that adheres to the core C# Language 4.0 Specification. To help you understand the various C# constructs, the book 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.

Code Samples

The code snippets in most of this text (see sample listing on the next page) 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 or 4.0 compliance is called out in the C# 3.0 and C# 4.0 indexes at the end of the book.

Here is a sample code listing.

LISTING 1.17: Commenting Your Code

```

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

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

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

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

        /* Display a greeting to the console
           using composite formatting. */ } delimited comment
        System.Console.WriteLine("Your full name is {0} {1}.",
            firstName, lastName);
        // This is the end
        // of the program listing
    }
}

```

The formatting is as follows.

- Comments are shown in italics.

```

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

```

- Keywords are shown in bold.

```

static void Main()

```

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

```
System.Console.Write /* No new Line */ (
```

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

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

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

```
// ...
```

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

OUTPUT 1.4:

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

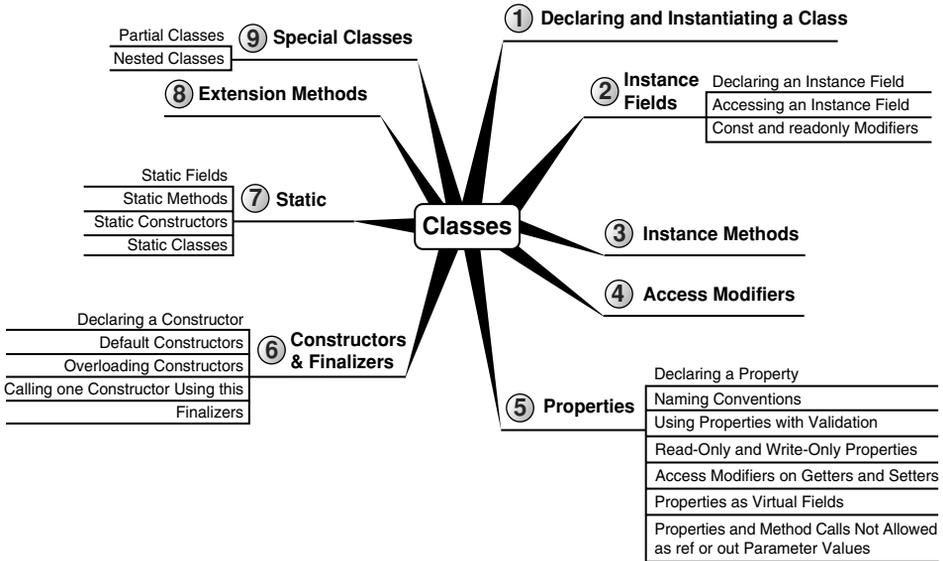
- User input for the program appears in italics.

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 and bonus material at intelliTeecture.com/EssentialCSharp and at informit.com/msdotnetseries.

Mind Maps

Each chapter's introduction includes a **mind map**, which serves as an outline that provides an at-a-glance reference to each chapter's content. Here is an example (taken from Chapter 5).



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

Helpful Notes

Depending on your level of experience, special code blocks and tabs will help you navigate through the text.

- Beginner Topics provide definitions or explanations 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# 4.0*. 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 or C# 4.0 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 a 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—*Synchronization and Other Multithreading Patterns***: 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 the CLI Platform***: This appendix provides instructions for setting up a C# compiler and the platform on which to run the code, Microsoft .NET or Mono.
- **Appendix B—*Full Source Code Listing***: In several cases, a full source code listing within a chapter would have made the chapter too long. To make



these listings still available to the reader, this appendix includes full listings from Chapters 3, 11, 12, 14, and 17.

- *Appendix C—Concurrent Classes from System.Collections.Concurrent*: This appendix provides overview diagrams of the concurrent collections that were added in the .NET Framework 4.
- *Appendixes D-F: C# 2.0, C# 3.0, C# 4.0 Topics*: These appendices provide a quick reference for any C# 2.0, C# 3.0, or C# 4.0 content. They are specifically designed to help programmers quickly get up to speed on C# features.

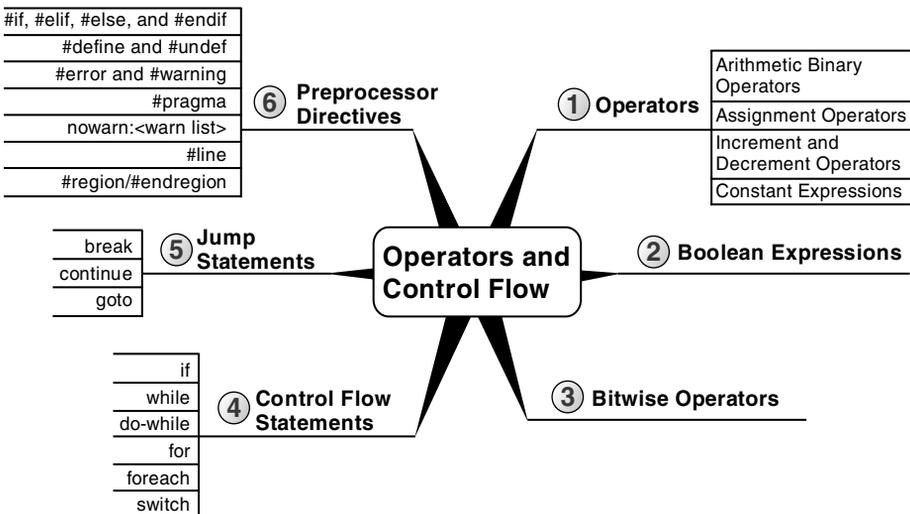
I hope you find this book to be a great resource in establishing your C# expertise and that you continue to reference it for the more obscure areas of C# and its inner workings.

—Mark Michaelis
mark.michaelis.net

3

Operators and Control Flow

IN THIS CHAPTER, you will learn about operators and control flow statements. 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 explicitly) to `bool` and the



advantages of this restriction. The chapter ends with a discussion of the C# “preprocessor” and its accompanying directives.

Operators

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

BEGINNER TOPIC

Operators

Operators specify operations within an expression, such as a mathematical expression, to be performed on a set of values, called **operands**, to produce a new value or result. For example, in Listing 3.1 there are two operands, the numbers 4 and 2, that are combined using the subtraction operator, -. You assign the result to the variable `difference`.

LISTING 3.1: A Simple Operator Example

```
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 variable. In these cases, the unary minus operator (-) comes in handy. For example, Listing 3.2 changes the total current U.S. debt to a negative value to indicate that it is an amount owed.

LISTING 3.2: Specifying Negative Values¹

```
//National Debt to the Penny  
decimal debt = -11719258192538.99M;
```

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

1. As of August 21, 2009, according to www.treasurydirect.gov.

The unary plus operator (+) has rarely² had 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 in order to process an equation: a left-hand side operand and a right-hand side operand. Binary operators also require that the code assign the resultant value to avoid losing it.

Language Contrast: C++ — Operator-Only Statements

Binary operators in C# require an assignment or call; they always return a new result. Neither operand in a binary operator expression can be modified. In contrast, C++ will allow a single statement, such as `4+5`, to compile even without an assignment. In C#, call, increment, decrement, and new object expressions are allowed for operator-only statements.

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

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

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

2. The unary + operator is not defined on a `short`; it is defined on `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal`. Therefore, using it on a `short` will convert it to one of these types as appropriate.

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

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

```
System.Console.WriteLine(
    "{0} / {1} = {2} with remainder {3}",
    numerator, denominator, quotient, remainder);
}
}
```

Output 3.1 shows the results of Listing 3.3.

OUTPUT 3.1:

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

Note the order of associativity when using binary operators. The binary operator order is from left to right. In contrast, the assignment operator order is from right to left. On its own, however, associativity does not specify whether the division will occur before or after the assignment. The order of precedence defines this. The precedence for the operators used so far is as follows:

1. *, /, and %
2. + and -
3. =

Therefore, you can assume that the statement behaves as expected, with the division and remainder operators occurring before the assignment.

If you forget to assign the result of one of these binary operators, you will receive the compile error shown in Output 3.2.

OUTPUT 3.2:

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

BEGINNER TOPIC

Associativity and Order of Precedence

As with mathematics, programming languages support the concept of **associativity**. Associativity refers to how operands are grouped and, therefore, the order in which operators are evaluated. Given a single operator that appears more than once in an expression, the operator associates the first duple and then the next operand until all operators are evaluated. For example, $a-b-c$ associates as $(a-b)-c$, and not $a-(b-c)$.

Associativity applies only when all the operators are the same. When different operators appear within a statement, the **order of precedence** for those operators dictates which operators are evaluated first. Order of precedence, for example, indicates that the multiplication operator be evaluated before the plus operator in the expression $a+b*c$.

Using the Plus Operator with Strings

Operators can also work with types that are not numeric. For example, it is possible to use the plus 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 plus operator with strings that require localization. Composite formatting is preferred (refer to Chapter 1).

Using Characters in Arithmetic Operations

When introducing the `char` type in the preceding chapter, I 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 `0x34`, or 52 in base 10. Adding 3 and 4 in Listing 3.5 results in a hexadecimal value of `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

```
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 one another. For example, the letter `f` is three characters away from the letter `c`. You can determine this value by subtracting the letter `c` from the letter `f`, as Listing 3.6 demonstrates.

LISTING 3.6: Determining the Character Difference between Two Characters

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

A `float`, with seven digits of precision, can hold the value 1,234,567 and the value 0.1234567. However, if you add these two floats together, the result will be rounded to 1234567, because the decimal portion of the number is past the seven significant digits that a `float` can hold. This type of rounding can become significant, especially with repeated calculations or checks for equality (see the upcoming Advanced Topic, Unexpected Inequality with Floating-Point Types).

Note that inaccuracies can occur with a simple assignment, such as `double number = 140.6F`. 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`; `140.600006103516` is 140.6 as a `float`, but not quite 140.6 when represented as a `double`.

ADVANCED TOPIC

Unexpected Inequality with Floating-Point Types

The inaccuracies of floats can be very disconcerting when comparing values for equality, since they can unexpectedly be unequal. Consider Listing 3.7.

LISTING 3.7: Unexpected Inequality Due to Floating-Point Inaccuracies

```
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: 4.19999980926514 != 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.

OUTPUT 3.6:

```

4.2 != 4.20000006258488
4.2 != 4.20000006258488
(float)4.2M != 4.2F
4.20000006258488 != 4.20000028610229
4.20000006258488 != 4.2
4.2F != 4.2D
4.19999980926514 != 4.2
4.2F != 4.2D

```

The `Assert()` methods are designed to display a debug dialog whenever the parameter evaluates to `false`. However, all of the `Assert()` statements in this code listing will evaluate to `true`. Therefore, 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`. Furthermore, there is not some compounding rounding error. The C# compiler performs the calculations instead of the runtime. Even if you simply assign `4.2F` rather than a calculation, the comparisons will remain unequal.

To avoid unexpected results caused by the inaccuracies of floating-point types, developers should avoid using equality conditionals with these types. Rather, equality evaluations should include a tolerance. One easy way to achieve this is to subtract one value (operand) from the other and then evaluate whether the absolute value of the result is less than the maximum tolerance. Even better is to use the decimal type in place of the float 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 precision data types such as `int` and `decimal`. `float` and `double`, however, allow for certain special values. Consider Listing 3.8, and its resultant output, Output 3.7.

LISTING 3.8: Dividing a Float by Zero, Displaying NaN

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

OUTPUT 3.7:

```
NaN
```

In mathematics, certain mathematical operations are undefined. In C#, the result of dividing `0F` by the value `0` results in “Not a Number,” and all attempts to print the output of such a number will result in `NaN`. Similarly, taking the square root of a negative number (`System.Math.Sqrt(-1)`) will result in `NaN`.

A floating-point number could overflow its bounds as well. For example, the upper bound of a `float` type is `3.4E38`. 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 `-3.4E38`, and assigning 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

```
// 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, then 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`.

Parenthesis Operator

Parentheses allow you to group operands and operators so that they are evaluated together. This is important because it provides a means of overriding the default order of precedence. For example, the following two expressions evaluate to something completely different:

```
(60 / 10) * 2
60 / (10 * 2)
```

The first expression is equal to 12; the second expression is equal to 3. In both cases, the parentheses affect the final value of the expression.

Sometimes the parenthesis operator does not actually change the result, because the order-of-precedence rules apply appropriately. However, it is

often still a good practice to use parentheses to make the code more readable. This expression, for example:

```
fahrenheit = (celsius * 9.0 / 5.0) + 32.0;
```

is easier to interpret confidently at a glance than this one is:

```
fahrenheit = celsius * 9.0 / 5.0 + 32.0;
```

Developers should use parentheses to make code more readable, disambiguating expressions explicitly instead of relying on operator precedence.

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. Other assignment operators combine common binary operator calculations with the assignment operator. Take Listing 3.10, for example.

LISTING 3.10: Common Increment Calculation

```
int x;  
x = x + 2;
```

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

```
int x;  
x += 2;
```

This code, therefore, is equivalent to Listing 3.10.

Numerous other combination assignment operators exist to provide similar functionality. You can use the assignment operator in conjunction with not only addition, but also subtraction, multiplication, division, and the remainder operators, as Listing 3.12 demonstrates.

LISTING 3.12: Other Assignment Operator Examples

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

Increment and Decrement Operators (++ , --)

C# includes special 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

```
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

```
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 ASCII Value in Descending Order

```
char current;
int asciiValue;

// Set the initial value of current.
```

```
current='z';

do
{
    // Retrieve the ASCII value of current.
    asciiValue = current;
    System.Console.WriteLine("{0}={1}\t", current, asciiValue);

    // 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:

```
z=122   y=121   x=120   w=119   v=118   u=117   t=116   s=115   r=114
q=113   p=112   o=111   n=110   m=109   l=108   k=107   j=106   i=105
h=104   g=103   f=102   e=101   d=100   c=99    b=98    a=97
```

The increment and decrement operators are used to count how many times to perform a particular operation. 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 “next” or “previous” for that data type.

Just as with the assignment operator, the increment operator also returns a value. In other words, it is possible to use the assignment operator simultaneously with the increment or decrement operator (see Listing 3.16 and Output 3.10).

LISTING 3.16: Using the Post-Increment Operator

```
int count;
int result;
count = 0;
result = count++;
System.Console.WriteLine("result = {0} and count = {1}",
    result, count);
```

OUTPUT 3.10:

```
result = 0 and count = 1
```

You might be surprised that `result` is assigned the value in `count` *before* `count` is incremented. In other words, `result` ends up with a value of 0 even though `count` ends up with a value of 1.

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, which affects how the code functions. If you want the value of `result` to include the increment (or decrement) calculation, 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

```
int count;
int result;
count = 0;
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 = 1 and count = 1
```

In this example, the increment operator appears before the operand so the value returned is the value assigned to the variable after the increment. If `x` is 1, then `++x` will return 2. However, if a postfix operator is used, `x++`, the value returned by the expression will still be 1. Regardless of whether the operator is postfix or prefix, the resultant value of `x` will be incremented. 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

```
class IncrementExample
{
    public static void Main()
    {
        int x;
```

```
x = 1;
// Display 1, 2.
System.Console.WriteLine("{0}, {1}, {2}", x++, x++, x);
// x now contains the value 3.

// Display 4, 5.
System.Console.WriteLine("{0}, {1}, {2}", ++x, ++x, x);
// x now contains the value 5.
// ...
}
}
```

OUTPUT 3.12:

```
1, 2, 3
4, 5, 5
```

As Listing 3.18 demonstrates, where the increment and decrement operators appear relative to the operand can affect the result returned from the operator. Pre-increment/decrement operators return the result after incrementing/decrementing the operand. Post-increment/decrement operators return the result before changing the operand. Developers should 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.

 **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 detail).

Constant Expressions (const)

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 of constant operands. For example, the number of seconds in a day can be assigned as a constant expression whose result can then be used in other expressions.

The `const` keyword in Listing 3.19 locks the value at compile time. Any attempt to modify the value later in the code results in a compile error.

LISTING 3.19:

```

// ...
public long Main()
{
    const int secondsPerDay = 60 * 60 * 24;
    const int secondsPerWeek = secondsPerDay * 7;
    // ...
}

```

Constant Expression
Constant

Note that even the value assigned to `secondsPerWeek` is a constant expression, because the operands in the expression are also constants, so the compiler can determine the result.

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.

TABLE 3.1: Control Flow Statements

Statement	General Syntax Structure	Example
if statement	<code>if(boolean-expression) embedded-statement</code>	<pre>if (input == "quit") { System.Console.WriteLine("Game end"); return; }</pre>
	<code>if(boolean-expression) embedded-statement else embedded-statement</code>	<pre>if (input == "quit") { System.Console.WriteLine("Game end"); return; } else GetNextMove();</pre>
while statement	<code>while(boolean-expression) embedded-statement</code>	<pre>while(count < total) { System.Console.WriteLine("count = {0}", count); count++; }</pre>

Continues

TABLE 3.1: Control Flow Statements (*Continued*)

Statement	General Syntax Structure	Example
do while statement	<pre>do embedded-statement while(boolean-expression);</pre>	<pre>do { System.Console.WriteLine("Enter name:"); input = System.Console.ReadLine(); } while(input != "exit");</pre>
for statement	<pre>for(for-initializer; boolean-expression; for-iterator) embedded-statement</pre>	<pre>for (int count = 1; count <= 10; count++) { System.Console.WriteLine("count = {0}", count); }</pre>
Foreach statement	<pre>foreach(type identifier in expression) embedded-statement</pre>	<pre>foreach (char letter in email) { if(!insideDomain) { if (letter == '@') { insideDomain = true; } continue; } System.Console.Write(letter); }</pre>
continue statement	<pre>continue;</pre>	<pre> } System.Console.Write(letter); }</pre>

TABLE 3.1: Control Flow Statements (*Continued*)

Statement	General Syntax Structure	Example
switch statement	<pre> switch(governing-type-expression) { ... case const-expression: statement-list jump-statement default: statement-list jump-statement } </pre>	<pre> switch(input) { Case "exit": case "quit": System.Console.WriteLine("Exiting app...."); break; case "restart": Reset(); goto case "start"; case "start": GetMove(); break; } </pre>
break statement	<pre> break; </pre>	<pre> default: </pre>
goto statement	<pre> goto identifier; </pre> <hr/> <pre> goto case const-expression; </pre> <hr/> <pre> goto default; </pre>	<pre> System.Console.WriteLine(input); break; } </pre>

An embedded-statement in Table 3.1 corresponds to any statement, including a code block (but not a declaration statement or a label).

Each C# control flow statement in Table 3.1 appears in the tic-tac-toe³ program found in Appendix B. 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# Pre-processor 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 returns a Boolean), and if the result is true, the following statement (or block) is executed. The general form is as follows:

```
if(condition)
    consequence
[else
    alternative]
```

There is also an optional `else` clause for when the Boolean expression is false. Listing 3.20 shows an example.

LISTING 3.20: if/else Statement Example

```
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();
    }
}
```

3. Known as noughts and crosses to readers outside the United States.

```
    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

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

```
24         "maximum turns.", input);
25     else
26         if(input>9)
27             // Input is greater than 9.
28             System.Console.WriteLine(
29                 "Tic-tac-toe has fewer than {0}" +
30                 "maximum turns.", input);
31     else
32         // Input equals 9.
33         System.Console.WriteLine(
34             "Correct, " +
35             "tic-tac-toe has a max. of 9 turns.");
36 }
37 }
```

Output 3.13 shows the results of Listing 3.21.

OUTPUT 3.13:

```
What's 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

```
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.

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

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

With curly braces, however, we can combine statements into a single unit called a **code block**, allowing the execution of multiple statements for a condition. 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

```
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.

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

```
if(radius>=0)
    area = 3.14*radius*radius;
    System.Console.WriteLine( // Logic Error!! Needs code block.
        "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

```
if(radius>=0)
{
    area = 3.14*radius*radius;
}
System.Console.WriteLine(    // Error!! Place within code block.
    "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.

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.

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`.

Scope and Declaration Space

Scope and **declaration space** are hierarchical contexts bound by a **code block**. Scope is the region of source code in which it is legal to refer to an item by its unqualified name because the name reference is unique and unambiguous.

The area in which declaring the name is unique is the declaration space. C# prevents two local variable declarations with the same name from appearing in the same declaration space. Similarly, it is not possible to declare two methods with the signature of `Main()` within the same class (declaration scope for the method name includes the full signature). The scope identifies what within a code block an unqualified name refers to; the declaration scope specifies the region in which declaring something with the same name will cause a conflict.

Scope restricts visibility. A local variable, for example, is not visible outside its defining method. Similarly, code that declares a variable in an `if` block makes the variable inaccessible outside the `if` block (even in the same method). In Listing 3.27, defining `message` inside the `if` statement restricts its scope to the statement only. To avoid the error, you must declare the string outside the `if` statement.

LISTING 3.27: Variables Inaccessible Outside Their Scope

```
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
```

Declaration space cascades down to child (or embedded) code blocks within a method. 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. The declaration space is the parent code block of a variable, including any child blocks within the parent code block. From Listing 3.27, because `args` and `playerCount` are declared within the method code block, they cannot be used again within declarations anywhere within the method.

Scope is also bound by the parent code block. The name `message` applies only within the `if` block, not outside it. Similarly, `playerCount` refers to the same variable throughout the method following where the variable is declared—including within both the `if` and `else` child blocks.

Boolean Expressions

The portion of the `if` statement within parentheses is the **Boolean expression**, sometimes referred to as a **conditional**. In Listing 3.28, the Boolean expression is highlighted.

LISTING 3.28: Boolean Expression

```
if(input < 9)
{
    // Input is less than 9.
    System.Console.WriteLine(
        "Tic-tac-toe has more than {0}" +
        " maximum turns.", input);
}
// ...
```

Boolean expressions appear within many control flow statements. The key characteristic is that they always evaluate to `true` or `false`. For `input < 9` to be allowed as a Boolean expression, it must return a `bool`. The compiler disallows `x=42`, for example, because it assigns `x`, returning the new value, instead of checking whether `x`'s value is 42.

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

The significant feature of Boolean expressions in C# is the elimination of a common coding error that historically appeared in C/C++. In C++, Listing 3.29 is allowed.

LISTING 3.29: C++, But Not C#, Allows Assignment as a Boolean Expression

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

Although 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.

Relational and Equality Operators

Included in the previous code examples was the use of relational operators. In those examples, relational operators were used to evaluate user input. Table 3.2 lists all the relational and equality operators.

TABLE 3.2: Relational and Equality Operators

Operator	Description	Example
<	Less than	<code>input<9;</code>
>	Greater than	<code>input>9;</code>
<=	Less than or equal to	<code>input<=9;</code>
>=	Greater than or equal to	<code>input>=9;</code>
==	Equality operator	<code>input==9;</code>
!=	Inequality operator	<code>input!=9;</code>

In addition to determining whether a value is greater than or less than another value, operators are also required to determine equivalency. You test for equivalence by using equality operators. In C#, the syntax follows the C/C++/Java pattern with `==`. 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, `!=`.

The relational and equality operators are binary operators, meaning they compare two operands. More significantly, they always return a Boolean data type. Therefore, you can assign the result of a relational operator to a `bool` variable, as shown in Listing 3.30.

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

```
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

```
if (input == "" || input == "quit")
{
    System.Console.WriteLine("Player {0} quit!!", currentPlayer);
    break;
}
```

Logical Boolean Operators

Logical operators have Boolean operands and return a Boolean result. Logical operators allow you to combine multiple Boolean expressions to form other Boolean expressions. The logical operators are `||`, `&&`, and `^`, corresponding to OR, AND, and exclusive OR, respectively.

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 returns a true value if *either* one of them is true (see Listing 3.32).

LISTING 3.32: Using the OR Operator

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

Note that with the Boolean OR operator, it is not necessary to evaluate both sides of the expression. Like all operators in C#, the OR operators go from left to right, so if the left portion of the expression evaluates to true, then the right portion is ignored. Therefore, if `hourOfDay` has the value 33 then `(hourOfDay > 23)` will return true and the OR operator ignores the second half of the expression—**short-circuiting** it. Short-circuiting an expression also occurs with the Boolean AND operator.

AND Operator (&&)

The Boolean AND operator, `&&`, evaluates to true only if both operands evaluate to true. If either operand is false, the combined expression will return false.

Listing 3.33 displays that it is time for work as long as the current hour is both greater than 10 and less than 24.⁴ As you saw with the OR operator, the AND operator will not always evaluate the right side of the expression. If the left operand returns false, then the overall result will be false regardless of the right operand, so the runtime ignores the right operand.

LISTING 3.33: Using the AND Operator

```
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.

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.

4. The typical hours that programmers work.

TABLE 3.3: Conditional Values for the XOR Operator

Left Operand	Right Operand	Result
True	True	False
True	False	True
False	True	True
False	False	False

Logical Negation Operator (!)

Sometimes called the NOT operator, the **logical negation operator**, `!`, inverts a `bool` data type to its opposite. 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

```
bool result;  
bool valid = false;  
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 a new 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 is a question mark (`?`), and the general format is as follows:

```
conditional? consequence: alternative;
```

The conditional operator is a ternary operator, because it has three operands: `conditional`, `consequence`, and `alternative`. If the `conditional`

evaluates to true, then the conditional operator returns consequence. Alternatively, if the conditional evaluates to false, then it returns alternative.

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

```
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 statement. If the result is true, then the conditional operator returns the value 1. Otherwise, it returns 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.

Use the conditional operator sparingly, because readability is often sacrificed and a simple if/else statement may be more appropriate.

Null Coalescing Operator (??)

Starting with C# 2.0, there is a shortcut to the conditional operator when checking for null. The shortcut is the **null coalescing operator**, and it evaluates an expression for null and returns a second expression if the value is null.

```
expression1?? expression2;
```

If the expression (expression1) is not null, then expression1 is returned. In other words, the null coalescing operator returns expression1 directly unless expression1 evaluates to null, in which case expression2 is returned. 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

```
string fileName;  
// ...  
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`.

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.

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

FIGURE 3.1: Corresponding Placeholder Values

In many instances, 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.

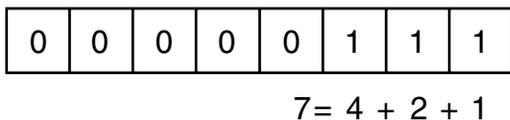


FIGURE 3.2: Calculating the Value of an Unsigned Byte

The binary translation just described is significantly different for signed numbers. Signed numbers (long, short, int) are represented using a 2s 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 left-most 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.

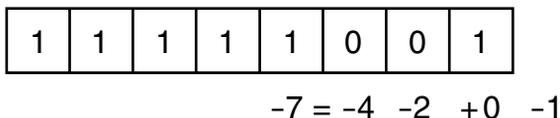


FIGURE 3.3: Calculating the Value of a Signed Byte

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

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.”

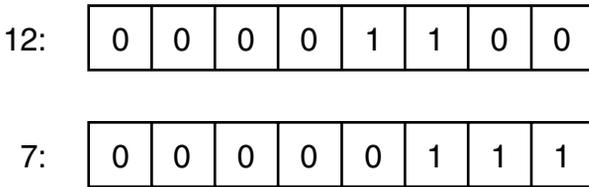


FIGURE 3.4: The Numbers 12 and 7 Represented in Binary

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 `00000100`, 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

```
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.

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

Bitwise 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

```
byte and, or, xor;
and = 12;
and &= 7;    // and = 4
or = 12;
or |= 7;     // or = 15
xor = 12;
xor ^= 7;    // xor = 11
System.Console.WriteLine(
    "and = {0} \nor = {1} \nxor = {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 `1111 1111 1111 1111 1111 1111 1111 1110` and `~(1<<31)` returns `0111 1111 1111 1111 1111 1111 1111 1111`.

Control Flow Statements, Continued

With the additional coverage of Boolean expressions, it's time to consider more of the control flow statements supported by C#. As indicated in the introduction to this chapter, many of these statements will be familiar to experienced programmers, so you can skim this section for information specific to C#. Note in particular the `foreach` loop, as this may be new to many programmers.

The while and do/while Loops

Until now, you have learned how to write programs that do something only once. However, one of the important capabilities of the computer is that it can perform the same operation multiple times. In order to do this, you need to create an instruction loop. The first instruction loop I will discuss is the `while` loop. The general form of the `while` statement is as follows:

```
while(boolean-expression )  
    statement
```

The computer will repeatedly execute `statement` as long as `boolean-expression` evaluates to `true`. If the expression evaluates to `false`, then code execution continues at the instruction following `statement`. (Note that `statement` will continue to execute even if it causes `boolean-expression` to be `false`. It isn't until the `boolean-expression` is reevaluated within the `while` condition that the loop exits.) The Fibonacci calculator shown in Listing 3.41 demonstrates the `while` loop.

LISTING 3.41: while Loop Example

```
class FibonacciCalculator  
{  
    static void Main()  
    {  
        decimal current;  
        decimal previous;  
        decimal temp;  
        decimal input;  
  
        System.Console.Write("Enter a positive integer:");
```

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

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

// While the current Fibonacci number in the series is
// less than the value input by the user.
while(current <= input)
{
    temp = current;
    current = previous + current;
    previous = temp;
}

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 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 types of statements that cause a block of code to execute repeatedly. The term *loop* refers to the block of code 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 loop 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 indeterminate when iterating begins. This pattern occurs most commonly when repeatedly prompting a user for input. Listing 3.42 is taken from the tic-tac-toe program.

LISTING 3.42: do/while Loop Example

```
// Repeatedly request player to move until he  
// enter a valid position on the board.  
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:

```
do  
    statement  
while(boolean-expression );
```

As with all the control flow statements, the code blocks are not part of the general form. However, a code block is generally used in place of a single statement in order to allow multiple statements.

expression within the for loop corresponds to a statement. (It is easy to remember that the separation character between expressions is a semicolon and not a comma, because each expression could be a statement.)

You write a for loop generically as follows:

```
for(initial; boolean-expression; Loop)
    statement
```

Here is a breakdown of the for loop.

- The `initial` expression 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. It is possible, for example, to declare the variable beforehand and simply initialize it in the for loop. Variables declared here, however, are bound within the scope of the for statement.
- The `boolean-expression` portion of the for loop specifies an end condition. The loop exits when this condition is false in a manner similar to the `while` loop's termination. The for loop will repeat only as long as `boolean-expression` evaluates to true. In the preceding example, the loop exits when `count` increments 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 Boolean expression is evaluated. During the sixty-fourth iteration, `count` increments to 64, causing `boolean-expression` to be false and, therefore, terminating the loop. Because each expression may be thought of as a separate statement, each expression in the for loop is separated by a semicolon.
- The `statement` portion of the for loop is the 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. Verify that `count` is less than 64.

3. Calculate `bit` and display it.
4. Shift the mask.
5. Increment `count` by one.
6. If `count < 64`, then jump back to line 3.

The `for` statement doesn't require any of the elements between parentheses. `for(;;){ ... }` is perfectly valid; although there still needs to be a means to escape from the loop to avoid executing infinitely. Similarly, the initial and loop expressions can be a complex expression involving multiple subexpressions, as shown in Listing 3.44.

LISTING 3.44: `for` Loop Using Multiple Expressions

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

The results of Listing 3.44 appear in Output 3.22.

OUTPUT 3.22:

```
0<5    1<4    2<3    3>2    4>1    5>0
```

In this case, the comma behaves exactly as it does in a declaration statement, one that declares and initializes multiple variables. However, programmers should avoid complex expressions such as this one because they are difficult to read and understand.

Generically, you can write the `for` loop as a `while` loop, as shown here:

```
initial;
while(boolean-expression)
{
    statement;
    loop;
}
```

BEGINNER TOPIC

Choosing between for and while Loops

Although you can use the two statements interchangeably, generally you would use the for loop whenever there is some type of counter, and the total number of iterations is known when the loop is initialized. In contrast, you would typically use the while loop when iterations are not based on a count or when the number of iterations is indeterminate when iterating commences.

The foreach Loop

The last loop statement within the C# language is `foreach`. `foreach` is designed to iterate through a collection of items, setting a variable to represent each item in turn. During the loop, operations may be performed on the item. One feature of the `foreach` loop is that it is not possible to accidentally miscount and iterate over the end of the collection.

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.
- `variable` is a read-only variable into which the `foreach` construct will automatically assign the next item within the collection. The scope of the variable is limited to the `foreach` loop.
- `collection` is an expression, such as an array, representing multiple items.
- `statement` is the code that executes for each iteration within the `foreach` 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

```
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 != '0' && 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 foreach statement block. The `if` statement determines whether the value of `cell` is `'0'` or `'X'`. If it is neither, then 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.

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

Given a variable to compare and a list of constant values to compare against, the switch statement is simpler to read and code than the if statement. The switch statement looks like this:

```
switch(test-expression)
{
    [case option-constant:
        statement
    [default:
        statement]
}
```

Here is a breakdown of the switch statement.

- *test-expression* returns a value that is compatible with the governing types. Allowable governing data types are `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, and an `enum` type (covered in Chapter 8).
- *constant* is any constant expression compatible with the data type of the governing type.
- *statement* is one or more statements to be executed when the governing type expression equals the constant value. The statement or statements must have no reachable endpoint. In other words, the statement, or last of the statements if there are more than one, must be a jump statement such as a `break`, `return`, or `goto` statement. If the switch statement appears within a loop, then `continue` is also allowed.

A switch statement should have at least one case statement or a default statement. In other words, `switch(x){}` will generate a warning.

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;
            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 governing type expression. Since `input` is a string, all of the constants are strings. If the value of `input` is 1, 2, ... 9, then 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, it immediately jumps to the instruction following the `switch` statement.

The next portion of the switch looks for "" or "quit", and sets `valid` to true if input equals one of these values. Ultimately, the `default` label is executed if no prior case constant was equivalent to the governing type.

There are several things to note about the switch statement.

- Placing nothing within the switch block will generate a compiler warning, but the statement will still compile.
- `default` does not have to appear last within the switch statement. case statements appearing after `default` are evaluated.
- When you use multiple constants for one case statement, they should appear consecutively, as shown in Listing 3.47.
- The compiler requires a jump statement (usually a `break`).

Language Contrast: C++ – switch Statement Fall-through

Unlike C++, C# does not allow a switch statement to fall through from one case block to the next if the case includes a statement. A jump statement is always required following the statement within a case. The C# founders believed it was better to be explicit and require the jump statement in favor of code readability. If programmers want to use a fall-through semantic, they may do so explicitly with a `goto` statement, as demonstrated in the section The `goto` Statement, later in this chapter.

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 conditional expression 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, the execution path

immediately jumps to the first instruction following the loop. 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 };

        // 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.


```
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

In some instances, you may have 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 use the `continue` statement to jump to the end of the current iteration and begin the next iteration. The C# `continue` statement allows you to exit the current iteration (regardless of which additional statements remain) and jump to the loop conditional. At that point, if the loop conditional remains 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

```
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 need to use a `continue` statement to jump to the next character in the email address.

In general, you can 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 exit points within the 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

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

The goto Statement

With the advent of object-oriented programming and the prevalence of well-structured code, the existence of a goto statement within C# seems like an aberration 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

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

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

// ...
}
```

Output 3.26 shows the results of Listing 3.52.

OUTPUT 3.26:

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

As demonstrated in Listing 3.52, `goto` statements are ugly. In this particular example, this is the only way to get the desired behavior of a `switch` statement. Although you can use `goto` statements outside `switch` statements, they generally cause poor program structure and you should deprecate them in favor of a more readable construct. Note also that you cannot use a `goto` statement to jump from outside a `switch` statement into a label within a `switch` statement. More generally, C# prevents using `goto` *into* something, and allows its use only *within* or *out of* something. By making this restriction, C# avoids most of the serious `goto` abuses available in other languages.

C# Preprocessor Directives

Control flow statements evaluate conditional 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++ contain a **preprocessor**, a separate utility from the compiler that sweeps over code, performing actions 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

Statement or Expression	General Syntax Structure	Example
#if directive	#if preprocessor-expression code #endif	#if CSHARP2 Console.Clear(); #endif
#elif directive	#if preprocessor-expression1 code #elif preprocessor-expression2 code #endif	#if LINUX ... #elif WINDOWS ... #endif
#else directive	#if code #else code #endif	#if CSHARP1 ... #else ... #endif
#define directive	#define conditional-symbol	#define CSHARP2
#undef directive	#undef conditional-symbol	#undef CSHARP2
#error directive	#error preproc-message	#error Buggy implementation
#warning directive	#warning preproc-message	#warning Needs code review
#pragma directive	#pragma warning	#pragma warning disable 1030
#line directive	#line org-line new-line #line default	#line 467 "TicTacToe.cs" ... #line default
#region directive	#region pre-proc-message code #endregion	#region Methods ... #endregion

Excluding and Including Code (`#if`, `#elif`, `#else`, `#endif`)

Perhaps the most common use of preprocessor directives is in controlling when and how code is included. For example, to write code that could be compiled by both C# 2.0 and later compilers and the prior version 1.2 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

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

In this case, you call the `System.Console.Clear()` method, which is available only in the 2.0 CLI version and later. 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

```
#if LINUX
...
#elif WINDOWS
...
#endif
```

Defining Preprocessor Symbols (#define, #undef)

You can define a preprocessor symbol in two ways. The first is with the #define directive, as shown in Listing 3.55.

LISTING 3.55: A #define Example

```
#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

```
#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

```
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

```

...
#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).

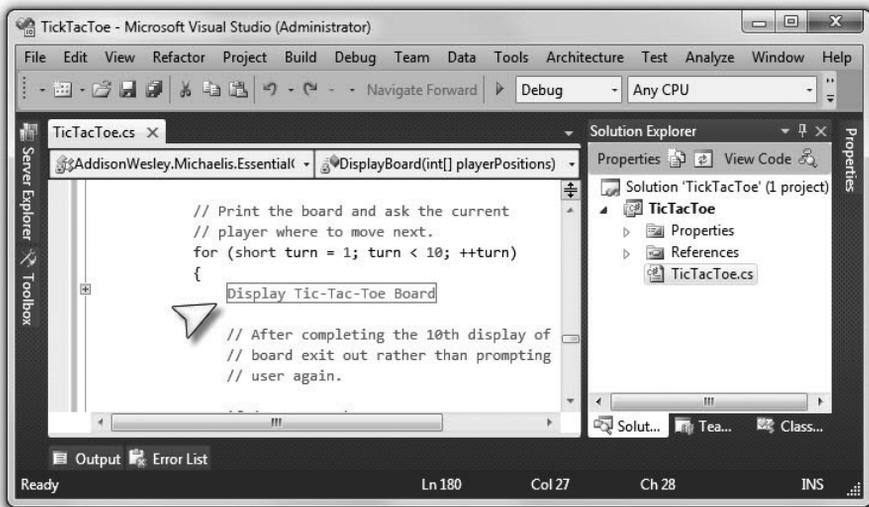


FIGURE 3.5: Collapsed Region in Microsoft Visual Studio .NET

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 constant expressions. Coverage of all of 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 I discussed the bitwise operators, especially regarding masks.

Operator precedence was discussed earlier in the chapter, but 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*

Category	Operators
Primary	<code>x.y</code> <code>f(x)</code> <code>a[x]</code> <code>x++</code> <code>x--</code> <code>new</code> <code>typeof(T)</code> <code>checked(x)</code> <code>unchecked(x)</code> <code>default(T)</code> <code>delegate{}</code> <code>()</code>
Unary	<code>+</code> <code>-</code> <code>!</code> <code>~</code> <code>++x</code> <code>--x</code> <code>(T)x</code>
Multiplicative	<code>*</code> <code>/</code> <code>%</code>
Additive	<code>+</code> <code>-</code>
Shift	<code><<</code> <code>>></code>
Relational and type testing	<code><</code> <code>></code> <code><=</code> <code>>=</code> <code>is</code> <code>as</code>
Equality	<code>==</code> <code>!=</code>
Logical AND	<code>&</code>
Logical XOR	<code>^</code>
Logical OR	<code> </code>
Conditional AND	<code>&&</code>
Conditional OR	<code> </code>
Null coalescing	<code>??</code>
Conditional	<code>?:</code>
Assignment	<code>=</code> <code>=></code> <code>*=</code> <code>/=</code> <code>%=</code> <code>+=</code> <code>-=</code> <code><<=</code> <code>>>=</code> <code>&=</code> <code>^=</code> <code> =</code>

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

Given coverage of most of the operators, the next topic was control flow statements. The last sections of the chapter detailed the preprocessor directives and the bit operators, which included code blocks, scope, Boolean expressions, and bitwise operators.

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

16-bit characters, 41
42 as strings *versus* as integers, 187

; (semicolons)
 statements without, 10–11
 whitespace, 11–12
~ (bitwise complement)
 operator, 120
(hash) symbol, 139
- (minus) operator, 84–92
+ (plus) operator, 84–92
= (simple assignment) operator, 14
_ (underscore), 15
!= (inequality) operator, 110, 370
! (logical notation) operator, 113
% (remainder) operator, 85
&& (AND) operator, 112, 373
' (single quote), 42
() (cast operator), 375–376
* (multiplication) operator, 85
+ (addition) operator, 85, 371–373
++ (increment) operator, 94–97
-- (decrement) operator, 94–97
/ (division) operator, 85

/// (three-forward-slash), 387
< (less than) operator, 110
<= (less than or equal to) operator, 110
== (equality) operator, 110, 370
> (greater than) operator, 110
>= (greater than or equal to) operator, 110
? (conditional) operator, 113–114
?? (null coalescing) operator, 114–115
@ symbol, 45
\
(backslash), 42
\
(newline) character, 42, 48
^ (exclusive OR) operator, 112
{ } (code blocks), 105–107
| | (OR) operator, 111–112, 373
constraints, 450

A

abstract classes, inheritance, 293–299
abstract members, 294
 declaring, 297
accessibility modifiers, 381
accessing
 arrays, 70

CAS (Code Access Security), 659, 852
class instances with `Me` keyword, 214
instance fields, 210–211
members, referent types, 839
metadata, reflection, 652–662
security code, 25
access modifiers, 220–222
circumventing, 852
classes, 380–381
 on getters and setters, 231–232
 `private`, 275
 `protected`, 276
acronyms, common C#, 862–863
actions, `System.Action`, 483–484
Active Template Library (ATL), 278
adding
 comments, 20–23
 `itemsToDictionary<TKey, TValue>`, 623
 operators, 371–373
addition (+) operator, 85, 371–373
`Add()` method, 543
addresses, pointers and, 830–839

- aggregation
 - multiple inheritance, 280
 - single inheritance, 279
- aliasing, 164–165
 - qualifiers, namespaces, 384–385
- allocating data on call stacks, 836
- AllowMultiple parameter, 674
- ambiguity, avoiding, 213–217
- AND operator (&&), 112, 373
- anonymous functions, 486
- anonymous methods, 480–482
 - internals, 494–495
 - parameterless, 482
- anonymous types, 245–246
 - arrays, initializing, 545–546
 - collection interfaces, 536–538
 - generating, 542–543
 - implicit local variables, 54
 - projection to, 558
 - within query expressions, 593
- APIs (application programming interfaces)
 - encapsulation, 826–828
 - VirtualAllocEx(), declaring, 818–819
 - wrappers, simplifying calls with, 828–829
- APMs (Asynchronous Programming Models), 783–797
 - TPL (Task Parallel Library), calling, 791–796
- AppDomain, unhandled exceptions on, 744–746
- applicable calls, 185
- applications
 - domains, CLI (Common Language Infrastructure), 854–855
 - HelloWorld program, 2–4, 28–30
 - single instance, 766–767
 - task-related finalization
 - exceptions suppressed during shutdown, 717
- applying
 - arrays, 70–76
 - bitwise operators, 118
 - characters in arithmetic operations, 88–89
 - factory inheritance, 451
 - FlagsAttribute, 354–355
 - generic classes, 427–429
 - lambda expressions as data, 498–499
 - post-increment operators, 95
 - pre-increment operators, 96
 - SafeHandle, 823–824
 - strings, 50
 - StructLayoutAttribute for sequential layout, 820–821
 - System.Threading.Interlocked class, 761–763
 - validation to properties, 228–229
 - variables, 12–16
 - variance in delegates, 485
 - weak references, 391–393
- ArgumentNullException, 407
- arithmetic operators, 85
- arrays, 64–80
 - accessing, 70
 - anonymous types, initializing, 545–546
 - applying, 70–76
 - assigning, 66–70
 - common errors, 78–80
 - declaring, 65–66, 70
 - errors, 69
 - foreach loops, 546–547
 - instance methods, 75–76
 - instantiation, 66–70
 - length of, 72
 - methods, 73–75
 - parameters, 173–176
 - redimensioning, 75
 - strings as, 76–78
 - support for covariance and contravariance in, 462–463
- as operators, 302
- assemblies, 3
 - attributes, 665
 - CLI (Common Language Infrastructure), 855–858
 - metadata reflection, 652–662
 - multimode, building, 856n5
 - referencing, 377–381
 - targets, modifying, 378–379
- Assert() methods, 91
- assigning
 - arrays, 66–70
 - indexer property names, 632–633
 - null to strings, 51
 - pointers, 834–837
 - variables, 13, 14–16
- assignment operators, 92–98
 - binary operators, combining, 373
 - bitwise, 120
- associating
 - data in classes, 250
 - XML comments with programming constructs, 386–388
- associativity, order of, 86
- Asynchronous Delegate Invocation, 797–801
- asynchronous operations
 - with System.Threading.Thread, 738–740
- Asynchronous Programming Models, See APMs
- AsyncState property, 710
- ATL (Active Template Library), 278
- atomicity, 704–705, 752
- attributes, 663–688
 - assemblies, 665

- command-line, 881–888
- constructors, initializing, 668–673
- custom, 666–667
- FlagsAttribute, 354–355, 675
- limiting, 674
- metadata reflection, 652–662
- predefined, 676–677
- return, specifying, 666
- searching, 667–668
- serialization, 680–682
- System.ConditionalAttribute, 677–679
- System.NonSerializable, 682–683
- System.ObsoleteAttribute, 679–680
- System.Runtime.Serialization.OptionalFieldAttribute, 686
- ThreadStaticAttribute, 775–777
- automatically implemented properties, 225–227
- AutoResetEvent, semaphores over, 772
- availability of types, 380
- Average function, 585
- avoiding
 - ambiguity, 213–217
 - copying, 345
 - deadlock, 759, 764–765
 - equality conditionals, 91
 - string types, 759–760
 - synchronization, 760
 - this type, 759–760
 - typeof types, 759–760
 - unboxing, 345
 - unnecessary locking, 765–766

B

- BackgroundWorker class patterns, 804–809
- backslash (\), 42
- base classes, 204
 - constraints, 444–445
 - overriding, 281–293

- refactoring, 271
- Base Class Library. *See* BCL
- base interfaces, using in class declarations, 320
- base members, 291–292
- base types, casting between
 - derived types, 272–273
- BCL (Base Class Library), 25
 - CLI (Common Language Infrastructure), 860
- behaviors, dynamic data type, 690–693
- benefits of generics, 430–431
- best practices, synchronization design, 674
- binary operators, 85, 371–373
- BinarySearch() method, 75, 620
- Binary Tree and Pair, full source code listings, 876–881
- BinaryTree<T> class with no constraints, declaring, 439
- binding, dynamic, 694
- bits, 115
- bitwise operators, 115–121
 - assignment, 120
 - complement (~) operator, 120
- blocks
 - catch, general, 409–410
 - code blocks, 831
 - code blocks (), 105–107
 - System.Exception, 195–196
 - unchecked, 418
- Boolean expressions, 109–115
- Boolean types, 40–41
- boxing, 339–346
- break statements, 132–135
- BubbleSort() method, 470–472
- buffers
 - overflow bugs, 72
 - overruns, 72
- building
 - custom collections, 611–612

- multimode assemblies, 856n5
- bytes, 115

C

- C#
 - acronyms, 862–863
 - CLI (Common Language Infrastructure), compiling to machine code, 847–849
 - compilers
 - downloading, 865
 - installing, 865–867
 - custom collection interfaces, 612–613
 - delegate instantiation, 477–480
 - general catch blocks in, 409–410
 - LINQ, projection using query expressions, 592–593
 - overview of, 1–2
 - preprocessor directives, 138–145
 - properties, 48
 - syntax fundamentals, 4–12
 - VirtualAllocEx() APIs, declaring, 818–819
 - without generics, 422–439
- C++
 - array declaration, 66
 - buffer overflow bugs, 72
 - delete operator, 208
 - deterministic destruction, 399, 850
 - dispatch method calls
 - during construction, 286
 - global methods, 158
 - global variables and functions, 248
 - header files, 160
 - implicit overriding, 283
 - multiple inheritance, 278
 - operators, errors, 110
 - pointers, declaring, 833
 - preprocessing, 138
 - pure virtual functions, 297

- C++ (*contd.*)
 - struct defines type with
 - public members, 337
 - switch statements, 132
 - templates, 442
 - var, 540
 - Variant, 540
 - void*, 540
 - void as data types, 52
 - calculating
 - pi, 725
 - values, 115
 - callbacks, invoking, 787
 - caller variables, matching
 - parameter names, 168
 - calling
 - APMs (Asynchronous Programming Models), 784–786, 791–796
 - applicable, 185
 - binary operators, 372–373
 - call sites, 168
 - constructors, 237, 243–244
 - external functions, 826–828
 - methods, 150–156
 - object initializers, 240
 - SelectMany() method, 580–582
 - stacks, 168
 - allocating data on, 836
 - exceptions, 412
 - wrappers, simplifying
 - APIs with, 828–829
 - cancellation
 - cooperative, 719
 - parallel loops, 729–734
 - tasks, 718–722
 - CancellationToken-
 - Source.Token property, 731
 - CAS (Code Access Security), 659, 852
 - case sensitivity, 2
 - casting
 - between base and derived types, 272–273
 - within inheritance chains, 274
 - inside generic methods, 456–457
 - cast operator (()), 58, 375–376
 - catch blocks
 - general, 409–410
 - System.Exception, 195–196
 - catching exceptions, 191–192, 196, 407–408, 411
 - categories of types, 55–57, 332–339
 - CD-ROM drives, 274
 - Cell type, 427
 - centralizing initialization, 244–245
 - chains, casting within inheritance, 274
 - characters
 - arithmetic operations, applying, 88–89
 - escape, 42, 43
 - newline (\n), 42, 48
 - Unicode, 41–43
 - char types, 41
 - checked conversions, 59–61, 417–419
 - checking
 - for null, 513–514
 - types, 851
 - child classes, 205
 - child collections, creating, 578
 - CIL (Common Intermediate Language), 23
 - boxing code in, 340
 - CLI (Common Language Infrastructure), 858
 - dynamic data type, 693
 - and ILDASM, 27–30
 - out variable
 - implementation, 496–498
 - representation of generics, 463–464
 - System.SerializableAttribute, 687–688
- circular wait condition, 765
- class, iterators, 645
- classes, 201–202
 - abstract, inheritance, 293–299
 - access modifiers, 220–222, 380–381
 - associated data, 250
 - BackgroundWorker
 - patterns, 804–809
 - base, 204
 - constraints, 444–445
 - overriding, 281–293
 - refactoring, 271
 - BinaryTree<T>, declaring
 - with no constraints, 439
 - concrete, 293
 - concurrent collection, 773–774
 - concurrent from
 - Systems.Collections.Concurrent, 895–898
 - ConsoleListControl, 307
 - constructors, 236–247
 - declaring, 205–209
 - defining, 206
 - definitions, 7
 - deriving, 270
 - encapsulation, 258–260
 - exceptions, inheritance, 192
 - extension methods, 256–258
 - generics, 427–429, 661–662
 - hierarchies, 204, 473n1
 - inner, 262
 - instances
 - fields, 209–211
 - methods, 211–212
 - instantiating, 205–209
 - interfaces
 - compared with, 328–329
 - duplicating, 433–434
 - iterators, creating multiple
 - in, 648–649
 - libraries, 377–378, 378
 - LinkedList<T>, 629
 - List<T>, 617–621
 - members, 209
 - Monitor, synchronization, 754–758

- nested, 260–262, 265
- object-oriented
 - programming, 203–205
- partial, 262–267
- primary collections, 617–630
- properties, 222–236
- Queue<T>, 629
- sealed, 281
- SortedDictionary<TKey, TValue>, 626–628
- SortedList<T>, 626–628
- Stack, 422, 425
- Stack<T>, 628
- static, 255
- static members, 247–256
- System.Threading.Interlocked, 761–763
- System.Threading.WaitHandle, 768–769
- this keyword, 213–220
- clauses
 - into, query continuation with, 605–606
 - Let, 600–602
 - query expressions, 590
 - where, converting expression trees to, 499
- cleanup, resources, 790–791, 823–824
 - well-formed types, 393–400
- Clear() method, 75
- CLI (Common Language Infrastructure), 1, 24, 843–844
 - application domains, 854–855
 - assemblies, 855–858
 - BCL (Base Class Library), 860
 - C#, compiling to machine code, 847–849
 - CIL (Common Intermediate Language), 858
 - CLS (Common Language Specification), 859–860
 - CTS (Common Type System), 858–859
 - defining, 844–845
 - implementation, 845–846
 - manifests, 855–858
 - metadata, 860–861
 - modules, 855–858
 - P/Invoke, 816–830
 - runtime, 849–854
 - CLS (Common Language Specification), 24
 - CLI (Common Language Infrastructure), 859–860
 - CLU language, 635
 - clusters, 635
 - code
 - access security, 25
 - Binary Tree and Pair, 876–881
 - CAS (code access security), 659
 - CIL, boxing in, 340
 - command-line attributes, 881–888
 - comments, 20–23
 - conventions, events, 526–528
 - declaration space, 107–109
 - HelloWorld program, 2–4
 - invalid, indentation, 106
 - machine, 844, 847–849
 - management, 24
 - multithreading. *See* multithreading
 - paths, 159
 - P/Invoke, 816–830
 - ProductSerialNumber, 874–876
 - pseudocode, executing, 752
 - reusing, 378
 - scope, 107–109
 - styles, avoiding ambiguity, 213–217
 - Tic-Tac-Toe, 869–874
 - unsafe, 831–832
 - values, hardcoding, 35–37
 - virtual computer detection using P/Invoke, 888–894
 - whitespace, formatting, 11–12
 - Code Access Security (CAS), 659, 852
 - code blocks (), 105–107
 - collections
 - concurrent, 773–774
 - custom, building, 611–612
 - dictionary, 622–626
 - IComparable<T> interfaces, 614–617
 - IDictionary<TKey, TValue> interface, 614–617
 - IList<T> interface, 614–617
 - index operators, 630–634
 - initializers, 240–241, 543–546
 - interfaces, 612–613
 - anonymous types, 536–538
 - IEnumerable<T>, 546–552
 - implicitly typed local variables, 538–540
 - with standard query operators, 535–536
 - iterators, 634–650
 - linked lists, 629–630
 - null, returning, 634
 - primary collections classes, 617–630
 - queues, 629
 - sorting, 626–628
 - stacks, 628
 - Collect() method, 391
 - COM
 - controlling, 813
 - DLL registration, 858
 - combining binary operators and assignment operators, 373
 - command-line
 - arguments to Main()
 - methods, passing, 166
 - attributes, full source code listings, 881–888
 - options, 76

- CommandLineHandler.TryParse() method, 671
- comments, 20–23
 - delimited, 21
 - single-line, 22
 - XML, 385–389
- common errors, arrays, 78–80
- Common Intermediate Language. *See* CIL
- Common Language Infrastructure. *See* CLI
- Common Language Specification. *See* CLS
- CompareTo() method, 442
- ComparisonHandler-Compatible method, 478–479
- compatibility, types between enums, 349–350
- compilers
 - C#
 - downloading, 865
 - installing, 865–867
 - extracting XML data, 385n2
 - compiling
 - case sensitivity, 2
 - C# to machine code, 847–849
 - HelloWorld program, 3–4
 - JIT (just-in-time) compilers, 848
 - LINQ query expressions, 607
 - static compilation *versus* dynamic programming, 695–696
 - string concatenation, 45
- computers, virtual, 816
- concatenation of strings
 - compile time, 45
- Concat() standard query operator, 584
- concrete classes, 293
- concurrent classes from Systems.Collections.Concurrent, 895–898
- concurrent collection classes, 773–774
- conditional (?) operator, 113–114
- conditionals, 109. *See also* Boolean expressions
- conditions, removing, 765
- connecting
 - publishers, 511–512
 - subscribers, 511–512
- console executable, 378
- ConsoleListControl class, 307
- consoles, input and output, 16–20
- ConsoleSyncObject, 797
- constants
 - expressions, 98
 - mathematics, 107
- const fields, 258–259
- constraints
 - base classes, 444–445
 - constructors, 446–447, 451
 - generics, 439–457
 - inheritance, 447–448, 450
 - interfaces, 442–444
 - limitations, 449–452
 - multiple, 446
 - struct/class, 445
- constructors
 - attributes, initializing, 668–673
 - calling, 237, 243–244
 - classes, 236–247
 - constraints, 446–447, 451
 - declaring, 237–238
 - default, 239
 - defining, 434–435
 - inheritance, 292–293
 - overloading, 241–242
 - static, 253–254
- constructs
 - metadata reflection, 652–662
 - programming, associating XML comments with, 386–388
- contextual keywords, 6–7
- Continuation Passing Style. *See* CPS
- continue statements, 135–136
- ContinueWith() method, 711–715, 717, 795–796
- contravariance, generics, 457–463
- control flow, 83–84
 - statements, 121–132
- controlling
 - COM, 813
 - threads, 706–738
- conventions
 - code, events, 526–528
 - naming. *See* naming conventions
- conversion
 - as operators, 302
 - checked, 59–61, 417–419
 - C# to CIL, 847
 - customizing, 274
 - between data types, 58–64
 - between enums and strings, 348, 350–351
 - expression trees to SQL where clauses, 499
 - generics to type parameters, 457
 - implicit, 62, 273
 - interfaces between implementing classes and, 318
 - numbers to Booleans, 61
 - numeric conversion with TryParse() method, 198–199
 - operators, 375
 - guidelines for, 377
 - implementation, 376
 - strings, 63
 - unchecked, 59–61, 417–419
- cooperative cancellation, 719
- copying, avoiding, 345
- Copy() method, 257
- CopyTo() method, 617
- CountdownEvent, 772
- Count() function, 585
- counting elements with Count() method, 561
- Count property, 617
- covariance, 438
 - generics, 457–463

- IEnumerable<out T>, 485n2
- C pointers, declaring, 833
- CPS (Continuation Passing Style), 787–789
- CTS (Common Type System), 858–859
- Current Programming with Windows*, 801n1
- custom attributes, 666–667
- custom collections
 - building, 611–612
 - IComparable<T> interfaces, 614–617
 - IDictionary<TKey, TValue> interface, 614–617
 - ICollection<T> interface, 614–617
 - index operators, 630–634
 - interfaces, 612–613
 - iterators, 634–650
 - linked lists, 629–630
 - null, returning, 634
 - primary collections classes, 617–630
 - queues, 629
 - sorting, 626–628
 - stacks, 628
- custom dynamic object implementation, 696–699
- customizing
 - conversions, defining, 274
 - event implementation, 532–533
 - exceptions, defining, 414–419
 - LINQ, 585
 - serialization, 683–684
- D**
- data
 - allocating on call stacks, 836
 - to and from an alternate thread, passing, 799–801
 - fixing, 835
 - persistence, 217
 - retrieval from files, 218
- DataStore() method, 545
- data types, 13–14, 31–32, 40–57
 - arrays, 64–80
 - categories of, 55–57
 - conversions between, 58–64
 - delegates, 472–473
 - dynamic, principles and behaviors, 690–693
 - fundamental numeric types, 32–40
 - nullable modifiers, 57–58
 - null keyword, 51–52
 - parameters, 818–819
 - short, 33
 - strings, 43–51
 - System.Text.
 - StringBuilder, 51
 - void keyword, 52–55
- deadlock, 705–706, 760
 - avoiding, 759, 764–765
- decimal types, 34–35
- declaration space, 107–109
- declaring
 - abstract members, 297
 - arrays, 65–66, 70
 - BinaryTree<T> class with no constraints, 439
 - classes, 8, 205–209
 - constant fields, 258
 - constructors, 237–238
 - delegates
 - data types, 475
 - with method returns, 522
 - events, 525–526
 - external functions, 817
 - fields as volatile, 760–761
 - finalizers, 393
 - generics
 - classes, 430
 - delegate types, 529
 - interfaces, 432
 - multiple type parameters, 436
 - instance fields, 209–210
 - interfaces, constraints, 443–444
- jagged arrays, 71
- Main() method, 9–10
- methods, 157–161, 159–160
- parameters, 159
- pointers, 832–834
- properties, 223–225
- static constructors, 253–254
- static properties, 254
- two-dimensional arrays, 68
- Type alias, 164
- variables, 13, 14
 - applying anonymous methods, 481
 - of the Class Type, 206
- VirtualAllocEx() APIs, 818–819
- Win32 APIs, 818n1
- decorating properties, 663, 664
- decrement (-) operator, 94–97
- default constructors, 239
- default() operators, 68, 338, 435
- default values, specifying, 435–436
- deferred execution
 - with LINQ query expressions, 593–598
 - standard query operators, 562–566
- defining
 - abstract classes, 294
 - abstract members, 295
 - cast operators, 275, 375
 - classes, 7, 206
 - CLI (Common Language Infrastructure), 844–845
 - constructors, 434–435
 - custom conversions, 274
 - custom exceptions, 414–419
 - delegates, types, 474–475
 - enums, 347
 - finalizers, 393–395, 434–435
 - generic methods, 453
 - index operators, 631–632
 - inheritance, 269–270
 - interfaces, 307
 - iterators, 636

- defining (*contd.*)
 - namespaces, well-formed types, 382–385
 - nested classes, 260, 265
 - objects, 206
 - preprocessor symbols, 141
 - properties, 224
 - publishers, events, 510–511
 - simple generic classes, 429–430
 - specialized `Stack` classes, 425
 - `struct`, 334
 - subroutines, 53
 - subscriber methods, 508–510
 - types, 7–8
 - delegates
 - class hierarchies, 473n1
 - data types, 472–473
 - events, 528–530
 - instantiating, 475–480
 - internals, 473–474
 - invoking, 512–513
 - multicast, 508
 - coding observer patterns with, 508–523
 - internals, 518–519
 - operators, 514–516
 - overview of, 470–480
 - passing, 829
 - types, defining, 474–475
 - variance, applying, 485
 - delete operator, 208
 - deleting whitespace, 12
 - delimited comments, 21
 - XML, 387
 - delimiters, statements, 10
 - dereferencing
 - pointers, 837–839
 - reference types, 334
 - deriving
 - base types, casting
 - between, 272–273
 - inheritance, 270–281
 - one interface from another, 318
 - preventing, 281
 - design, synchronization best practices, 674
 - destruction, deterministic, 208, 399, 850
 - detecting virtual computers
 - using `P/Invoke`, 888–894
 - deterministic destruction, 208, 399, 850
 - deterministic finalization, 395–398
 - diagrams
 - interfaces, 325
 - sequences, 520
 - Venn, 568
 - dialog boxes, Windows Error Reporting, 715
 - dictionary collections, 622–626
 - directives
 - `import`, wildcards in, 162
 - preprocessor, C#, 138–145
 - using, 161–168
 - disambiguation, multiple `Main()` methods, 167
 - dispatch method calls during construction, 286
 - `Dispose()` method, 397
 - disposing tasks, 723–724
 - distinct members, 606–607
 - `Distinct()` standard query operator, 584
 - dividing `float` by zero, 91
 - division (`/`) operator, 85
 - documentation, generating
 - XML, 388–389
 - domains, applications, 854–855
 - `double` type, 36
 - `do/while` loops, 121–123
 - downloading C# compilers, 865
 - Duffy, Joe, 801n1
 - duplicating interfaces, 433–434
 - dynamic binding, 694
 - dynamic data type principles and behaviors, 690–693
 - dynamic objects
 - custom implementation, 696–699
 - programming with, 688–699
 - reflection, invoking, 689–690
 - dynamic programming, static compilation *versus*, 695–696
- ## E
- EAPs (Event-based Asynchronous Patterns), 801–804
 - editors, visual hints for, 144–145
 - `Eject()` method, 274
 - emitting errors, 141–142
 - empty catch block internals, 411
 - empty collections, returning, 634
 - enabling Intellisense, 592
 - encapsulation, 203
 - APIs, 826–828
 - circumventing, 852
 - classes, 258–260
 - information hiding, 220
 - objects group data with methods, 208–209
 - publication, 524–535
 - subscriptions, 523–524
 - of types, 379–380
 - enums
 - defining, 347
 - flags, 351–355
 - `FlagsAttribute`, 354–355
 - string conversion, 350–351
 - type compatibility
 - between, 349–350
 - value types, 346–355
 - equality conditionals, avoiding, 91
 - equality (`==`) operators, 110–111, 370
 - `Equals()` method, overriding, 361–369
 - errors
 - arrays, 69, 78–80
 - emitting, 141–142
 - handling
 - C# 3.0, 519–520
 - `P/Invoke`, 821–823
 - infinite recursion, 178

- methods, 186–199
- operators, 110
- reporting, 196
- trapping, 187–192
- escape sequences, 42
- Event-based Asynchronous Patterns. *See* EAPs
- events, 507–508
 - code conventions, 526–528
 - declaring, 525–526
 - delegates, 528–530
 - generics, 528–530
 - implementation,
 - customizing, 532–533
 - internals, 530–523
 - multicast delegates, coding
 - observer patterns with, 508–523
 - notifications
 - firing, 527–528
 - with multiple threads, 763–764
 - overview of, 523–533
 - publishers, defining, 510–511
 - reset, 768–771
- exceptions
 - catching, 191–192, 196, 407–408, 411
 - class inheritance, 192
 - customizing, defining, 414–419
 - error handling, 186–199
 - general catch blocks, 409–410
 - handling, 405–419
 - background worker patterns, 808–809
 - subscribers, 520
 - unhandled exception handling on Task, 715–718
 - hiding, 411–412
 - inner, 415
 - multiple types, 405–407
 - reports, 412
 - rethrowing, 197, 413
 - serializable, 416
 - throwing, 406–407
 - types, 193–194
 - unhandled exceptions on AppDomain, 744–746
 - exclusive OR (^) operator, 112
 - executing
 - deferred
 - with LINQ query expressions, 593–598
 - standard query operators, 562–566
 - implicit execution,
 - implementing, 607–608
 - iterations in Parallel, 724–734
 - management, 23–30
 - ManualResetEvent
 - synchronization, 770
 - pseudocode, 752
 - threads, 704. *See also* multithreading
 - time, 24
 - VES (Virtual Execution System), 844
 - explicit cast, 58–59
 - explicit member implementation, 314–315
 - exponential notation, 37
 - exposing Async methods, 810
 - expressions. *See also* LINQ
 - Boolean, 109–115
 - constants, 98
 - lambda, 401, 486–505
 - queries
 - LINQ, 589–590
 - PLINQ (Parallel LINQ), 736
 - trees, 498–505
 - converting to SQL where clauses, 499
 - object graphs, 499–501
 - viewing, 503–505
 - typeof, 654–655
 - Extensible Markup Language. *See* XML
 - extensions
 - interfaces, 322–323
 - IQueryable<T>, 585
 - methods, 256–258, 278
 - external functions, calling, 826–828
 - extracting XML data, 385n2

F

 - factory inheritance, 451
 - false operator, 373–375
 - FCL (Framework Class Library), 860
 - fields
 - const, 258–259
 - instances, 209–211, 249
 - static, 248–250
 - virtual, properties as, 232–234
 - volatile, declaring as, 760–761
 - filenames, must match class names (Java), 4
 - files
 - data retrieval, 218
 - header, 160
 - loading, 216
 - XML, 22–23, 388–389. *See also* XML
 - filtering
 - LINQ query expressions, 598–599
 - with
 - System.Linq.Enumerable.Where(), 562
 - with Where() methods, 556–557
 - finalization
 - deterministic, 395–398
 - garbage collection and, 398–399
 - guidelines, 400
 - task-related, 717
 - finalizers, 241, 393–395
 - defining, 434–435
 - FindAll() method, 621–622
 - firing event notifications, 527–528
 - fixing data, 835
 - flags, enums, 351–355
 - FlagsAttribute, 354–355, 675

- floating-point types, 33–34
 - inequality with, 89–92
 - special characteristics of, 89
 - flow. *See* control flow
 - foreach loops
 - with `IEnumerable<T>`, 547–551
 - without `IEnumerable<T>`, 551–552
 - foreach loops, 127–130
 - with arrays, 546–547
 - collections, iterating over, 613
 - modifying, 552
 - parallel execution of, 727
 - for loops, 124–127
 - format items, 19
 - `Format()` method, 46
 - formatting
 - code, avoiding ambiguity, 213–217
 - indentation, 12
 - Java
 - lowercase, 9
 - uppercase, 9
 - numbers as hexadecimal, 38–39
 - PLINQ (Parallel LINQ), 736–738
 - round-trip, 39–40
 - single instance
 - applications, 766–767
 - whitespace, 11–12
 - Forms, Windows, 809–811
 - Framework Class Library (FCL), 860
 - f-reachable objects, 390
 - from clause, 590
 - full outer joins, 569
 - full source code listings
 - `Binary Tree and Pair`, 876–881
 - command-line attributes, 881–888
 - `ProductSerialNumber`, 874–876
 - `Tic-Tac-Toe`, 869–874
 - virtual computer detection using `P/Invoke`, 888–894
 - functions
 - anonymous, 486
 - `Average`, 585
 - `Count()`, 585
 - external
 - calling, 826–828
 - declaring, 817
 - global variables and, 248
 - `Max()`, 585
 - `Min()`, 585
 - pointers, passing delegates, 829
 - pure virtual, 297
 - `Sum()`, 585
 - fundamental numeric types, 32–40
- G**
- garbage collection, 25, 849–851
 - and finalization, 398–399
 - well-formed types, 390–393
 - general catch blocks, 409–410
 - generating
 - anonymous types, 542–543
 - XML documentation files, 388–389
 - generics, 421
 - benefits of, 430–431
 - catch, 194
 - classes, 427–429
 - collection interface
 - hierarchies, 613
 - constraints, 439–457
 - contravariance, 457–463
 - covariance, 457–463
 - C# without, 422–439
 - events, 528–530
 - interfaces, 432–433
 - internals, 463–467
 - lazy loading and, 401
 - methods, 453–457
 - structs, 432–433
 - types, 427–439
 - nested, 438–439
 - reflection, 660–662
 - `Tuple`, 437–438
 - `GetHashCode()` method, overriding, 358–361
 - `GetSummary()` member, 296
 - getters, access modifiers, 231–232
 - `GetType()` member, 653–654
 - `GhostDoc`, 389n3
 - global variables and functions, 248
 - `goto` statements, 137–138
 - graphs, objects, 499–501
 - greater than (>) operator, 110
 - greater than or equal to (>=) operator, 110
 - `groupby` clause, 590
 - `GroupBy()` method, grouping results with, 575–577
 - grouping
 - LINQ query expressions, 602–605
 - results with `GroupBy()` method, 575–577
 - statements into methods, 150
 - `GroupJoin()` method, 577–580
 - guidelines
 - for conversion operators, 377
 - for exception handling, 411–413
 - finalization, 400
 - `P/Invoke`, 829–830
- H**
- handling
 - errors
 - C# 3.0, 519–520
 - methods, 186–199
 - `P/Invoke`, 821–823
 - exceptions, 405–419
 - background worker patterns, 808–809
 - subscribers, 520
 - unhandled exception handling on `Task`, 715–718
 - hardcoding values, 35–37
 - hash symbol (#), 139
 - header files, 160

- heaps, reference types, 333
 - HelloWorld program, 2–4
 - CIL output for, 28–30
 - hexadecimal notation, 38
 - hiding
 - exceptions, 411–412
 - information, 220
 - hierarchies
 - classes, 204, 473n1
 - collections, 613
 - hints for visual editors, 144–145
 - hold and wait condition, 764
 - hooking up background worker patterns, 807–808
- I**
- ICollection<T> interface, 616–617
 - IComparable<T> interface, 443, 614–617
 - IComparer<T> interface, sorting, 614–615
 - identifiers, 6–7
 - keywords used as, 7
 - type parameters, 429
 - IDictionary<TKey, TValue> interface, 614–617
 - IDisposable interface, using explicitly in place of SafeHandle, 825–826
 - Id property, 710
 - IEnumerable<T>
 - collections interfaces, 546–552
 - foreach loops with, 547–551
 - foreach loops without, 551–552
 - IEnumerable<out T>, covariance, 485n2
 - if statements, 102–103
 - followed by code blocks (), 105
 - ILDASM, CIL and, 27–30
 - ILList<T> interface, 614–617
 - immutable anonymous types, 541
 - immutable strings, 16, 49–51
 - immutable value types, 336
 - implementing
 - CLI (Common Language Infrastructure), 845–846
 - conversion operators, 376
 - custom dynamic objects, 696–699
 - Equals() method, 366
 - events, customizing, 532–533
 - explicit member, 314–315
 - generic interfaces, 432
 - GetHashCode() method, 359
 - implicit execution, 607–608
 - implicit member, 315–316
 - interfaces, 308–312, 312–318, 433–434
 - multiple interface inheritance, 324–326
 - new operator, 238
 - one-to-many relationships, 577–580
 - outer joins, 579
 - virtual methods, 283
 - implicit base type casting, 273
 - implicit conversion, 62, 273
 - cast operators, 376
 - implicit execution, implementing, 607–608
 - implicitly typed local variables, 53–55, 538–540
 - implicit member implementation, 315–316
 - implicit overriding, 283
 - import directive, wildcards in, 162
 - incompatibilities, 6n6
 - increment (++) operator, 94–97
 - indentation
 - formatting, 12
 - invalid code, 106
 - indexer property names, assigning, 632–633
 - index operators, 630–634
 - items to Dictionary<TKey, TValue>, adding, 623
 - indiscriminate synchronization, 758
 - inequality (!=) operator, 110, 370
 - inequality with floating-point types, 89–92
 - inferencing types, 454–455
 - infinite recursion errors, 178
 - infinity, negative, 92
 - information hiding, 220
 - infrastructure, languages, 23–30. *See also* CLI
 - inheritance, 203, 269–270
 - abstract classes, 293–299
 - as operators, 302
 - base classes, overriding, 281–293
 - chains, casting within, 274
 - constraints, 447–448, 450
 - definitions, 269–270
 - derivation, 270–281
 - exceptions, classes, 192
 - factory, 451
 - interfaces, 318–321
 - is operators, 301
 - methods, 271
 - multiple, 278
 - multiple interfaces, 321–322, 324–326
 - polymorphism, 297–299
 - single, 278–281
 - System.Object, 299–301
 - types, 205
 - value types, 338–339
 - initializers
 - collection, 240–241, 543–546
 - objects, 239–241
 - initializing
 - anonymous type arrays, 545–546
 - attributes through constructors, 668–673
 - centralizing, 244–245
 - jagged arrays, 70
 - lazy initialization, well-formed types, 400–402
 - structs, 336–337

- initializing (*contd.*)
 - three-dimensional arrays, 69
 - two-dimensional arrays, 69
 - inner classes, 262
 - inner exceptions, 415
 - inner joins, 568
 - with `Join()` method, performing, 572–575
 - input, consoles, 16–20
 - installing C# compilers, 865–867
 - instances
 - array methods, 75–76
 - custom attributes, retrieving, 670
 - fields, 209–211, 249
 - methods, 47, 211–212
 - single applications, 766–767
 - instantiating, 9
 - arrays, 66–70
 - classes, 205–209
 - delegates, 475–480
 - generics
 - based on reference types, 465–467
 - based on value types, 464–465
 - integers
 - types, 32–33
 - values, overflowing, 59
 - Intellisense, enabling, 592
 - interfaces, 305–307
 - collection, 535–536. *See also* collection interfaces
 - compared with classes, 328–329
 - constraints, 442–444
 - conversion between implementing classes and, 318
 - custom collections, 612–613
 - defining, 307
 - diagramming, 325
 - duplicating, 433–434
 - explicit member implementation, 314–315
 - extension methods on, 322–323
 - generics, 432–433
 - `ICollection<T>`, 616–617
 - `IComparable<T>`, 443, 614–617
 - `IComparer<T>`, 614–615
 - `IDictionary<TKey, TValue>`, 614–617
 - `IDisposable`, using explicitly in place of `SafeHandle`, 825–826
 - `IList<T>`, 614–617
 - implementation, 312–318
 - implicit member implementation, 315–316
 - inheritance, 318–321
 - multiple inheritance, 321–322, 324–326
 - `Parallel.For()` API, 726
 - polymorphism through, 307–312
 - support, 440
 - value types, 338–339
 - versioning, 327–328
 - `VirtualAllocEx()`, declaring, 818–819
 - Windows UI programming, 809–813
 - internals
 - anonymous methods, 494–495
 - delegates, 473–474
 - events, 530–523
 - generics, 463–467
 - lambda expressions, 494–495
 - multicast delegates, 518–519
 - properties, 235–236
 - interoperability of languages, 25
 - `Intersect()` standard query operator, 584
 - into clauses, query continuation with, 605–606
 - in type parameter, enabling contravariance with, 460–462
 - invalid code, indenting, 106
 - invalid reference types, 833
 - invoking
 - callbacks, 787
 - delegates, 512–513, 522
 - members, 655–660
 - `P/Invoke` (Platform Invoke), 816–830
 - reflection, dynamic objects, 689–690
 - sequential invocation, 516–517
 - using statements, 397
 - `IQueryable<T>`, 585
 - `IsCompleted` property, 710
 - is operators, 301
 - items, formatting, 19
 - iterations
 - `Dictionary<Tkey, TValue>`, 624
 - executing in `Parallel`, 724–734
 - foreach loops, modifying, 552
 - over foreach loops, 613
 - iterators
 - class, 645
 - classes, creating multiple in, 648–649
 - collections, 634–650
 - defining, 636
 - examples of, 641–643
 - overview of, 646–648
 - and state, 639–641
 - struct, 645
 - syntax, 636–637
 - values, yielding, 637–639
 - yield break, 645–646
 - yield statements, 649
- ## J
- jagged arrays. *See also* arrays
 - declaring, 71
 - initializing, 70
 - Java
 - array declaration, 66
 - exception specifiers, 408
 - filenames must match class names, 4
 - generics, 467
 - implicit overriding, 283
 - inner classes, 262

- virtual methods by default, 282
- wildcards in `import` directive, 162
- JavaScript
 - `var`, 540
 - `Variant`, 540
 - `void*`, 540
- JIT (just-in-time) compilers, 848
- jitting, 24
- `Join()` method, performing
 - with inner joins, 572–575
- joins, 568, 569
- jump statements, 132–138
- just-in-time (JIT) compilers, 848

K

- keywords, 4–6
 - contextual, 6–7
 - lock, 757–758
 - `Me`, accessing class instances with, 214
 - new, 67
 - `null`, 51–52
 - string, 163n2
 - `this`, classes, 213–220
 - used as identifiers, 7
 - `var`, 53
 - `void`, 52–55
 - `yield`, 6n5
- Knoppix, 867

L

- lambdas
 - expressions, 401, 486–505
 - statements, 486–489
- languages, 158
 - accessing class instances with `Me` keyword, 214
 - buffer overflow bugs, 72
 - CIL (Common Intermediate Language), 23
 - COM DLL registration, 858
 - `delete` operator, 208

- deterministic destruction, 399, 850
- dispatch method calls
 - during construction, 286
- exception specifiers, 408
- generics, 467
- global variables and functions, 248
- header files, 160
- implicit overriding, 283
- infrastructure, 23–30
- inner classes, 262
- interoperability, 25
- Java
 - filename must match class names, 4
 - `main()` is all lowercase, 9
 - multiple inheritance, 278
 - operator errors, 110
 - origin of iterators, 635
 - preprocessing, 138
 - project scope `Imports` directive, 162
 - pure virtual functions, 297
 - redimensioning arrays, 75
 - returning `void`, 53
 - short data types, 33
 - string concatenation at compile time, 45
 - `struct` defines type with public members, 337
 - templates, 442
 - UML (Unified Modeling Language), 325n1
 - virtual methods by default, 282
 - Visual Basic line-based statements, 10
 - `void*`, 540
 - `void` as data types, 52
 - wildcards in `import` directive, 162
- last in, first out (LIFO), 422
- lazy initialization, well-formed types, 400–402
- left outer joins, 568
- length
 - of arrays, 72
 - strings, 48–49

- less than (<) operator, 110
- less than or equal to (<=) operator, 110
- `Let` clause, 600–602
- libraries
 - class, 378
 - classes, 377–378
- LIFO (last in, first out), 422
- limiting
 - attributes, 674
 - constraints, 449–452
- line-based statements, 10
- lines, specifying numbers, 143–144
- linked lists, collections, 629–630
- `LinkedList<T>` class, 629
- LINQ
 - customizing, 585
 - distinct members, 606–607
 - implicit execution, implementing, 607–608
 - `Let` clause, 600–602
 - queries
 - continuation with `into` clauses, 605–606
 - running in parallel, 734–738
 - query expressions, 589–590
 - compiling, 607
 - deferred execution with, 593–598
 - filtering, 598–599
 - grouping, 602–605
 - as method invocations, 608–609
 - overview of, 590–592
 - projection using, 592–593
 - sorting, 599–600
- Linux, 867
- Liskov, Barbara, 635
- `List<T>` class, 617–621
- literals
 - strings, 44–46
 - values, 35, 68
- loading files, 216
- local storage, threads, 774–777

- local variables, 13
 - implicitly typed, 53–55, 538–540
 - multiple threads, 753–753
 - lock keyword, 757–758
 - ConsoleSyncObject, 797
 - objects, selecting, 758–759
 - locks, avoiding unnecessary, 765–766
 - lock statements, value types
 - in, 343
 - logical Boolean operators, 111–113
 - logical notation (!) operator, 113
 - logical operators, 117–118
 - logs, exceptions, 412
 - long-running threads, 722–723
 - loops
 - for, 124–127
 - decrement (- -) operators, 94
 - do/while, 121–123
 - foreach, 127–130
 - with arrays, 546–547
 - with IEnumerable<T>, 547–551
 - iterating over, 613
 - modifying, 552
 - parallel execution of, 727
 - without
 - IEnumerable<T>, 551–552
 - parallel, canceling, 729–734
 - while, 121–123
 - yield returns, placing in, 643–645
 - lowercase, Java, 9
- M**
- machine code, 844, 847–849
 - Main() method, 8
 - declarations, 9–10
 - parameters, 165–168
 - returns, 165–168
 - managing
 - code, 24
 - execution, 23–30
 - resources, 823–824
 - threads, 740–742
 - manifests, CLI (Common Language Infrastructure), 855–858
 - ManualResetEvent, 768–771
 - ManualResetEventSlim, 768–771
 - many-to-many relationships, 569
 - matching caller variables
 - with parameter names, 168
 - mathematics constants, 107
 - Max() function, 585
 - Me keyword, accessing class instances with, 214
 - members
 - abstract, 294
 - base, 291–292
 - classes, 209
 - distinct, 606–607
 - explicit member
 - implementation, 314–315
 - GetSummary(), 296
 - GetType(), 653–654
 - implicit member
 - implementation, 315–316
 - invoking, 655–660
 - object, overriding, 357–369
 - private, 220
 - referent types, accessing, 839
 - static, 247–256
 - System.Object, 299–301
 - variables, 209
 - messages, turning off warning (#pragma), 142–143
 - metadata, 25
 - CLI (Common Language Infrastructure), 860–861
 - reflection, 652–662
 - methodImpAttribute,
 - avoiding
 - synchronization, 760
 - methods, 149–150
 - Add(), 543
 - anonymous, 480–482
 - internals, 494–495
 - parameterless, 482
 - arrays, 73–75
 - Assert(), 91
 - BinarySearch(), 75, 620
 - BubbleSort(), 470–472
 - calling, 150–156
 - Clear(), 75
 - Collect(), 391
 - CommandLineHandler.TryParse(), 671
 - CompareTo(), 442
 - ComparisonHandler-
 - Compatible, 478–479
 - ContinueWith(), 711–715, 717, 795–796
 - Copy(), 257
 - CopyTo(), 617
 - Count(), counting
 - elements with, 561
 - DataStore(), 545
 - declaring, 157–161
 - Dispose(), 397
 - Eject(), 274
 - Equals(), overriding, 361–369
 - error handling, 186–199
 - extension, 256–258
 - extensions, 278
 - FindAll(), 621–622
 - Format(), 46
 - generics, 453–457
 - casting inside, 456–457
 - determining support for, 661–662
 - GetHashCode(),
 - overriding, 358–361
 - GroupBy(), grouping
 - results with, 575–577
 - GroupJoin(), 577–580
 - inheritance, 271
 - instances, 47, 75–76, 211–212
 - Join(), performing inner joins with, 572–575
 - Main(), 8. *See also* Main() method
 - declarations, 9–10

- multiple `Main()`,
 disambiguation, 167
 - optional parameters,
 182–185
 - `OrderBy()`, sorting with,
 566–572
 - overloading, 179–182
 - overview of, 150–152
 - parameters, 168–176
 - partial, 264–267
 - `Pop()`, 422
 - `Pulse()`, 756
 - `Push()`, 422
 - query expressions as
 invocations, 608–609
 - recursion, 176–179
 - refactoring into, 158
 - resolution, 185
 - returns, 155–156, 522–523
 - `Run()`, 285
 - `Select()`, 557–560, 734
 - `SelectMany()`, 580–582
 - `SetName()`, 213
 - starting, 707
 - static, 251–253
 - `Store()`, 216
 - strings, 46–47
 - `stringStatic`, 46
 - subscriber, defining,
 508–510
 - `System.Console.ReadKey()`
), 18
 - `System.Console.ReadLine()`
), 16
 - `System.Console.Write()`,
 18–20
 - `System.Threading.Interlocked`, 762
 - `ThenBy()`, sorting with,
 566–572
 - `ToString()`, overriding,
 358
 - `ToUpper()`, 50
 - `TryParse()`, 63, 198–199
 - type names, 154
 - for unsafe code, 831
 - `Where()`, filtering with,
 556–557
 - `Min()` function, 585
 - minus (-) operator, 84–92
 - models, APMs (Asynchronous Programming Models), 783–797
 - modifiers
 access, 220–222, 852
 - accessibility, 381
 - new, 286–291
 - nullable, 57–58
 - readonly, 259
 - sealed, 291
 - virtual, 282–286
 - modifying
 foreach loops, 552
 - targets, assemblies,
 378–379
 - values, variables, 15
 - modules, 378
 - CLI (Common Language Infrastructure),
 855–858
 - `Monitor` class synchronization, 754–758
 - Mono compilers, 3n4,
 866–867
 - MTA (Multithreaded Apartment), 813
 - multicast delegates, 508
 - coding observer patterns
 with, 508–523
 - internals, 518–519
 - multidimensional array
 errors, 69
 - multimode assemblies,
 building, 856n5
 - multiple constraints, 446
 - multiple duplication of inter-
 faces, 433–434
 - multiple exception types,
 405–407
 - multiple inheritance, 278
 - multiple interface inheritance,
 321–322
 - implementing, 324–326
 - multiple iterators, creating,
 648–649
 - multiple `Main()` methods,
 disambiguation, 167
 - multiple threads
 event notification with,
 763–764
 - and local variables,
 753–753
 - thread-safe, 752
 - multiple type parameters,
 436
 - multiplication (*) operator,
 85
 - Multithreaded Apartment
 (MTA), 813
 - multithreading, 701–706
 - before .NET Framework 4,
 738–743
 - uncertainty, 706
 - unhandled exceptions on
 AppDomain, 744–746
 - mutual exclusion condition,
 764
- ## N
- names
 index property,
 assigning, 632–633
 - parameters, 184, 674–676
 - type methods, 154
 - namespaces, 152–154, 161
 - aliasing, 164–165
 - alias qualifiers, 384–385
 - nesting, 383
 - well-formed types,
 defining, 382–385
 - naming conventions
 parameter types, 431
 - properties, 228–229
 - types, 7
 - NDoc, 389n4
 - negative infinity, 92
 - nesting
 classes, 260–262, 265
 - delegate data types,
 declaring, 475
 - generic types, 438–439
 - if statements, 103–105
 - namespaces, 383
 - using declaratives, 163
 - .NET, 865–866
 - Framework,
 multithreading before
 version 4, 738–743
 - garbage collection, 849–850

- .NET (*contd.*)
 - garbage collection in, 390–391
 - lazy initialization, 401
 - versioning, 26–27
 - new keyword, 67
 - newline (\n) characters, 42, 48
 - new modifiers, 286–291
 - new operator
 - implementation, 238
 - value types, 337
 - NGEN tool, 848
 - no preemption condition, 764
 - notation
 - exponential, 37
 - hexidecimal, 38
 - notifications, events
 - firing, 527–528
 - with multiple threads, 763–764
 - Novell, 3n4
 - nowarn:<warn list> option, 143
 - null
 - checking for, 513–514
 - returning, 634
 - nullable modifiers, 57–58
 - nullable value types, 425–427
 - null coalescing (??) operator, 114–115
 - null keyword, 51–52
 - numbers
 - to Booleans, conversion, 61
 - conversion with
 - TryParse() method, 198–199
 - hexidecimal, formatting, 38–39
 - lines, specifying, 143–144
 - types, 32–40
- O**
- object members, overriding, 357–369
 - object-oriented programming, classes, 203–205
 - objects
 - associated data, 250
 - CTS (Common Type System), 859
 - defining, 206
 - dynamic
 - implementing custom, 696–699
 - invoking reflection, 689–690
 - programming with, 688–699
 - f-reachable, 390
 - graphs, 499–501
 - group data with methods, 208–209
 - initializers, 239–241
 - lock, selecting, 758–759
 - resurrecting, 399–400
 - observers, 508
 - patterns, coding multicast delegates with, 508–523
 - OfType<T> () standard query operator, 584
 - omitting parameter types
 - from statement lambdas, 488
 - one-to-many relationships, 569
 - implementing, 577–580
 - operands, 84, 92
 - operators, 83–84, 84–98
 - AND (&&), 112, 373
 - adding, 371–373
 - addition (+), 85, 371–373
 - arithmetic, 85
 - as, 302
 - assignment, 92–98, 120
 - binary, 371–373
 - bitwise, 115–121
 - bitwise complement (~), 120
 - cast, 58, 275
 - cast (), 375–376
 - conditional (?), 113–114
 - constraints, 449
 - conversion, 375, 377
 - decrement (- -), 94–97
 - default(), 68, 338, 435
 - delegates, 514–516
 - delete, 208
 - division (/), 85
 - equality (==), 110–111, 370
 - errors, 110
 - exclusive OR (^), 112
 - false, 373–375
 - greater than (>), 110
 - greater than or equal to (>=), 110
 - increment (++), 94–97
 - index, 623, 630–634
 - inequality (!=), 110, 370
 - is, 301
 - less than (<), 110
 - less than or equal to (<=), 110
 - logical, 117–118
 - logical Boolean, 111–113
 - logical notation (!), 113
 - minus (-), 84–92
 - multiplication (*), 85
 - new
 - implementation, 238
 - value types, 337
 - null coalescing (??), 114–115
 - OR (| |), 111–112, 373, 450
 - overloading, 369–377
 - parenthesis, 92–98
 - plus (+), 84–92
 - postfix increment, 96
 - post-increment, 95
 - precedence, 86
 - prefix increment, 96
 - pre-increment, 96
 - remainder (%), 85
 - shift, 116–117
 - simple assignment (=), 14
 - standard query, 535–536.
 - See also collection interfaces; standard query operators
 - true, 373–375
 - unary, 373–375
 - optional parameters, 182–185
 - options
 - command-line, 76
 - nowarn:<warn list>, 143
 - parallel, 731–734

`OrderBy()` method, sorting
 with `ThenBy()` method,
 566–572

order of associativity, 86

origin of iterators, 635

OR (`|`) operator, 111–112,
 373

- constraints, 450
- outer joins, 568
- implementing, 579
- outer variables, 495–496
- out parameter values,
 234–235
- output
 - consoles, 16–20
 - parameters, 171–173
- out type parameter, enabling
 covariance with,
 458–460
- overflowing
 - bounds of a `float`, 92
 - integer values, 59
- overloading
 - constructors, 241–242
 - methods, 179–182
 - operators, 369–377
 - `System.Threading.Interl`
`ocked` class methods,
 762
- overriding
 - base classes, 281–293
 - `Equals()` method, 361–369
 - `GetHashCode()` method,
 358–361
 - implicit, 283
 - object members, 357–369
 - properties, 282
 - `ToString()` method, 358
- overruns, buffer, 72

P

parallel

- exception handling with
`System.AggregateExc`
`ption`, 728–729
- iterations, executing in,
 724–734
- loops, canceling, 729–734
- results and options,
 731–734

`Parallel.For()` API, 726

Parallel LINQ (PLINQ),
 559–560, 703, 736–738

parameterized types, 427

parameterless anonymous
 methods, 482

parameterless statement
 lambdas, 488

parameters, 149–150

- `AllowMultiple`, 674
- arrays, 173–176
- data types, 818–819
- declaring, 159
- `Main()` method, 165–168
- methods, 155, 168–176
- named, 184, 674–676
- optional, 182–185
- output, 171–173
- references, 170–171
- single input, statement
 lambdas with, 489
- types, 429, 660–661
- `in`, 460–462
- inferring, 454–455
- multiple, 436
- naming conventions, 431
- `out`, 458–460
- values, 168–169
- variables, defining index
 operators, 633–634

parent classes, 205

parenthesis operator, 92–98

partial classes, 262–267

partial methods, 264–267

pass-by references, 522

passing

- anonymous methods,
 480–481
- command-line arguments
 to `Main()` methods,
 166
- data to and from an
 alternate thread,
 799–801
- delegates, 486–487,
 489–490, 829
- states between APM
 (Synchronous
 Programming Model)
 methods, 789–790

paths, code, 159

patterns

- `BackgroundWorker` class,
 804–809
- EAPs (Event-based
 Asynchronous
 Patterns), 801–804
- observers, coding multicast
 delegates with,
 508–523
- publish-subscribe, 508

performance, 853–854

- synchronization, affect on,
 758

performing inner joins with
`Join()` method,
 572–575

permanent values, 259

permissions, CAS (code
 access security), 659

persistence, data, 217

pi, calculating, 725

P/Invoke (Platform Invoke),
 816–830

- errors, handling, 821–823
- virtual computer detection
 using, 888–894

placeholders, 19

- values, 115

Platform Invoke. *See* P/
 Invoke

platform portability, 852–853

platforms, 865–867

- portability, 25

PLINQ (Parallel LINQ),
 559–560, 703, 736–738

plus (+) operator, 84–92

pointers

- and addresses, 830–839
- assigning, 834–837
- declaring, 832–834
- dereferencing, 837–839
- functions, passing
 delegates, 829

polymorphism, 205

- inheritance, 297–299
- through interfaces,
 307–312

pools, threads, 706, 742–743

`Pop()` method, 422

portability
 platform, 852–853
 platforms, 25
 postfix increment operators, 96
 post-increment operators, applying, 95
 precedence, operators, 86
 predefined attributes, 676–677
 predefined types, 31
 prefix increment operators, 96
 pre-increment operators, applying, 96
 preprocessor directives, C#, 138–145
 preventing
 covariance maintains homogeneity, 457
 derivation, 281
 primary collections classes, 617–630
 primitives, 31
 principles, dynamic data type, 690–693
 private access modifiers, 275
 private members, 220
 ProductSerialNumber, 874–876
 programming
 APMs (Asynchronous Programming Models), 783–797
 Binary Tree and Pair, 876–881
 command-line attributes, 881–888
 comments, 20–23
 constructs, associating XML comments with, 386–388
 dynamic, static compilation *versus*, 695–696
 with dynamic objects, 688–699
 HelloWorld program, 2–4

object-oriented, classes, 203–205
 ProductSerialNumber, 874–876
 Tic-Tac-Toe, 869–874
 values, hardcoding, 35–37
 virtual computer detection using P/Invoke, 888–894
 Windows UI, 809–813
 programs
 CIL output for, 28–30
 HelloWorld, 2–4
 projecting
 LINQ query expressions, 592–593
 with Select() method, 557–560
 project scope Imports directive, 162
 properties
 attributes, 663, 664
 automatically implemented, 225–227
 C#, 48
 classes, 222–236
 Count, 617
 declaring, 223–225
 defining, 224
 indexer property names, assigning, 632–633
 internals, 235–236
 lazy loading, 402
 naming conventions, 228–229
 overriding, 282
 read-only, 230–231
 static, 254–256
 validation, applying, 228–229
 as virtual fields, 232–234
 write-only, 230–231
 protected access modifiers, 276
 pseudocode, executing, 752
 publication, encapsulating, 524–535
 public constants, 259
 publishers
 connecting, 511–512

events
 defining, 510–511
 publish-subscribe patterns, 508
 Pulse() method, 756
 pure virtual functions, 297
 Push() method, 422

Q

qualifiers, aliasing namespaces, 384–385
 quantum, 704
 queries. *See also* LINQ
 continuation with into clauses, 605–606
 LINQ, 589–590, 734–738
 PLINQ (Parallel LINQ), 559–560, 736–738
 standard query operators. *See* standard query operators
 queues, collections, 629
 Queue<T> class, 629

R

RCW (runtime callable wrapper), 813
 readonly modifiers, 259
 read-only properties, 230–231
 recursion
 infinite recursion errors, 178
 methods, 176–179
 redimensioning arrays, 75
 reentrant (locks), 765
 refactoring
 base classes, 271
 into methods, 158
 references
 assemblies, 377–381
 parameters, 170–171
 pass-by, 522
 root, 390
 strong, 391
 types, 56–57, 169–170, 333–336, 465–467
 weak, 391–393

- referent types, 832
 - members, accessing, 839
 - reflection, 652–662
 - dynamic objects, invoking, 689–690
 - on generic types, 660–662
 - ref parameter values, 234–235, 819–820
 - registering
 - COM DLL, 858
 - for unhandled exceptions, 744–745
 - relational operators, 110–111
 - relationships
 - many-to-many, 569
 - one-to-many, 569, 577–580
 - remainder (%) operator, 85
 - removing
 - conditions, 765
 - whitespace, 12
 - reports
 - errors, 196
 - exceptions, 412
 - reserved words, 4. *See also*
 - keywords
 - reset events, 768–771
 - resolution, methods, 185
 - resources
 - cleanup, 393–400, 790–791
 - managing, 823–824
 - utilization, 400
 - results
 - GroupBy() method, 575–577
 - parallel, 731–734
 - tasks, returning, 709
 - resurrecting objects, 399–400
 - rethrowing exceptions, 197, 413
 - retrieving
 - attributes, 667–668
 - specific attributes, 669
 - return attributes, specifying, 666
 - returning
 - empty collections, 634
 - null, 634
 - task results, 709
 - void, 53
 - returns
 - Main() method, 165–168
 - methods, 159–160, 522–523
 - yield returns, placing in loops, 643–645
 - return statements, 160
 - return values, 15
 - reusing code, 378
 - Reverse() standard query operator, 584
 - reversing strings, 77
 - right outer joins, 569
 - root references, 390
 - round-trip formatting, 39–40
 - Run() method, 285
 - running
 - HelloWorld program, 3–4
 - LINQ queries in parallel, 734–738
 - Parallel LINQ (PLINQ) queries, 559–560
 - threads, 706–738
 - canceling tasks, 718–722
 - disposing tasks, 723–724
 - long-running threads, 722–723
 - unhandled exception handling on Task, 715–718
 - runtime, 24
 - arrays, defining array size at, 68
 - CLI (Common Language Infrastructure), 849–854
 - metadata, reflection, 652–662
 - virtual methods, 283
 - runtime callable wrapper (RCW), 813
- ## S
- SafeHandle, applying, 823–824
 - safety, types, 25, 541, 851
 - scope, 107–109, 155
 - sealed classes, 281
 - sealed modifiers, 291
 - searching
 - attributes, 667–668
 - List<T> class, 619
 - security
 - access, 25
 - CAS (Code Access Security), 659, 852
 - select clause, 590
 - selecting lock objects, 758–759
 - SelectMany() method, calling, 580–582
 - Select() method, 734
 - projecting with, 557–560
 - SemaphoreSlim, 772
 - semaphores over
 - AutoResetEvent, 772
 - semicolons (;)
 - statements without, 10–11
 - whitespace, 11–12
 - SequenceEquals() standard query operator, 584
 - sequences
 - deferred execution, 565
 - escape, 42
 - invocation, 516–517
 - layout,
 - StructLayoutAttribute for, 820–821
 - multithreading, 703. *See also* multithreading
 - serialization
 - attributes, 680–682
 - customizing, 683–684
 - exceptions, 416
 - versioning, 684–687
 - SetName() method, 213
 - setters, access modifiers, 231–232
 - shift operators, 116–117
 - short data types, 33
 - shutdown, applications, 717
 - signatures, APMs (Asynchronous Programming Models), 786–787
 - Silverlight, 536n1
 - simple assignment (=) operators, 14
 - simple generic classes, defining, 429–430
 - simplifying API calls with wrappers, 828–829
 - single inheritance, 278–281
 - single input parameters, statement lambdas with, 489

- single instance applications,
 - creating, 766–767
- single-line comments, 22
- single-line XML comments,
 - 386–387
- single quote ('), 42
- sites, call, 168
- sizing
 - arrays at runtime, 68
 - types, 752
- SortedDictionary<TKey, T> class, 626–628
- SortedList<T> class,
 - 626–628
- sorting
 - collections, 626–628
 - IComparer<T> interface,
 - 614–615
 - LINQ query expressions,
 - 599–600
 - with OrderBy() method
 - and ThenBy() method,
 - 566–572
- space, declaring, 107–109
- specialized Stack classes,
 - defining, 425
- specializing types, 205
- specifiers, exceptions, 408
- specifying
 - constraints, 455
 - default values, 435–436
 - line numbers, 143–144
 - literals, 36
 - multiple constraints, 446
 - parameters by name, 184
 - return attributes, 666
- SQL
 - query expressions, 592
 - where clauses, converting
 - expression trees to,
 - 499
- Stack class, 422
 - specialized, defining, 425
- stacks
 - calling, 168, 836
 - collections, 628
 - unwinding, 168
- Stack<T> class, 628
- standard query operators,
 - 552–586, 582–586
- collection interfaces with,
 - 535–536. *See also*
 - collection interfaces
- Count() method,
 - counting elements
 - with, 561
- deferred execution,
 - 562–566
- grouping results with
 - GroupBy() method,
 - 575–577
- implementing one-to-many relationships,
 - 577–580
- performing inner joins
 - with Join() method,
 - 572–575
- Select() method,
 - projecting with,
 - 557–560
- sorting with OrderBy()
 - method and ThenBy()
 - method, 566–572
- Where() method, filtering
 - with, 556–557
- starting methods, 707
- statements, 10
 - Assert(), 92
 - break, 132–135
 - continue, 135–136
 - control flow, 121–132
 - delimiters, 10
 - goto, 137–138
 - groups into methods, 150
 - if, 102–103, 105
 - jump, 132–138
 - lambdas, 486–489
 - line-based, 10
 - lock, 343
 - versus method calls, 156
 - nested if, 103–105
 - return, 160
 - switch, 130–132, 160
 - System.Console.WriteLine(), 10
 - Throw, 196
 - using, 217n1, 395–398
 - without semicolons (;),
 - 10–11
 - yield, 649
- states
 - APM (Synchronous Programming Model)
 - methods, passing
 - between, 789–790
 - callbacks, invoking,
 - 787
 - iterators and, 639–641
 - unsynchronized, 750
 - STAThreadAttribute, controlling COM threading
 - models with, 813
 - static classes, 255
 - static compilation *versus*
 - dynamic programming,
 - 695–696
 - static constructors,
 - 253–254
 - static fields, 248–250
 - static members, 247–256
 - static methods, 251–253
 - static properties, 254–256
 - Status property, 710
 - storage, local, 774–777
 - Store() method, 216
 - string keyword, 163n2
 - strings, 43–51
 - applying, 50
 - as arrays, 76–78
 - concatenation at compile
 - time, 45
 - conversion, 63
 - enums, 350–351
 - immutable, 16, 49–51
 - length, 48–49
 - literals, 44–46
 - methods, 46–47
 - plus (+) operator, using
 - with, 87–88
 - reversing, 77
 - stringStatic methods, 46
 - string type, avoiding,
 - 759–760
 - strong references, 391
 - struct
 - class constraints, 445
 - defining, 334
 - generics, 432–433
 - initializing, 336–337
 - iterators, 645

- StructLayoutAttribute for sequential layout, applying, 820–821
 - styles
 - code, avoiding ambiguity, 213–217
 - CPS (Continuation Passing Style), 787–789
 - subroutines, defining, 53
 - subscribers
 - connecting, 511–512
 - exceptions, handling, 520
 - methods, defining, 508–510
 - subscriptions, encapsulating, 523–524
 - subtypes, 204
 - Sum() function, 585
 - super types, 204
 - support, interfaces, 440
 - switch statements, 130–132, 160
 - synchronization
 - design best practices, 674
 - lock,
 - ConsoleSyncObject, 797
 - methodImpAttribute, avoiding, 760
 - Monitor class, 754–758
 - threads, 750–777
 - types, 766–774
 - when to provide, 765
 - syntax, 1–2
 - fundamentals, 4–12
 - iterators, 636–637
 - System.Action, 483–484
 - System.ArgumentException, 405
 - System.AsyncCallback, 787–789
 - System.AttributeUsageAttribute, 673–674
 - System.Collections.Generic.ICollection<T>, 544
 - System.Collections.Generic namespace, 153
 - System.Collections namespace, 153
 - System.Collection.Stack, 423
 - System.ConditionalAttribute, 677–679
 - System.Console.ReadKey() method, 18
 - System.Console.ReadLine() method, 16
 - System.Console.WriteLine() statement, 10
 - System.Console.Write() method, 18–20
 - System.Data namespace, 153
 - System-defined delegates: Func, 483–485
 - System.Drawing namespace, 153
 - System.Exception
 - catch blocks, 195–196
 - use of, 412
 - System.IO namespace, 153
 - System.Linq.Enumerable.Where(), 562
 - System.Linq namespace, 153
 - System.Linq.Queryable, 585
 - System namespace, 153
 - System.NonSerializable attribute, 682–683
 - System.Object inheritance, 299–301
 - System.ObsoleteAttribute, 679–680
 - System.Runtime.CompilerServices.CompilerGeneratedAttribute, 236
 - System.Runtime.Serialization.OptionalFieldAttribute, 686
 - Systems.Collections.Concurrent, 895–898
 - System.SerializableAttribute, 687–688
 - Systems.Timer.Timer, 780
 - System.Text namespace, 153
 - System.Text.StringBuilder data type, 51
 - System.Threading.Interlocked class, 761–763
 - System.Threading.Mutex, 766–767
 - System.Threading namespace, 153
 - System.Threading.Tasks namespace, 153
 - System.Threading.Thread, 738–740
 - System.Threading.WaitHandle class, 768–769
 - System.Type, accessing metadata, 653–655
 - System.Web namespace, 154
 - System.Web.Services namespace, 154
 - System.Windows.Forms namespace, 154
 - System.Xml namespace, 154
- T**
- targets, modifying assemblies, 378–379
 - Task.CurrentID property, 711
 - Task Parallel Library (TPL), 703
 - task-related finalization, 717
 - tasks
 - canceling, 718–722
 - disposing, 723–724
 - results, returning, 709
 - templates, C++, 442
 - text, comments, 20–23
 - ThenBy() method, sorting with OrderBy() method, 566–572
 - thermostat, 508n1
 - this keyword, 213–220
 - this type, avoiding, 759–760
 - ThreadLocal<T>, 774–775
 - threads. *See also* multithreading
 - controlling, 706–738
 - data to and from an alternate, passing, 799–801
 - local storage, 774–777
 - long-running, 722–723
 - managing, 740–742

- threads (*contd.*)
 - multiple. *See* multiple threads
 - overview of, 703–706
 - pools, 706, 742–743
 - running, 706–738
 - canceling tasks, 718–722
 - disposing tasks, 723–724
 - long-running threads, 722–723
 - unhandled exception handling on Task, 715–718
 - synchronization, 750–777
 - thread-safe, 752
 - incrementing and decrementing, 96
 - ThreadStaticAttribute, 775–777
 - three-dimensional arrays, initializing, 69
 - three-forward-slash (///), 387
 - throwing exceptions, 406–407
 - Throw statement, 196
 - Tic-Tac-Toe, 869–874
 - timers, 778–783
 - time slices, 704
 - torn reads, 753
 - ToString() method, overriding, 358
 - ToUpper() method, 50
 - TPL (Task Parallel Library), 703
 - APMs (Asynchronous Programming Models), calling, 791–796
 - trapping errors, 187–192
 - trees, expressions, 498–505
 - object graphs, 499–501
 - viewing, 503–505
 - troubleshooting arrays, 69, 78–80
 - true operator, 373–375
 - TryParse() method, 63
 - numeric conversion with, 198–199
 - Tuple generic types, 437–438
 - turning off warning messages (#pragma), 142–143
 - two-dimensional arrays. *See also* arrays
 - declaring, 68
 - initializing, 69
 - Type alias, declaring, 164
 - typeof expressions, 654–655
 - typeof type, avoiding, 759–760
 - types
 - aliasing, 164–165
 - anonymous, 245–246
 - collection interfaces, 536–538
 - implicit local variables, 54
 - projection to, 558
 - base, casting between
 - derived and, 272–273
 - Boolean, 40–41
 - categories of, 55–57, 332–339
 - Cell, 427
 - char, 41
 - checking, 851
 - comments, 21–22
 - compatibility between
 - enums, 349–350
 - conversion without casting, 62
 - data, 13–14. *See also* data types
 - delegates, 472–473
 - parameters, 818–819
 - decimal, 34–35
 - definitions, 7–8
 - delegates, defining, 474–475
 - encapsulation of, 379–380
 - enums, defining, 348
 - exceptions, 193–194
 - floating-point, 33–34
 - inequality with, 89–92
 - special characteristics of, 89
 - generics, 427–439
 - nested, 438–439
 - reflection, 660–662
 - Tuple, 437–438
 - inferencing, 454–455
 - inheritance, 205
 - integers, 32–33
 - metadata, reflection, 652–662
 - multiple exception, 405–407
 - names, methods, 154
 - numeric, 32–40
 - parameterized, 427
 - parameters, 429
 - in, 460–462
 - determining type of, 660–661
 - multiple, 436
 - naming conventions, 431
 - out, 458–460
 - predefined, 31
 - references, 56–57, 169–170, 333–336, 465–467
 - referent, 832, 839
 - safety, 25, 541, 851
 - sizes, 752
 - specializing, 205
 - string, avoiding, 759–760
 - synchronization, 766–774
 - this, avoiding, 759–760
 - typeof, avoiding, 759–760
 - underlying
 - unboxing, 342
 - verifying, 301
 - unmanaged, 833
 - values, 55–56, 169–170, 331, 332
 - boxing, 339–346
 - enums, 346–355
 - inheritance, 338–339
 - instantiating generics based on, 464–465
 - interfaces, 338–339
 - nullable, 425–427
 - well-formed, 357. *See also* well-formed types

U

UML (Unified Modeling Language), 204, 325n1

unary operators, 373–375

- minus (-), 84–92
- plus (+), 84–92

unboxing, 339, 342

- avoiding, 345

uncertainty, multithreading, 706

unchecked conversions, 59–61, 417–419

underlying types

- unboxing, 342
- verifying, 301

underscore (_), 15

unhandled exceptions

- on `AppDomain`, 744–746
- handling on `Task`, 715–718

Unicode characters, 41–43

Unified Modeling Language.

- See* UML

`Union()` standard query operator, 584

unmanaged types, 833

unnecessary locking, avoiding, 765–766

unsafe code, 831–832

unsynchronized states, 750

unwinding stacks, 168

updating `CommandLineHandler.TryParse()` method, 671

uppercase, Java, 9

using directive, 161–168

using statements, 217n1, 395–398

utilization of resources, 400

V

hexidecimal notation, 38

integers, overflowing, 59

iterators, yielding, 637–639

literals, 35, 68

parameters, 168–169

permanent, 259

placeholders, 115

types, 55–56, 169–170, 331, 332

- boxing, 339–346
- enums, 346–355
- inheritance, 338–339
- instantiating generics based on, 464–465
- interfaces, 338–339
- nullable, 425–427
- variables, modifying, 15

variables

- applying, 12–16
- assigning, 13, 14–16
- declaring, 13, 14
- applying anonymous methods, 481
- of the `Class Type`, 206
- implicitly typed local, 53–55
- local, 13
- implicitly typed, 538–540
- multiple threads, 753–753
- members, 209
- outer, 495–496
- parameters, defining index operators, 633–634
- values, modifying, 15

variance, applying delegates, 485

`var` keyword, 53

Venn diagrams, 568

verbatim string literals, 44

verifying underlying types, 301

versioning

- interfaces, 327–328
- .NET, 26–27
- serialization, 684–687

VES (Virtual Execution System), 24, 844

viewing expression trees, 503–505

`VirtualAllocEx()` APIs, declaring, 818–819

virtual computers, 816

- detection using `P/Invoke`, 888–894

Virtual Execution System.

- See* VES

virtual fields, properties as, 232–234

virtual modifiers, 282–286

Visual Basic

- accessing class instances with `Me` keyword, 214
- global methods, 158
- global variables and functions, 248
- line-based statements, 10
- redimensioning arrays, 75
- `var`, 540
- `Variant`, 540
- `void*`, 540

Visual Basic.NET, project scope `Imports` directive, 162

visual editors, hints for, 144–145

Visual Studio, XML comments in, 386

`void` keyword, 52–55

`volatile`, declaring fields as, 760–761

W

weak references, 391–393

well-formed types, 357

- accessibility modifiers, 381
- assemblies, referencing, 377–381
- garbage collection, 390–393
- lazy initialization, 400–402
- namespaces, defining, 382–385
- object members, overriding, 357–369
- operators, overloading, 369–377
- resource cleanup, 393–400
- XML comments, 385–389

where clauses, converting expression trees to, 499

- where() method, filtering
 - with, 556–557
- while loops, 121–123
- whitespace, formatting,
 - 11–12
- wildcards in import directive, 162
- Win32 APIs, declaring, 818n1
- Windows
 - Error Reporting dialog box, 715
 - executable, 378
 - Forms, 809–811
 - Presentation Foundation (WPF), 811–813
 - UI programming, 809–813
- WPF (Windows Presentation Foundation), 811–813
- wrappers
 - RCW (runtime callable wrapper), 813
 - simplifying API calls with, 828–829
- write-only properties, 230–231
- writing
 - comments, 20–23
 - output to consoles, 18–20
- www.pinvoke.net, 818n1
- X**
 - XML (Extensible Markup Language), 22–23
 - comments, 385–389
 - delimited comments, 22
 - documentation files, generating, 388–389
 - single-line comments, 22
 - XOR (exclusive OR) operator, 112
- Y**
 - yield break, iterators, 645–646
 - yielding iterator values, 637–639
 - yield keyword, 6n5
 - yield returns, placing in loops, 643–645
 - yield statements, 649