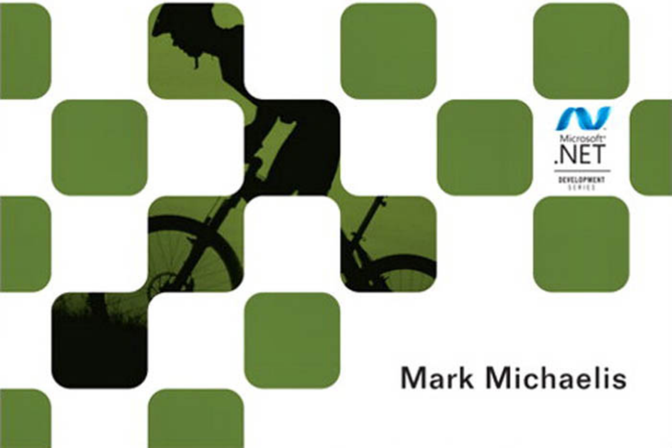
"If you want to be a C# developer, or if you want to enhance your C# programming skills, there is no more useful tool than a well-crafted book on the subject. You are holding such a book in your hands."

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

Essential C# 4.0



Mark Michaelis

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

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
```

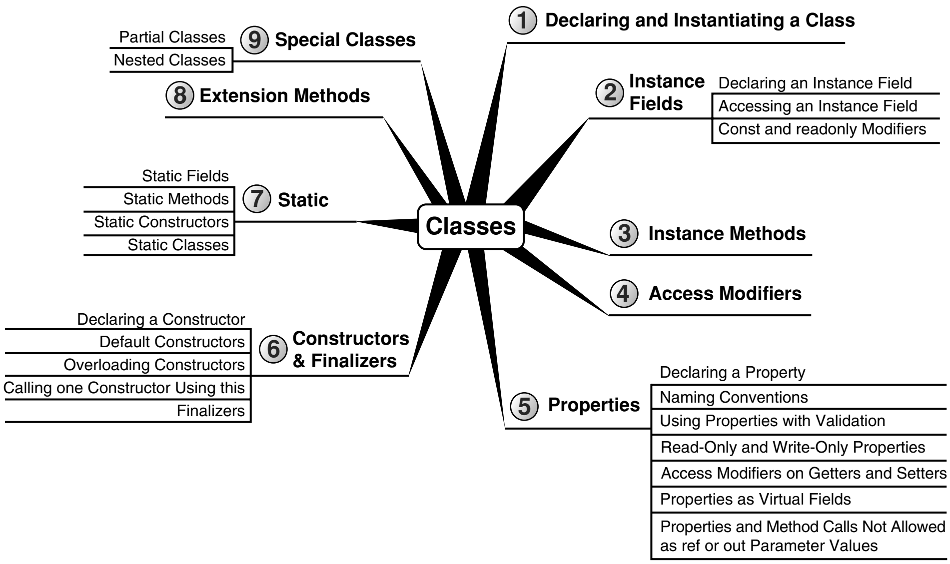
- User input for the program appears in italics.

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 mateial at intelliTechture.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.

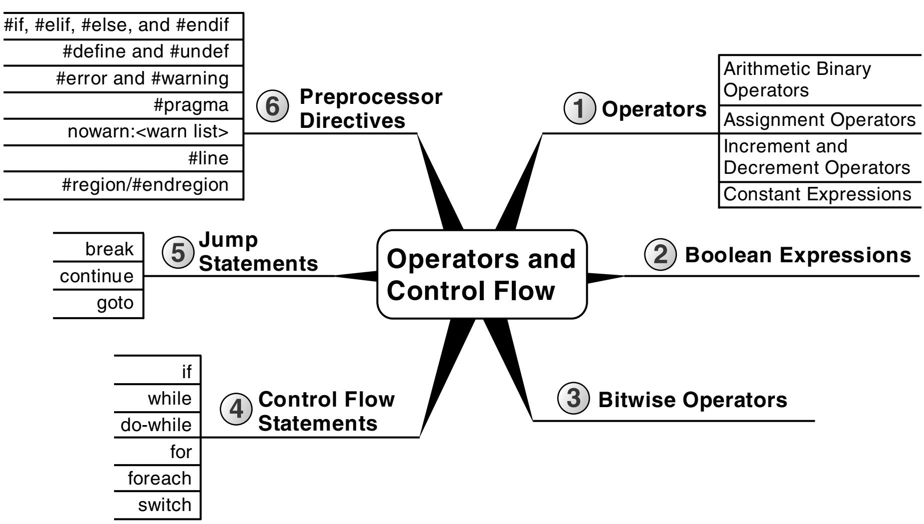
I hope you find this book to be a great resource in establishing your C# expertise and that you continue to reference it for the more obscure areas of C# and its inner workings.

—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 Values[1]

```
//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 rarely[2] 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.Write("{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()
{                                          Constant Expression
    const int secondsPerDay = 60 * 60 * 24;
    const int secondsPerWeek = secondsPerDay * 7;
                  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	`if(boolean-expression)`
	`  embedded-statement`
		`if (input == "quit")`
		`{`
		`  System.Console.WriteLine(`
		`    "Game end");`
		`  return;`
		`}`

	`if(boolean-expression)`
	`  embedded-statement`
	`else`
	`  embedded-statement`
		`if (input == "quit")`
		`{`
		`  System.Console.WriteLine(`
		`    "Game end");`
		`  return;`
		`}`
		`else`
		`  GetNextMove();`

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

Continues

TABLE 3.1: Control Flow Statements *(Continued)*

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

TABLE 3.1: Control Flow Statements *(Continued)*

<table>
<tr><th>Statement</th><th>General Syntax Structure</th><th>Example</th></tr>
<tr><td>switch statement</td><td>switch(governing-type-expression)
{
  ...
  case const-expression:
    statement-list
    jump-statement
  default:
    statement-list
    jump-statement
}</td><td rowspan="5">switch(input)
{
  Case "exit":
  case "quit":
    System.Console.WriteLine(
      "Exiting app....");
    break;
  case "restart":
    Reset();
    goto case "start";
  case "start":
    GetMove();
    break;

  default:

    System.Console.WriteLine(
      input);

    break;
}</td></tr>
<tr><td>break statement</td><td>break;</td></tr>
<tr><td rowspan="3">goto statement</td><td>goto identifier;</td></tr>
<tr><td>goto case const-expression;</td></tr>
<tr><td>goto default;</td></tr>
</table>

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-toe[3] 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# Preprocessor Directives or skip to the Summary section at the end of the chapter.

`if` Statement

The `if` statement is one of the most common statements in C#. It evaluates a **Boolean expression** (an expression that 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

```
class TicTacToeTrivia
{
  static void Main()
  {
      int input;    // Declare a variable to store the input.

      System.Console.Write(
          "What is the maximum number " +
          "of turns in tic-tac-toe?" +
          "(Enter 0 to exit.): ");

      // int.Parse() converts the ReadLine()
      // return to an int data type.
      input = int.Parse(System.Console.ReadLine());

      if (input <= 0)
              // Input is less than or equal to 0.
          System.Console.WriteLine("Exiting...");
      else
          if (input < 9)
              // Input is less than 9.
              System.Console.WriteLine(
                  "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)    // COMPILE 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	`input<9;`
`>`	Greater than	`input>9;`
`<=`	Less than or equal to	`input<=9;`
`>=`	Greater than or equal to	`input>=9;`
`==`	Equality operator	`input==9;`
`!=`	Inequality operator	`input!=9;`

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((hourOfTheDay > 23) || (hourOfTheDay < 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 `hourOfTheDay` has the value `33` then `(hourOfTheDay > 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 < hourOfTheDay) && (hourOfTheDay < 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.

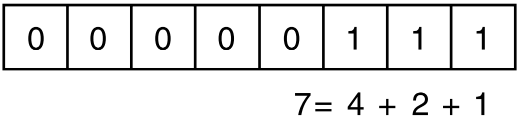


FIGURE 3.2: Calculating the Value of an Unsigned Byte

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 leftmost location. If the leftmost location contains a 1, you add the locations with 0s rather than the locations with 1s. Each location corresponds to the negative power of 2 value. Furthermore, from the result, it is also necessary to subtract 1. This is demonstrated in Figure 3.3.

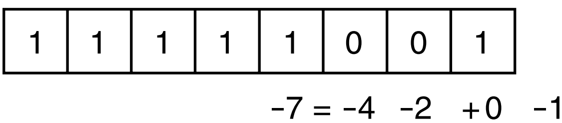


FIGURE 3.3: Calculating the Value of a Signed Byte

Therefore, `1111 1111 1111 1111` corresponds to –1 and `1111 1111 1111 1001` holds the value –7. `1000 0000 0000 0000` corresponds to the lowest negative value that a 16-bit integer can hold.

Shift Operators (<<, >>, <<=, >>=)

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

Consider the following example. Suppose you had the `int` value `-7`, which would have a binary representation of `1111 1111 1111 1111 1111 1111 1111 1001`. In Listing 3.37, you right-shift the binary representation of the number –7 by two locations.

Listing 3.37: Using the Right-Shift Operator

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

Output 3.17 shows the results of Listing 3.37.

Output 3.17:

```
x = -2.
```

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

Bitwise Operators (&, |, ^)

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

BEGINNER TOPIC

Logical Operators Explained

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

12:	0	0	0	0	1	1	0	0
7:	0	0	0	0	0	1	1	1

FIGURE 3.4: The Numbers 12 and 7 Represented in Binary

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

Listing 3.38 demonstrates how to use these bitwise operators. The results of Listing 3.38 appear in Output 3.18.

LISTING 3.38: Using Bitwise Operators

```
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 that converts an integer to a string of its binary representation. The results of Listing 3.39 appear in Output 3.19.

LISTING 3.39: Getting a String Representation of a Binary Display

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

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

      // Set initial mask to 100....
      ulong mask = 1ul << size - 1;
      for (int count = 0; count < size; count++)
      {
          bit = ((mask & value) > 0) ? '1': '0';
          System.Console.Write(bit);
          // Shift mask one location over to the right
          mask >>= 1;
      }
      System.Console.WriteLine();
  }
}
```

OUTPUT 3.19:

```
Enter an integer: 42
0000000000000000000000000000000000000000000000000000000000101010
```

Notice that within each iteration of the `for` loop (discussed shortly), you use the right-shift assignment operator to create a mask corresponding to each bit in `value`. By using the `&` bit operator to mask a particular bit, you can determine whether the bit is set. If the mask returns a positive result, you set the corresponding bit to `1`; otherwise, it is set to `0`. In this way, you create a string representing the binary value of an unsigned long.

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.

The for Loop

Increment and decrement operators are frequently used within a `for` loop. The `for` loop iterates a code block until a specified condition is reached in a way similar to the `while` loop. The difference is that the `for` loop has built-in syntax for initializing, incrementing, and testing the value of a counter.

Listing 3.43 shows the `for` loop used to display an integer in binary form. The results of this listing appear in Output 3.21.

Listing 3.43: Using the for Loop

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

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

      // Set initial mask to 100....
      ulong mask = 1ul << size - 1;
      for (int count = 0; count < size; count++)
      {
          bit = ((mask & value) > 0) ? '1': '0';
          System.Console.Write(bit);
          // Shift mask one location over to the right
          mask >>= 1;
      }
  }
}
```

Output 3.21:

```
Enter an integer: -42
1111111111111111111111111111111111111111111111111111111111010110
```

Listing 3.43 performs a bit mask 64 times, once for each bit in the number. The `for` loop declares and initializes the variable `count`, escapes once the count reaches 64, and increments the count during each iteration. Each 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.Write("{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 != 'O' && cell != 'X')
        {
            System.Console.Write("{0} ", cell);
        }
      }
  }
}
```

Output 3.23 shows the results of Listing 3.45.

Output 3.23:

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

When the execution engine reaches the `foreach` statement, it assigns to the variable `cell` the first item in the `cells` array—in this case, the value `'1'`. It then executes the code within the `foreach` statement block. The `if` statement determines whether the value of `cell` is `'O'` 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 | O
      // ---+---+---
      //  O | O | 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.

OUTPUT 3.24:

```
Player 1 was the winner
```

Listing 3.48 uses a `break` statement when a player holds a winning position. The `break` statement forces its enclosing loop (or a `switch` statement) to cease execution, and the program moves to the next line outside the loop. For this listing, if the bit comparison returns `true` (if the board holds a winning position), the `break` statement causes execution to jump and display the winner.

BEGINNER TOPIC

Bitwise Operators for Positions

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

To begin, both `playerPositions` are `0`. As each player moves, the bit corresponding to the move is set. If, for example, the player selects cell 3, `shifter` is set to `3 - 1`. The code subtracts 1 because C# is zero-based and you need to adjust for 0 as the first position instead of 1. Next, the code sets `position`, the bit corresponding to cell 3, using the shift operator `000000000000001 << shifter`, where `shifter` now has a value of `2`. Lastly, it sets `playerPositions` for the current player (subtracting 1 again to shift to zero-based) to `0000000000000100`. Listing 3.49 uses `|=` so that previous moves are combined with the current move.

LISTING 3.49: Setting the Bit That Corresponds to Each Player's Move

```
int shifter;  // The number of places to shift
              // over in order to set a bit.
int position;  // The bit which is to be set.

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

// Shift mask of 00000000000000000000000000000001
// over by cellLocations.
```

```
position = 1 << shifter;

// Take the current player cells and OR them to set the
// new position as well.
// Since currentPlayer is either 1 or 2,
// subtract one to use currentPlayer as an
// index in a 0-based array.
playerPositions[currentPlayer-1] |= position;
```

Later in the program, you can iterate over each mask corresponding to winning positions on the board to determine whether the current player has a winning position, as shown in Listing 3.48.

The `continue` Statement

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 complier 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).

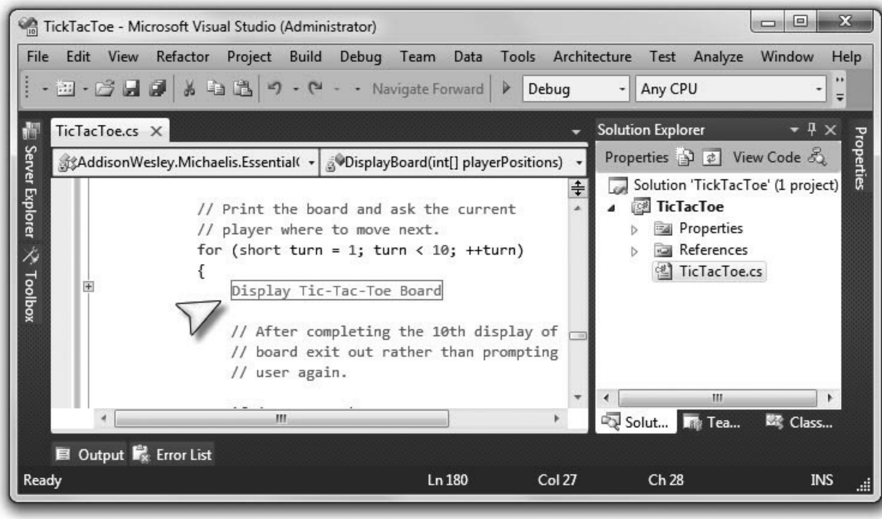


FIGURE 3.5: Collapsed Region in Microsoft Visual Studio .NET

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*

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

* 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

F

I

J

N

O

P

Q

R

S

T

U

V

W