Microsoft

Microsoft Visual C# Step by Step

Eighth Edition

Professional

John Sharp

Microsoft

Microsoft Visual C# Step by Step, 8th Edition

John Sharp

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2015940217
ISBN: 978-1-5093-0104-1

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

Acquisitions and Developmental Editor: Devon Musgrave
Project Editor: John Pierce
Editorial Production: Rob Nance and John Pierce
Technical Reviewer: Marc Young
Copyeditor: John Pierce
Indexer: Christina Yeager, Emerald Editorial Services
Cover: Twist Creative • Seattle and Joel Panchot

Contents at a glance

Contents

Introduction

Microsoft Visual C# is a powerful but simple language aimed primarily at developers who create applications built on the Microsoft .NET Framework. Visual C# inherits many of the best features of C++ and Microsoft Visual Basic, but few of the inconsistencies and anachronisms, which results in a cleaner and more logical language. C# 1.0 made its public debut in 2001. With the advent of C# 2.0 with Visual Studio 2005, several important new features were added to the language, including generics, iterators, and anonymous methods. C# 3.0, which was released with Visual Studio 2008, added extension methods, lambda expressions, and most famously of all, the Language-Integrated Query facility, or LINQ. C# 4.0, was released in 2010 and provided further enhancements that improved its interoperability with other languages and technologies. These features included support for named and optional arguments and the *dynamic* type, which indicates that the language runtime should implement late binding for an object. An important addition to the .NET Framework, and released concurrently with C# 4.0, were the classes and types that constitute the Task Parallel Library (TPL). Using the TPL, you can build highly scalable applications that can take full advantage of multicore processors. C# 5.0 added native support for asynchronous task-based processing through the *async* method modifier and the *await* operator. C# 6.0 is an incremental upgrade with features that are intended to make life simpler for developers. These features include items such as string interpolation (you need never use *String.Format* again!), enhancements to the ways in which properties are implemented, expression-bodied methods, and others. They are all described in this book.

Another important event for Microsoft is the launch of Windows 10. This new version of Windows combines the best (and most loved) aspects of previous versions of the operating system and supports highly interactive applications that can share data and collaborate as well as connect to services running in the cloud. The key notion in Windows 10 is Universal Windows Platform (UWP) apps—applications designed to run on any Windows 10 device, whether a fully fledged desktop system, a laptop, a tablet, a smartphone, or even an IoT (Internet of Things) device with limited resources. Once you have mastered the core features of C#, gaining the skills to build applications that can run on all these platforms is important.

Voice activation is another feature that has come to the fore, and Windows 10 includes Cortana, your personal voice-activated digital assistant. You can integrate your own apps with Cortana to allow them to participate in data searches and other operations. Despite the complexity normally associated with natural-language speech analysis, it is surprisingly easy to enable your apps to respond to Cortana's requests, and I cover this in Chapter 26. Additionally, the cloud has become such an important

element in the architecture of many systems, ranging from large-scale enterprise applications to mobile apps running on users smartphones, that I decided to focus on this aspect of development in the final chapter of the book.

The development environment provided by Visual Studio 2015 makes these features easy to use, and the many new wizards and enhancements included in the latest version of Visual Studio can greatly improve your productivity as a developer. I hope you have as much fun working through this book as I had writing it!

Who should read this book

This book assumes that you are a developer who wants to learn the fundamentals of programming with C# by using Visual Studio 2015 and the .NET Framework version 4.6. By the time you complete this book, you will have a thorough understanding of C# and will have used it to build responsive and scalable applications that can run on the Windows 10 operating system.

Who should not read this book

This book is aimed at developers new to C# but not completely new to programming. As such, it concentrates primarily on the C# language. This book is not intended to provide detailed coverage of the multitude of technologies available for building enterprise-level applications for Windows, such as ADO.NET, ASP.NET, Windows Communication Foundation, or Windows Workflow Foundation. If you require more information on any of these items, you might consider reading some of the other titles available from Microsoft Press.

Organization of this book

This book is divided into four sections:

- Part I, "Introducing Microsoft Visual C# and Microsoft Visual Studio 2015," provides an introduction to the core syntax of the C# language and the Visual Studio programming environment.
- Part II, "Understanding the C# object model," goes into detail on how to create and manage new types in C# and how to manage the resources referenced by these types.

- Part III, "Defining extensible types with C#," includes extended coverage of the elements that C# provides for building types that you can reuse across multiple applications.
- Part IV, "Building Universal Windows Platform applications with C#," describes the universal Windows 10 programming model and how you can use C# to build interactive applications for this new model.

Finding your best starting point in this book

This book is designed to help you build skills in a number of essential areas. You can use this book if you are new to programming or if you are switching from another programming language such as C, C++, Java, or Visual Basic. Use the following table to find your best starting point.

If you are	Follow these steps
New to object-oriented programming	1. Install the practice files as described in the upcoming section, "Code samples." 2. Work through the chapters in Parts I, II, and III sequentially. 3. Complete Part IV as your level of experience and interest dictates.
Familiar with procedural programming languages such as C but new to C#	1. Install the practice files as described in the upcoming section, "Code samples." 2. Skim the first five chapters to get an overview of C# and Visual Studio 2015, and then concentrate on Chapters 6 through 22. 3. Complete Part IV as your level of experience and interest dictates.
Migrating from an object-oriented language such as C++ or Java	1. Install the practice files as described in the upcoming section, "Code samples." 2. Skim the first seven chapters to get an overview of C# and Visual Studio 2015, and then concentrate on Chapters 8 through 22. 3. For information about building Universal Windows Platform applications, read Part IV.

If you are	Follow these steps
Switching from Visual Basic to C#	1. Install the practice files as described in the upcoming section, "Code samples." 2. Work through the chapters in Parts I, II, and III sequentially. 3. For information about building Universal Windows Platform applications, read Part IV. 4. Read the Quick Reference sections at the end of the chapters for information about specific C# and Visual Studio 2015 constructs.
Referencing the book after working through the exercises	1. Use the index or the table of contents to find information about particular subjects. 2. Read the Quick Reference sections at the end of each chapter to find a brief review of the syntax and techniques presented in the chapter.

Most of the book's chapters include hands-on samples that let you try out the concepts you just learned. No matter which sections you choose to focus on, be sure to download and install the sample applications on your system.

Conventions and features in this book

This book presents information by using conventions designed to make the information readable and easy to follow.

- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.
- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- Windows 10 Professional (or above) edition.
- Visual Studio Community 2015 edition, Visual Studio Professional 2015 edition, or Visual Studio Enterprise 2015 edition.

Important You must install the Windows 10 developer tools with Visual Studio 2015.

- Computer that has a 1.6 GHz or faster processor (2 GHz recommended).
- 1 GB (32-bit) or 2 GB (64-bit) RAM (add 512 MB if running in a virtual machine).
- 10 GB of available hard disk space.
- 5400 RPM hard-disk drive.
- DirectX 9–capable video card running at 1024 × 768 or higher resolution display.
- DVD-ROM drive (if installing Visual Studio from a DVD).
- Internet connection to download software or chapter examples.

Depending on your Windows configuration, you might require local Administrator rights to install or configure Visual Studio 2015.

You also need to enable developer mode on your computer to be able to create and run UWP apps. For details on how to do this, see "Enable Your Device for Development," at *https://msdn.microsoft.com/library/windows/apps/dn706236.aspx*.

Code samples

Most of the chapters in this book include exercises with which you can interactively try out new material learned in the main text. You can download all the sample projects, in both their preexercise and postexercise formats, from the following page:

http://aka.ms/sharp8e/companioncontent

Note In addition to the code samples, your system should have Visual Studio 2015 installed. If available, install the latest service packs for Windows and Visual Studio.

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. Unzip the CSharpSBS.zip file that you downloaded from the book's website, extracting the files into your Documents folder.
2. If prompted, review the end-user license agreement. If you accept the terms, select the Accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the CSharpSBS.zip file.

Using the code samples

Each chapter in this book explains when and how to use any code samples for that chapter. When it's time to use a code sample, the book will list the instructions for how to open the files.

Important Many of the code samples have dependencies on NuGet packages that are not included with the code. These packages are downloaded automatically the first time you build a project. As a result, if you open a project and examine the code before doing a build, Visual Studio may report a large number of errors for unresolved references. Building the project will cause these references to be resolved, and the errors should disappear.

For those of you who like to know all the details, here's a list of the sample Visual Studio 2015 projects and solutions, grouped by the folders where you can find them. In many cases, the exercises provide starter files and completed versions of the same projects that you can use as a reference. The completed projects for each chapter are stored in folders with the suffix "- Complete".

Project/Solution	Description
Chapter 1	
TextHello	This project gets you started. It steps through the creation of a simple program that displays a text-based greeting.
Hello	This project opens a window that prompts the user for his or her name and then displays a greeting.
Chapter 2	
PrimitiveDataTypes	This project demonstrates how to declare variables by using each of the primitive types, how to assign values to these variables, and how to display their values in a window.
MathsOperators	This program introduces the arithmetic operators (+ – * / %).
Chapter 3	
Methods	In this project, you'll reexamine the code in the MathsOperators project and investigate how it uses methods to structure the code.
DailyRate	This project walks you through writing your own methods, running the methods, and stepping through the method calls by using the Visual Studio 2015 debugger.
DailyRate Using Optional Parameters	This project shows you how to define a method that takes optional parameters and call the method by using named arguments.
Chapter 4	
Selection	This project shows you how to use a cascading *if* statement to implement complex logic, such as comparing the equivalence of two dates.
SwitchStatement	This simple program uses a *switch* statement to convert characters into their XML representations.
Chapter 5	
WhileStatement	This project demonstrates a *while* statement that reads the contents of a source file one line at a time and displays each line in a text box on a form.
DoStatement	This project uses a *do* statement to convert a decimal number to its octal representation.

Project/Solution	Description
Chapter 6	
MathsOperators	This project revisits the MathsOperators project from Chapter 2 and shows how various unhandled exceptions can make the program fail. The *try* and *catch* keywords then make the application more robust so that it no longer fails.
Chapter 7	
Classes	This project covers the basics of defining your own classes, complete with public constructors, methods, and private fields. It also shows how to create class instances by using the *new* keyword and how to define static methods and fields.
Chapter 8	
Parameters	This program investigates the difference between value parameters and reference parameters. It demonstrates how to use the *ref* and *out* keywords.
Chapter 9	
StructsAndEnums	This project defines a *struct* type to represent a calendar date.
Chapter 10	
Cards	This project shows how to use arrays to model hands of cards in a card game.
Chapter 11	
ParamsArray	This project demonstrates how to use the *params* keyword to create a single method that can accept any number of *int* arguments.
Chapter 12	
Vehicles	This project creates a simple hierarchy of vehicle classes by using inheritance. It also demonstrates how to define a virtual method.
ExtensionMethod	This project shows how to create an extension method for the *int* type, providing a method that converts an integer value from base 10 to a different number base.
Chapter 13	
Drawing	This project implements part of a graphical drawing package. The project uses interfaces to define the methods that drawing shapes expose and implement.
Drawing Using Interfaces	This project acts as a starting point for extending the Drawing project to factor common functionality for shape objects into abstract classes.

Project/Solution	Description
Chapter 14	
GarbageCollectionDemo	This project shows how to implement exception-safe disposal of resources by using the Dispose pattern.
Chapter 15	
Drawing Using Properties	This project extends the application in the Drawing project developed in Chapter 13 to encapsulate data in a class by using properties.
AutomaticProperties	This project shows how to create automatic properties for a class and use them to initialize instances of the class.
Chapter 16	
Indexers	This project uses two indexers: one to look up a person's phone number when given a name and the other to look up a person's name when given a phone number.
Chapter 17	
BinaryTree	This solution shows you how to use generics to build a type-safe structure that can contain elements of any type.
BuildTree	This project demonstrates how to use generics to implement a type-safe method that can take parameters of any type.
Chapter 18	
Cards	This project updates the code from Chapter 10 to show how to use collections to model hands of cards in a card game.
Chapter 19	
BinaryTree	This project shows you how to implement the generic *IEnumerator<T>* interface to create an enumerator for the generic *Tree* class.
IteratorBinaryTree	This solution uses an iterator to generate an enumerator for the generic *Tree* class.
Chapter 20	
Delegates	This project shows how to decouple a method from the application logic that invokes it by using a delegate. The project is then extended to show how to use an event to alert an object to a significant occurrence, and how to catch an event and perform any processing required
Chapter 21	
QueryBinaryTree	This project shows how to use LINQ queries to retrieve data from a binary tree object.

Project/Solution	Description
Chapter 22	
ComplexNumbers	This project defines a new type that models complex numbers and implements common operators for this type.
Chapter 23	
GraphDemo	This project generates and displays a complex graph on a UWP form. It uses a single thread to perform the calculations.
Parallel GraphDemo	This version of the GraphDemo project uses the *Parallel* class to abstract out the process of creating and managing tasks.
GraphDemo With Cancellation	This project shows how to implement cancellation to halt tasks in a controlled manner before they have completed.
ParallelLoop	This application provides an example showing when you should not use the *Parallel* class to create and run tasks.
Chapter 24	
GraphDemo	This is a version of the GraphDemo project from Chapter 23 that uses the *async* keyword and the *await* operator to perform the calculations that generate the graph data asynchronously.
PLINQ	This project shows some examples of using PLINQ to query data by using parallel tasks.
CalculatePI	This project uses a statistical sampling algorithm to calculate an approximation for pi. It uses parallel tasks.
Chapter 25	
Customers	This project implements a scalable user interface that can adapt to different device layouts and form factors. The user interface applies XAML styling to change the fonts and background image displayed by the application.
Chapter 26	
DataBinding	This is a version of the Customers project that uses data binding to display customer information retrieved from a data source in the user interface. It also shows how to implement the *INotifyPropertyChanged* interface so that the user interface can update customer information and send these changes back to the data source.
ViewModel	This version of the Customers project separates the user interface from the logic that accesses the data source by implementing the Model-View-ViewModel pattern.
Cortana	This project integrates the Customers app with Cortana. A user can issue voice commands to search for customers by name.

Project/Solution	Description
Chapter 27	
Web Service	This solution includes a web application that provides an ASP.NET Web API web service that the Customers application uses to retrieve customer data from a SQL Server database. The web service uses an entity model created with the Entity Framework to access the database.

Acknowledgments

Despite the fact that my name is on the cover, authoring a book such as this is far from a one-man project. I'd like to thank the following people who have provided unstinting support and assistance throughout this exercise.

First, Devon Musgrave at Microsoft Press, who awoke me from my interedition slumber. (I was actually quite busy writing material for Microsoft Patterns & Practices but managed to take a sabbatical to work on this edition of the book.) He prodded, cajoled, and generally made sure I was aware of the imminent arrival of Windows 10 and Visual Studio 2015, drew up the contract, and made sure that I signed it in blood, with agreed delivery dates!

Next, Jason Lee, my former underling and now immediate boss at Content Master (it's a somewhat complicated story, but he seems to have found some interesting photographic negatives I left lying carelessly around). He took on much of the initial donkey work generating new screen shots and making sure that the code for the first 20 or so chapters was updated (and corrected) for this edition. If there are any errors, I would like to point the finger at him, but of course any issues and mistakes are entirely my responsibility for missing them during review.

I also need to thank Marc Young, who had the rather tedious task of examining my code to make sure it stood a decent chance of compiling and running. His advice was extremely useful.

Of course, like many programmers, I might understand the technology, but my prose is not always as fluent or clear as it could be. I would like to show my gratitude to John Pierce for correcting my grammar, fixing my speling, and generally making my material much easier to understand.

Finally, I must thank my long-suffering wife, Diana, who thought I was going slowly mad (maybe I was) when I started uttering peculiar phrases at my laptop to try to coax Cortana into understanding my application code. She thought I was continually on the phone to Orlando Gee (one of the sample customers used in the exercises toward the

end of the book) because I kept repeating his name quite loudly. Sadly, because of my accent, Cortana kept thinking I was asking for Orlando T, Orlando Key, or even Orlando Quay, so I subsequently changed the example to refer to Brian Johnson instead. At one point I overheard a conversation Diana was having with our decorator, Jason (who was painting our hallway at the time), about whether he would be able to convert one of the bedrooms into a padded cell, such was her concern with my state of mind! Still, that is now all water under the bridge, or "water under the breach" if you are Cortana trying to recognize my hybrid Gloucestershire/Kentish mode of speech.

And finally, finally, my daughter Francesca would be frightfully upset if I didn't mention her. Although she still lives at home, she is all grown up and now works for a software development company in Cam, Gloucestershire (they didn't offer me any freebies, so I am not going to mention their name).

Errata and book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at:

http://aka.ms/sharp8e/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback is our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 4

Using decision statements

After completing this chapter, you will be able to:

- Declare Boolean variables.
- Use Boolean operators to create expressions whose outcome is either true or false.
- Write *if* statements to make decisions based on the result of a Boolean expression.
- Write *switch* statements to make more complex decisions.

Chapter 3, "Writing methods and applying scope," shows how to group related statements into methods. It also demonstrates how to use parameters to pass information to a method and how to use *return* statements to pass information out of a method. Dividing a program into a set of discrete methods, each designed to perform a specific task or calculation, is a necessary design strategy. Many programs need to solve large and complex problems. Breaking up a program into methods helps you to understand these problems and focus on how to solve them, one piece at a time.

The methods in Chapter 3 are very straightforward, with each statement executing sequentially after the previous statement completes. However, to solve many real-world problems, you also need to be able to write code that selectively performs different actions and that takes different paths through a method depending on the circumstances. In this chapter, you'll learn how to accomplish this task.

Declaring Boolean variables

In the world of C# programming (unlike in the real world), everything is black or white, right or wrong, true or false. For example, if you create an integer variable called *x*, assign the value 99 to it, and then ask whether *x* contains the value 99, the answer is definitely true. If you ask if *x* is less than 10, the answer is definitely false. These are examples of *Boolean expressions*. A Boolean expression always evaluates to true or false.

Note The answers to these questions are not necessarily definitive for all other programming languages. An unassigned variable has an undefined value, and you cannot, for example, say that it is definitely less than 10. Issues such as this one are a common source of errors in C and C++ programs. The Microsoft Visual C# compiler solves this problem by ensuring that you always assign a value to a variable before examining it. If you try to examine the contents of an unassigned variable, your program will not compile.

Visual C# provides a data type called *bool*. A *bool* variable can hold one of two values: *true* or *false*. For example, the following three statements declare a *bool* variable called *areYouReady*, assign *true* to that variable, and then write its value to the console:

```
bool areYouReady;
areYouReady = true;
Console.WriteLine(areYouReady); // writes True to the console
```

Using Boolean operators

A Boolean operator is an operator that performs a calculation whose result is either true or false. C# has several very useful Boolean operators, the simplest of which is the *NOT* operator, represented by the exclamation point (!). The *!* operator negates a Boolean value, yielding the opposite of that value. In the preceding example, if the value of the variable *areYouReady* is true, the value of the expression *!areYouReady* is false.

Understanding equality and relational operators

Two Boolean operators that you will frequently use are equality (==) and inequality (*!*=). These are binary operators with which you can determine whether one value is the same as another value of the same type, yielding a Boolean result. The following table summarizes how these operators work, using an *int* variable called *age* as an example.

Operator	Meaning	Example	Outcome if age is 42
==	Equal to	`age == 100`	false
!=	Not equal to	`age != 0`	true

Don't confuse the *equality* operator == with the *assignment* operator =. The expression *x*==*y* compares *x* with *y* and has the value *true* if the values are the same. The expression *x*=*y* assigns the value of *y* to *x* and returns the value of *y* as its result.

Closely related to == and *!*= are the *relational* operators. You use these operators to find out whether a value is less than or greater than another value of the same type. The following table shows how to use these operators.

Operator	Meaning	Example	Outcome if age is 42
<	Less than	`age < 21`	false
<=	Less than or equal to	`age <= 18`	false
>	Greater than	`age > 16`	true
>=	Greater than or equal to	`age >= 30`	true

Understanding conditional logical operators

C# also provides two other binary Boolean operators: the logical AND operator, which is represented by the *&&* symbol, and the logical OR operator, which is represented by the || symbol. Collectively, these are known as the *conditional logical operators*. Their purpose is to combine two Boolean expressions or values into a single Boolean result. These operators are similar to the equality and relational operators in that the value of the expressions in which they appear is either true or false, but they differ in that the values on which they operate must also be either true or false.

The outcome of the *&&* operator is *true* if and only if both of the Boolean expressions it's evaluating are *true*. For example, the following statement assigns the value *true* to *validPercentage* if and only if the value of *percent* is greater than or equal to 0 and the value of *percent* is less than or equal to 100:

```
bool validPercentage;
validPercentage = (percent >= 0) && (percent <= 100);
```

Tip A common beginner's error is to try to combine the two tests by naming the *percent* variable only once, like this:

```
percent >= 0 && <= 100 // this statement will not compile
```

Using parentheses helps to avoid this type of mistake and also clarifies the purpose of the expression. For example, compare

```
validPercentage = percent >= 0 && percent <= 100
```

and

```
validPercentage = (percent >= 0) && (percent <= 100)
```

Both expressions return the same value because the precedence of the *&&* operator is less than that of >= and <=. However, the second expression conveys its purpose in a more readable manner.

The outcome of the || operator is *true* if either of the Boolean expressions it evaluates is *true*. You use the || operator to determine whether any one of a combination of Boolean expressions is *true*. For example, the following statement assigns the value *true* to *invalidPercentage* if the value of *percent* is less than 0 or the value of *percent* is greater than 100:

```
bool invalidPercentage;
invalidPercentage = (percent < 0) || (percent > 100);
```

Short circuiting

The *&&* and || operators both exhibit a feature called *short circuiting*. Sometimes, it is not necessary to evaluate both operands when ascertaining the result of a conditional logical expression. For example, if the left operand of the *&&* operator evaluates to *false*, the result of the entire expression must be *false*, regardless of the value of the right operand. Similarly, if the value of the left operand of the || operator evaluates to *true*, the result of the entire expression must be *true*, irrespective of the value of the right operand. In these cases, the *&&* and || operators bypass the evaluation of the right operand. Here are some examples:

```
(percent >= 0) && (percent <= 100)
```

In this expression, if the value of *percent* is less than 0, the Boolean expression on the left side of *&&* evaluates to *false*. This value means that the result of the entire expression must be *false*, and the Boolean expression to the right of the *&&* operator is not evaluated.

```
(percent < 0) || (percent > 100)
```

In this expression, if the value of *percent* is less than 0, the Boolean expression on the left side of || evaluates to *true*. This value means that the result of the entire expression must be *true,* and the Boolean expression to the right of the || operator is not evaluated.

If you carefully design expressions that use the conditional logical operators, you can boost the performance of your code by avoiding unnecessary work. Place simple Boolean expressions that can be evaluated easily on the left side of a conditional logical operator, and put more complex expressions on the right side. In many cases, you will find that the program does not need to evaluate the more complex expressions.

Summarizing operator precedence and associativity

The following table summarizes the precedence and associativity of all the operators you have learned about so far. Operators in the same category have the same precedence. The operators in categories higher up in the table take precedence over operators in categories lower down.

Category	Operators	Description	Associativity
Primary	()	Precedence override	Left
	++	Post-increment	
	--	Post-decrement	

Category	Operators	Description	Associativity
Unary	*!* + - ++ --	Logical NOT Returns the value of the operand unchanged Returns the value of the operand negated Pre-increment Pre-decrement	Left
Multiplicative	* / %	Multiply Divide Division remainder (modulus)	Left
Additive	+ -	Addition Subtraction	Left
Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left
Equality	== *!*=	Equal to Not equal to	Left
Conditional AND	*&&*	Conditional AND	Left
Conditional OR	||	Conditional OR	Left
Assignment	=	Assigns the right-hand operand to the left and returns the value that was assigned	Right

Notice that the *&&* operator and the || operator have a different precedence: *&&* is higher than ||.

Using *if* statements to make decisions

In a method, when you want to choose between executing two different statements depending on the result of a Boolean expression, you can use an *if* statement.

Understanding *if* statement syntax

The syntax of an *if* statement is as follows (*if* and *else* are C# keywords):

```
if ( booleanExpression )
    statement-1;
else
    statement-2;
```

If *booleanExpression* evaluates to *true, statement-1* runs; otherwise, *statement-2* runs. The *else* keyword and the subsequent *statement-2* are optional. If there is no *else* clause and the *booleanExpression* is *false*, execution continues with whatever code follows the *if* statement. Also,

notice that the Boolean expression must be enclosed in parentheses; otherwise, the code will not compile.

For example, here's an *if* statement that increments a variable representing the second hand of a stopwatch. (Minutes are ignored for now.) If the value of the *seconds* variable is 59, it is reset to 0; otherwise, it is incremented by using the ++ operator:

```
int seconds;
...
if (seconds == 59)
    seconds = 0;
else
    seconds++;
```

Boolean expressions only, please!

The expression in an *if* statement must be enclosed in parentheses. Additionally, the expression must be a Boolean expression. In some other languages—notably C and C++—you can write an integer expression, and the compiler will silently convert the integer value to *true* (nonzero) or *false* (0). C# does not support this behavior, and the compiler reports an error if you write such an expression.

If you accidentally specify the assignment operator (=) instead of the equality test operator (==) in an *if* statement, the C# compiler recognizes your mistake and refuses to compile your code, such as in the following example:

```
int seconds;
...
if (seconds = 59)  // compile-time error
...
if (seconds == 59) // ok
```

Accidental assignments were another common source of bugs in C and C++ programs, which would silently convert the value assigned (59) to a Boolean expression (with anything nonzero considered to be true), with the result being that the code following the *if* statement would be performed every time.

Incidentally, you can use a Boolean variable as the expression for an *if* statement, although it must still be enclosed in parentheses, as shown in this example:

```
bool inWord;
...
if (inWord == true) // ok, but not commonly used
...
if (inWord)         // more common and considered better style
```

Using blocks to group statements

Notice that the syntax of the *if* statement shown earlier specifies a single statement after the *if (booleanExpression)* and a single statement after the *else* keyword. Sometimes, you'll want to perform more than one statement when a Boolean expression is true. You could group the statements inside a new method and then call the new method, but a simpler solution is to group the statements inside a *block*. A block is simply a sequence of statements grouped between an opening brace and a closing brace.

In the following example, two statements that reset the *seconds* variable to 0 and increment the *minutes* variable are grouped inside a block, and the entire block executes if the value of *seconds* is equal to 59:

```
int seconds = 0;
int minutes = 0;
...
if (seconds == 59)
{
    seconds = 0;
    minutes++;
}
else
{
    seconds++;
}
```

Important If you omit the braces, the C# compiler associates only the first statement (*seconds = 0;*) with the *if* statement. The subsequent statement (*minutes++;*) will not be recognized by the compiler as part of the *if* statement when the program is compiled. Furthermore, when the compiler reaches the *else* keyword, it will not associate it with the previous *if* statement; instead, it will report a syntax error. Therefore, it is good practice to always define the statements for each branch of an *if* statement within a block, even if a block consists of only a single statement. It might save you some grief later if you want to add additional code.

A block also starts a new scope. You can define variables inside a block, but they will disappear at the end of the block. The following code fragment illustrates this point:

```
if (...)
{
    int myVar = 0;
    ... // myVar can be used here
} // myVar disappears here
else
{
    // myVar cannot be used here
    ...
}
// myVar cannot be used here
```

Cascading *if* statements

You can nest *if* statements inside other *if* statements. In this way, you can chain together a sequence of Boolean expressions, which are tested one after the other until one of them evaluates to *true*. In the following example, if the value of *day* is 0, the first test evaluates to *true* and *dayName* is assigned the string *"Sunday"*. If the value of *day* is not 0, the first test fails and control passes to the *else* clause, which runs the second *if* statement and compares the value of *day* with 1. The second *if* statement executes only if the first test is *false*. Similarly, the third *if* statement executes only if the first and second tests are *false*.

```
if (day == 0)
{
    dayName = "Sunday";
}
else if (day == 1)
{
    dayName = "Monday";
}
else if (day == 2)
{
    dayName = "Tuesday";
}
else if (day == 3)
{
    dayName = "Wednesday";
}
else if (day == 4)
{
    dayName = "Thursday";
}
else if (day == 5)
{
    dayName = "Friday";
}
else if (day == 6)
{
    dayName = "Saturday";
}
else
{
    dayName = "unknown";
}
```

In the following exercise, you'll write a method that uses a cascading *if* statement to compare two dates.

Write *if* statements

1. Start Microsoft Visual Studio 2015 if it is not already running.
2. Open the Selection project, which is located in the \Microsoft Press\VCSBS\Chapter 4\Selection folder in your Documents folder.

3. On the Debug menu, click Start Debugging.

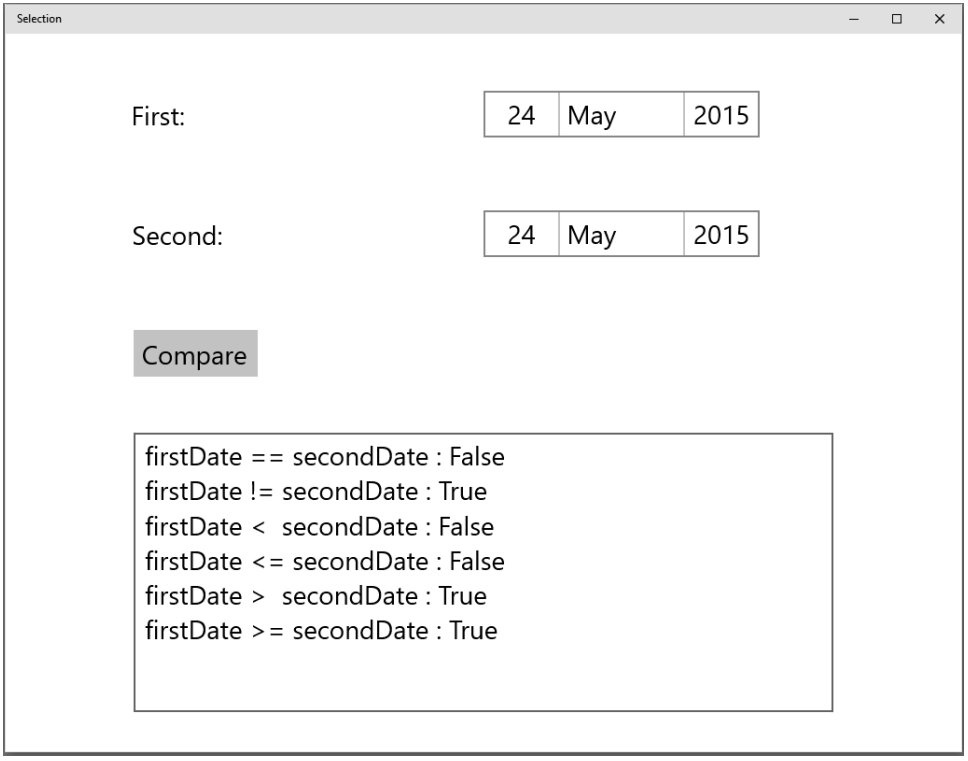
   Visual Studio 2015 builds and runs the application. The form displays two *DatePicker* controls, called *firstDate* and *secondDate*. Both controls display the current date.

4. Click Compare.

   The following text appears in the text box in the lower half of the window:

```
firstDate == secondDate : False
firstDate != secondDate : True
firstDate <  secondDate : False
firstDate <= secondDate : False
firstDate >  secondDate : True
firstDate >= secondDate : True
```

   The Boolean expression, *firstDate == secondDate*, should be *true* because both *firstDate* and *secondDate* are set to the current date. In fact, only the less-than operator and the greater-than-or-equal-to operator seem to be working correctly. The following image shows the application running.



5. Return to Visual Studio 2015. On the Debug menu, click Stop Debugging.

6. Display the code for the MainPage.xaml.cs file in the Code and Text Editor window.

7. Locate the *compareClick* method, which should look like this:

```
private void compareClick(object sender, RoutedEventArgs e)
{
    int diff = dateCompare(firstDate.Date.LocalDateTime, secondDate.Date.LocalDateTime);
    info.Text = "";
    show("firstDate == secondDate", diff == 0);
    show("firstDate != secondDate", diff != 0);
    show("firstDate < secondDate", diff < 0);
    show("firstDate <= secondDate", diff <= 0);
    show("firstDate > secondDate", diff > 0);
    show("firstDate >= secondDate", diff >= 0);
}
```

This method runs whenever the user clicks the Compare button on the form. The expressions *firstDate.Date.LocalDateTime* and *secondDate.Date.LocalDateTime* hold *DateTime* values; they represent the dates displayed in the *firstDate* and *secondDate* controls on the form elsewhere in the application. The *DateTime* data type is just another data type, like *int* or *float*, except that it contains subelements with which you can access the individual pieces of a date, such as the year, month, or day.

The *compareClick* method passes the two *DateTime* values to the *dateCompare* method. The purpose of this method is to compare dates and return the *int* value 0 if they are the same, –1 if the first date is less than the second, and +1 if the first date is greater than the second. A date is considered greater than another date if it comes after it chronologically. You will examine the *dateCompare* method in the next step.

The *show* method displays the results of the comparison in the *info* text box control in the lower half of the form.

8. Locate the *dateCompare* method, which should look like this:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    // TO DO
    return 42;
}
```

This method currently returns the same value whenever it is called—rather than 0, –1, or +1—regardless of the values of its parameters. This explains why the application is not working as expected. You need to implement the logic in this method to compare two dates correctly.

9. Remove the *// TO DO* comment and the *return* statement from the *dateCompare* method.

10. Add the following statements shown in bold to the body of the *dateCompare* method:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    int result = 0;

    if (leftHandSide.Year < rightHandSide.Year)
    {
        result = -1;
    }
    else if (leftHandSide.Year > rightHandSide.Year)
```

```
    {
        result = 1;
    }
}
```

Note Don't try to build the application yet. The *dateCompare* method is not complete and the build will fail.

If the expression *leftHandSide.Year* < *rightHandSide.Year* is *true*, the date in *leftHandSide* must be earlier than the date in *rightHandSide*, so the program sets the *result* variable to *–1*. Otherwise, if the expression *leftHandSide.Year* > *rightHandSide.Year* is *true*, the date in *leftHandSide* must be later than the date in *rightHandSide*, and the program sets the *result* variable to *1*.

If the expression *leftHandSide.Year* < *rightHandSide.Year* is *false* and the expression *leftHandSide.Year* > *rightHandSide.Year* is also *false*, the *Year* property of both dates must be the same, so the program needs to compare the months in each date.

11. Add the following statements shown in bold to the body of the *dateCompare* method. Type them after the code you entered in the preceding step:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    ...
    else if (leftHandSide.Month < rightHandSide.Month)
    {
        result = -1;
    }
    else if (leftHandSide.Month > rightHandSide.Month)
    {
        result = 1;
    }
}
```

These statements compare months following a logic similar to that used to compare years in the preceding step.

If the expression *leftHandSide.Month* < *rightHandSide.Month* is *false* and the expression *leftHandSide.Month* > *rightHandSide.Month* is also *false*, the *Month* property of both dates must be the same, so the program finally needs to compare the days in each date.

12. Add the following statements to the body of the *dateCompare* method after the code you entered in the preceding two steps:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
    ...
    else if (leftHandSide.Day < rightHandSide.Day)
    {
        result = -1;
    }
```

```
    else if (leftHandSide.Day > rightHandSide.Day)
    {
        result = 1;
    }
    else
    {
       result = 0;
    }

    return result;
}
```

You should recognize the pattern in this logic by now.

If *leftHandSide.Day* < *rightHandSide.Day* and *leftHandSide.Day* > *rightHandSide.Day* both are *false*, the value in the *Day* properties in both variables must be the same. The *Month* values and the *Year* values must also be identical, respectively, for the program logic to have reached this point, so the two dates must be the same, and the program sets the value of *result* to *0*.

The final statement returns the value stored in the *result* variable.

13. On the Debug menu, click Start Debugging.

    The application is rebuilt and runs.

14. Click Compare.

    The following text appears in the text box:

```
firstDate == secondDate : True
firstDate != secondDate : False
firstDate <  secondDate: False
firstDate <= secondDate: True
firstDate >  secondDate: False
firstDate >= secondDate: True
```

    These are the correct results for identical dates.

15. Use the *DatePicker* controls to select a later date for the second date and then click Compare.

    The following text appears in the text box:

```
firstDate == secondDate: False
firstDate != secondDate: True
firstDate <  secondDate: True
firstDate <= secondDate: True
firstDate >  secondDate: False
firstDate >= secondDate: False
```

    Again, these are the correct results when the first date is earlier than the second date.

16. Test some other dates, and verify that the results are as you would expect. Return to Visual Studio 2015 and stop debugging when you have finished.

Comparing dates in real-world applications

Now that you have seen how to use a rather long and complicated series of *if* and *else* statements, I should mention that this is not the technique you would employ to compare dates in a real-world application. If you look at the *dateCompare* method from the preceding exercise, you will see that the two parameters, *leftHandSide* and *rightHandSide*, are *DateTime* values. The logic you have written compares only the date part of these parameters, but they also contain a time element that you have not considered (or displayed). For two *DateTime* values to be considered equal, they should have not only the same date but also the same time. Comparing dates and times is such a common operation that the *DateTime* type actually has a built-in method called *Compare* for doing just that: it takes two *DateTime* arguments and compares them, returning a value indicating whether the first argument is less than the second, in which case the result will be negative; whether the first argument is greater than the second, in which case the result will be positive; or whether both arguments represent the same date and time, in which case the result will be 0.

Using *switch* statements

Sometimes when you write a cascading *if* statement, each of the *if* statements look similar because they all evaluate an identical expression. The only difference is that each *if* compares the result of the expression with a different value. For example, consider the following block of code that uses an *if* statement to examine the value in the *day* variable and work out which day of the week it is:

```
if (day == 0)
{
    dayName = "Sunday";
}
else if (day == 1)
{
    dayName = "Monday";
}
else if (day == 2)
{
    dayName = "Tuesday";
}
else if (day == 3)
{
    ...
}
else
{
    dayName = "Unknown";
}
```

Often in these situations, you can rewrite the cascading *if* statement as a *switch* statement to make your program more efficient and more readable.

Understanding *switch* statement syntax

The syntax of a *switch* statement is as follows (*switch*, *case*, and *default* are keywords):

```
switch ( controllingExpression )
{
    case constantExpression :
        statements
        break;
    case constantExpression :
        statements
        break;
    ...
    default :
        statements
        break;
}
```

The *controllingExpression*, which must be enclosed in parentheses, is evaluated once. Control then jumps to the block of code identified by the *constantExpression* whose value is equal to the result of the *controllingExpression*. (The *constantExpression* identifier is also called a *case label*.) Execution runs as far as the *break* statement, at which point the *switch* statement finishes and the program continues at the first statement that follows the closing brace of the *switch* statement. If none of the *constantExpression* values is equal to the value of the *controllingExpression*, the statements below the optional *default* label run.

Note Each *constantExpression* value must be unique so that the *controllingExpression* will match only one of them. If the value of the *controllingExpression* does not match any *constantExpression* value and there is no *default* label, program execution continues with the first statement that follows the closing brace of the *switch* statement.

So, you can rewrite the previous cascading *if* statement as the following *switch* statement:

```
switch (day)
{
    case 0 :
        dayName = "Sunday";
        break;
    case 1 :
        dayName = "Monday";
        break;
    case 2 :
        dayName = "Tuesday";
        break;
    ...
    default :
        dayName = "Unknown";
        break;
}
```

Following the *switch* statement rules

The *switch* statement is very useful, but unfortunately, you can't always use it when you might like to. Any *switch* statement you write must adhere to the following rules:

- You can use *switch* only on certain data types, such as *int*, *char*, or *string*. With any other types (including *float* and *double*), you must use an *if* statement.
- The *case* labels must be constant expressions, such as 42 if the *switch* data type is an *int*, '4' if the *switch* data type is a *char*, or "42" if the *switch* data type is a *string*. If you need to calculate your *case* label values at run time, you must use an *if* statement.
- The *case* labels must be unique expressions. In other words, two *case* labels cannot have the same value.
- You can specify that you want to run the same statements for more than one value by providing a list of *case* labels and no intervening statements, in which case the code for the final label in the list is executed for all cases in that list. However, if a label has one or more associated statements, execution cannot fall through to subsequent labels; in this case, the compiler generates an error. The following code fragment illustrates these points:

```
switch (trumps)
{
    case Hearts :
    case Diamonds :       // Fall-through allowed - no code between labels
        color = "Red";    // Code executed for Hearts and Diamonds
        break;
    case Clubs :
        color = "Black";
    case Spades :         // Error - code between labels
        color = "Black";
        break;
}
```

Note The *break* statement is the most common way to stop fall-through, but you can also use a *return* statement to exit from the method containing the *switch* statement or a *throw* statement to generate an exception and abort the *switch* statement. The *throw* statement is described in Chapter 6, "Managing errors and exceptions."

switch fall-through rules

Because you cannot accidentally fall through from one *case* label to the next if there is any intervening code, you can freely rearrange the sections of a *switch* statement without affecting its meaning (including the *default* label, which by convention is usually—but does not have to be—placed as the last label).

C and C++ programmers should note that the *break* statement is mandatory for every case

in a *switch* statement (even the default case). This requirement is a good thing—it is common in C or C++ programs to forget the *break* statement, allowing execution to fall through to the next label and leading to bugs that are difficult to spot.

If you really want to, you can mimic C/C++ fall-through in C# by using a *goto* statement to go to the following *case* or *default* label. Using *goto* in general is not recommended, though, and this book does not show you how to do it.

In the following exercise, you will complete a program that reads the characters of a string and maps each character to its XML representation. For example, the left angle bracket character (<) has a special meaning in XML (it's used to form elements). If you have data that contains this character, it must be translated into the text entity *<* so that an XML processor knows that it is data and not part of an XML instruction. Similar rules apply to the right angle bracket (>), ampersand (*&*), single quotation mark ('), and double quotation mark (") characters. You will write a *switch* statement that tests the value of the character and traps the special XML characters as *case* labels.

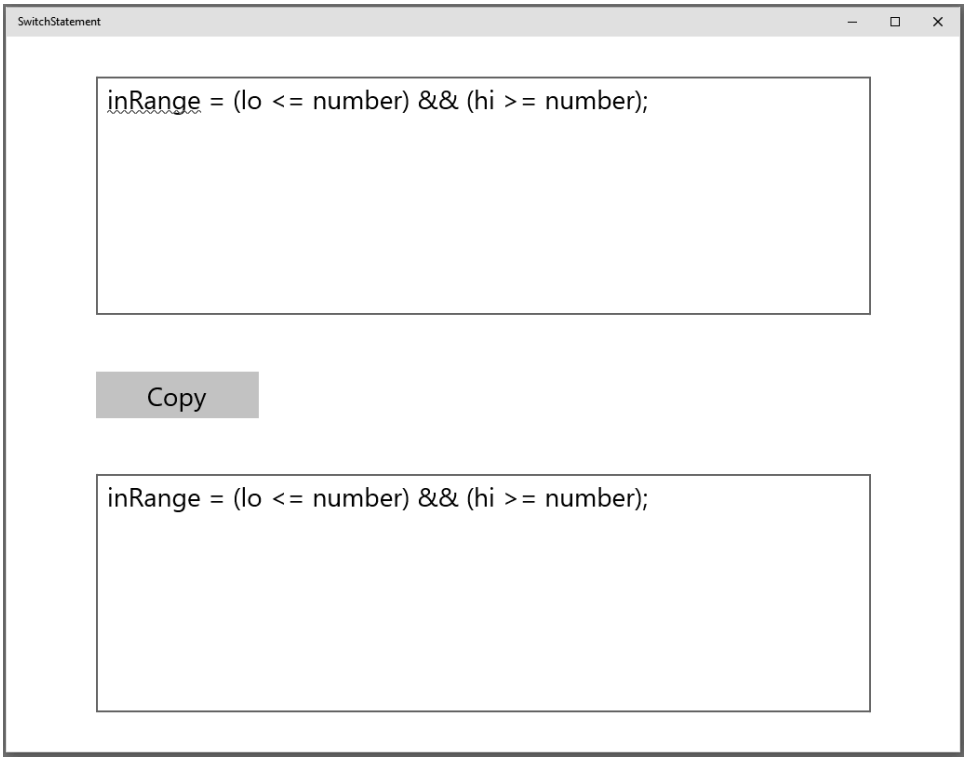
Write *switch* statements

1. Start Visual Studio 2015 if it is not already running.
2. Open the SwitchStatement project, which is located in the \Microsoft Press\VCSBS\Chapter 4\SwitchStatement folder in your Documents folder.
3. On the Debug menu, click Start Debugging.

   Visual Studio 2015 builds and runs the application. The application displays a form containing two text boxes separated by a Copy button.
4. Type the following sample text into the upper text box:

   inRange = (lo <= number) && (hi >= number);
5. Click Copy.

   The statement is copied verbatim into the lower text box, and no translation of the <, &, or > characters occurs, as shown in the following screen shot.



6. Return to Visual Studio 2015 and stop debugging.

7. Display the code for MainPage.xaml.cs in the Code and Text Editor window and locate the *copyOne* method.

   The *copyOne* method copies the character specified as its input parameter to the end of the text displayed in the lower text box. At the moment, *copyOne* contains a *switch* statement with a single *default* action. In the following few steps, you will modify this *switch* statement to convert characters that are significant in XML to their XML mapping. For example, the < character will be converted to the string *<*.

8. Add the following statements shown in bold to the *switch* statement after the opening brace for the statement and directly before the *default* label:

```
switch (current)
{
    case '<' :
        target.Text += "&lt;";
        break;
    default:
        target.Text += current;
        break;
}
```

If the current character being copied is a left angle bracket (<), the preceding code appends the string *"<"* to the text being output in its place.

9. Add the following statements to the *switch* statement after the *break* statement you have just added and above the *default* label:

```
case '>' :
    target.Text += "&gt;";
    break;
case '&' :
    target.Text += "&amp;";
    break;
case '\"' :
    target.Text += "&#34;";
    break;
case '\'' :
    target.Text += "&#39;";
    break;
```

Note The single quotation mark (') and double quotation mark (") have a special meaning in C#—they are used to delimit character and string constants. The backslash (\) in the final two *case* labels is an escape character that causes the C# compiler to treat these characters as literals rather than as delimiters.

10. On the Debug menu, click Start Debugging.

11. Type the following text into the upper text box:

    inRange = (lo <= number) && (hi >= number);

12. Click Copy.

    The statement is copied into the lower text box. This time, each character undergoes the XML mapping implemented in the *switch* statement. The target text box displays the following text:

    inRange = (lo <= number) && (hi >= number);

13. Experiment with other strings, and verify that all special characters (<, >, &, " , and ') are handled correctly.

14. Return to Visual Studio and stop debugging.

Summary

In this chapter, you learned about Boolean expressions and variables. You saw how to use Boolean expressions with the *if* and *switch* statements to make decisions in your programs, and you combined Boolean expressions by using the Boolean operators.

- If you want to continue to the next chapter, keep Visual Studio 2015 running and turn to Chapter 5, "Using compound assignment and iteration statements."
- If you want to exit Visual Studio 2015 now, on the File menu, click Exit. If you see a Save dialog box, click Yes and save the project.

Quick reference

To	Do this	Example
Determine whether two values are equivalent	Use the == operator or the != operator.	`answer == 42`
Compare the value of two expressions	Use the <, <=, >, or >= operator.	`age >= 21`
Declare a Boolean variable	Use the *bool* keyword as the type of the variable.	`bool inRange;`
Create a Boolean expression that is true only if two conditions are both true	Use the && operator.	`inRange = (lo <= number)` `  && (number <= hi);`
Create a Boolean expression that is true if either of two conditions is true	Use the || operator.	`outOfRange = (number < lo)` `  || (hi < number);`
Run a statement if a condition is true	Use an *if* statement.	`if (inRange)` `  process();`

Run more than one statement if a condition is true	Use an *if* statement and a block.	`if (seconds == 59)`
		`{`
		`  seconds = 0;`
		`  minutes++;`
		`}`

Associate different statements with different values of a controlling expression	Use a *switch* statement.	`switch (current)`
		`{`
		`  case 0:`
		`  ...`
		`  break;`
		
		`  case 1:`
		`  ...`
		`  break;`
		`  default :`
		`  ...`
		`  break;`
		`}`

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

About the author

John Sharp is a principal technologist for CM Group Ltd, a software development and consultancy company in the United Kingdom. He is well versed as a software consultant, developer, author, and trainer, with nearly 30 years of experience, ranging from Pascal programming on CP/M and C/Oracle application development on various flavors of UNIX to the design of C# and JavaScript distributed applications and development on Windows 10 and Microsoft Azure. He is an expert on building applications with the Microsoft .NET Framework and is also the author of *Windows Communication Foundation 4 Step By Step* (Microsoft Press).