Microsoft®

Microsoft® Visual C#® 2012

John Sharp

ebook+exercises

Step by Step

Microsoft Visual C# 2012 Step by Step

Your hands-on, step-by-step guide to the fundamentals of Visual C# development.

Teach yourself how to build applications with Microsoft Visual C# 2012 and Visual Studio® 2012—one step at a time. Ideal for those with fundamental programming skills, this tutorial provides practical, learn-by-doing exercises for mastering core C# language features and creating working applications and components for Windows®.

Discover how to:

- Work with variables, statements, operators, and methods
- Write robust code that can catch and handle exceptions
- Respond to user input and gestures
- Handle events arising from multiple sources
- Manipulate data sets using arrays and collections
- Establish new data types by using classes, interfaces, and structures
- Use LINQ expressions to enumerate data
- Optimize processing with tasks and asynchronous operations
- Build your first Windows Store app

Your *Step by Step* digital content includes:

- Downloadable practice files
  See http://go.microsoft.com/FWLink/?Linkid=273785
- Fully searchable online edition of this book—with unlimited access on the web. *Free online account required; see inside back*

microsoft.com/mspress

ISBN: 978-0-7356-6801-0

9 780735 668010 90000

U.S.A. $44.99
Canada $47.99
[Recommended]

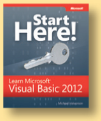
Programming/Microsoft Visual C#

About the Author

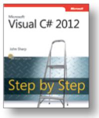
John Sharp is an expert on developing applications with the Microsoft .NET Framework and interoperability issues. He has coauthored guides for the Microsoft Patterns and Practices group, and is the author of *Microsoft Visual C# 2010 Step by Step* and *Microsoft Windows Communication Foundation Step by Step.*

DEVELOPER ROADMAP

Start Here!
- Beginner-level instruction
- Easy-to-follow explanations and examples
- Exercises to build your first projects

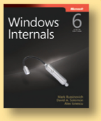


Step by Step
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook



Developer Reference
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples

Focused Topics
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples



Microsoft®

Microsoft® Visual C#® 2012 Step by Step

John Sharp

ISBN: 978-0-7356-6801-0

2 3 4 5 6 7 8 9 10 LSI 8 7 6 5 4 3

Printed and bound in the United States of America.

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

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation,
nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Development Editor: Russell Jones

Production Editor: Rachel Steely

Editorial Production: Zyg Group, LLC

Technical Reviewer: John Mueller

Copyeditor: Nicole Flores

Indexer: BIM Indexing Services

Cover Design: Twist Creative

Cover Composition: Zyg Group, LLC

Illustrator: Rebecca Demarest

[2013-05-24]

I dedicate this book to Diana, my wife and fellow Warwickshire supporter, for keeping me sane and giving me the perfect excuse to spend time watching cricket.

—John Sharp

Contents at a Glance

PART IV BUILDING PROFESSIONAL WINDOWS 8 APPLICATIONS WITH C#

Contents

PART IV BUILDING PROFESSIONAL WINDOWS 8 APPLICATIONS WITH C#

Introduction

Microsoft Visual C# is a powerful but simple language aimed primarily at developers creating applications by using the Microsoft .NET Framework. It inherits many of the best features of C++ and Microsoft Visual Basic, but few of the inconsistencies and anachronisms, resulting in a cleaner and more logical language. C# 1.0 made its public debut in 2001. The advent of C# 2.0 with Visual Studio 2005 saw several important new features 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, released in 2010, 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 in the .NET Framework released concurrently with C# 4.0 was 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 quickly and easily. C# 5.0 adds native support for asynchronous task-based processing through the *async* method modifier and the *await* operator.

Another key event for Microsoft has been the launch of Windows 8. This new version of Windows supports highly interactive applications that can share data and collaborate with each other as well as connect to services running in the cloud. The development environment provided by Microsoft Visual Studio 2012 makes all these powerful features easy to use, and the many new wizards and enhancements included in Visual Studio 2012 can greatly improve your productivity as a developer. The combination of Visual Studio 2012, Windows 8, and C# 5.0 provides a comprehensive platform and toolset for building the next generation of powerful, intuitive, and portable applications. However, even if you are not using Windows 8, Visual Studio 2012 and C# 5.0 have much to offer, and they form an invaluable partnership for helping you to build great solutions.

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 2012 and the .NET Framework version 4.5. 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 Windows Presentation Foundation (WPF) applications that can run on both Windows 7 and Windows 8.

You can build and run C# 5.0 applications on Windows 7 and Windows 8, although the user interfaces provided by these two operating systems have some significant differences. Consequently, Parts I to III of this book provide exercises and worked examples that will run in both environments. Part IV focuses on the application development model used by Windows 8, and the material in this section provides an introduction to building interactive applications for this new platform.

Who Should Not Read This Book

This book is aimed at developers new to C#, and 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 Workflow Foundation. If you require more information on any of these items, you might consider reading some of the other titles in the Step by Step for Developers series available from Microsoft Press, such as *Microsoft ASP.NET 4 Step by Step*, *Microsoft ADO.NET 4 Step by Step*, and *Microsoft Windows Communication Foundation 4 Step by Step*.

Organization of This Book

This book is divided into four sections:

- Part I, "Introducing Microsoft Visual C# and Microsoft Visual Studio 2012," 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 by using 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 Professional Window 8 Applications with C#," describes the Windows 8 programming model, and how you can use C# to build interactive applications for this new model.

Note Although Part IV is aimed at Windows 8, many of the concepts described in Chapters 23 and 24 are applicable to Windows 7 applications.

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." Skim the first five chapters to get an overview of C# and Visual Studio 2012, and then concentrate on Chapters 6 through 22. 2. 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 2012, and then concentrate on Chapters 7 through 22. 3. For information about building scalable Windows 8 applications, read Part IV.
Switching from Visual Basic 6 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 Windows 8 applications, read Part IV. 4. Read the Quick Reference sections at the end of the chapters for information about specific C# and Visual Studio 2012 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 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 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.
- A vertical bar between two or more menu items (for example, File | Close) means that you should select the first menu or menu item, then the next, and so on.

System Requirements

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

- Windows 7 (x86 and x64), Windows 8 (x86 and x64), Windows Server 2008 R2 (x64), Windows Server 2012 (x64).

Note Visual Studio 2012 is also available for Windows Vista, Windows XP, and Windows Server 2003. However, the exercises and code in this book have not been tested on these platforms.

- Visual Studio 2012 (any edition except Visual Studio Express for Windows 8).

Note You can use Visual Studio Express 2012 for Windows Desktop, but you can only perform the Windows 7 version of the exercises in this book by using this software. You cannot use this software to perform the exercises in Part IV of this book.

- 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; If you are using Windows 8, a resolution of 1366 × 768 or greater is recommended.
- 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 2012.

Code Samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects, in both their pre-exercise and postexercise formats, can be downloaded from the following page:

http://www.microsoftpressstore.com/title/9780735668010

Follow the instructions to download the 9780735668010_files.zip file.

> **Note** In addition to the code samples, your system should have Visual Studio 2012 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. Move to your Documents folder and create a new folder called Microsoft Press.

2. Copy the file that you downloaded from the book's website into the Microsoft Press folder.

3. Unzip the file and allow it to create the folder Visual CSharp Step By Step.

> **Note** If the license agreement doesn’t appear, you can access it from the same webpage from which you downloaded the <yoursamplefile.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.

For those of you who like to know all the details, here’s a list of the code sample Visual Studio 2012 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 code samples provide versions of the code for Window 7 and Windows 8, and the exercise instructions call out any differences in the tasks that you need to perform or the code that you need to write for these two operating systems. The completed projects for each chapter are stored in folders with the suffix "- Complete".

> **Note** If you are using Windows Server 2008 R2, follow the instructions for Windows 7. If you are using Windows Server 2012, follow the instructions for Windows 8.

Project	Description
Chapter 1	
TextHello	This project gets you started. It steps through the creation of a simple program that displays a text-based greeting.
WPFHello	This project displays the greeting in a window by using Windows Presentation Foundation (WPF).
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 (+ – * / %).

Project	Description
Chapter 3	
Methods	In this project, you'll reexamine the code in the previous 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 2012 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.
Chapter 6	
MathsOperators	This project revisits the MathsOperators project from Chapter 2, "Working with Variables, Operators, and Expressions," 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.

Project	Description
Chapter 11	
ParamsArrays	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 Using Interfaces	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 Abstract Classes	This project extends the Drawing Using Interfaces project to factor common functionality for shape objects into abstract classes.
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 Using Abstract Classes 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 *typesafe* structure that can contain elements of any type.
BuildTree	This project demonstrates how to use generics to implement a *typesafe* 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.

Project	Description
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.
Delegates With Event	This project shows 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.
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 WPF form. It uses a single thread to perform the calculations.
GraphDemo With Tasks	This version of the GraphDemo project creates multiple tasks to perform the calculations for the graph in parallel.
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.

Project	Description
Chapter 25	
Customers Without Scalable UI	This project uses the default *Grid* control to lay out the user interface for the Adventure Works Customers application. The user interface uses absolute positioning for the controls and does not scale to different screen resolutions and form factors.
Customers With Scalable UI	This project uses nested *Grid* controls with row and column definitions to enable relative positioning of controls. This version of the user interface scales to different screen resolutions and form factors, but it does not adapt well to Snapped view.
Customers With Adaptive UI	This project extends the version with the scalable user interface. It uses the Visual State Manager to detect whether the application is running in Snapped view, and it changes the layout of the controls accordingly.
Customers With Styles	This version of the Customers project uses XAML styling to change the font and background image displayed by the application.
Chapter 26	
DataBinding	This project 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 to enable the user interface to 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.
Search	This project implements the Windows 8 Search contract. A user can search for customers by first name or last name.
Chapter 27	
Data Service	This solution includes a web application that provides a WCF Data Service that the Customers application uses to retrieve customer data from a SQL Server database. The WCF Data Service uses an entity model created by using the Entity Framework to access the database.
Updatable ViewModel	The Customers project in this solution contains an extended ViewModel with commands that enable the user interface to insert and update customer information by using the WCF Data Service.

Acknowledgments

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

First, Russell Jones, with whom I have been communicating for the better part of a year over what we should include in this edition and how we should structure the book.

He has been incredibly patient while I pondered how to address the chapters in the final section of this book.

Next, Mike Sumsion and Paul Barnes, my esteemed colleagues at Content Master, who performed sterling work reviewing the material for each chapter, testing my code, and pointing out the numerous mistakes that I had made! I think I have now caught them all, but of course any errors that remain are entirely my responsibility.

Also, John Mueller, who has done a remarkable job in performing a technical review of the content. His writing experience and understanding of the technologies covered herein have been extremely helpful, and this book has been enriched by his efforts.

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 thank Rachel Steely and Nicole LeClerc for correcting my grammar, fixing my spelling, and generally making my material much easier to understand.

Finally, I would like to thank my wife Diana, for the copious cups of tea and numerous sandwiches she prepared for me while I had my head down writing. She smoothed my furrowed brow many times while I was fathoming out how to make the code in the exercises work.

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:

http://www.microsoftpressstore.com/title/9780735668010

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://www.microsoft.com/learning/booksurvey

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*

PART I

Introducing Microsoft Visual C# and Microsoft Visual Studio 2012

Microsoft Visual C# is Microsoft's powerful component-oriented language. C# plays an important role in the architecture of the Microsoft .NET Framework, and some people have compared it to the role that C played in the development of UNIX. If you already know a language such as C, C++, or Java, you'll find the syntax of C# reassuringly familiar. If you are used to programming in other languages, you should soon be able to pick up the syntax and feel of C#; you just need to learn to put the braces and semicolons in the right place.

In Part I, you'll learn the fundamentals of C#. You'll discover how to declare variables and how to use arithmetic operators such as the plus sign (+) and minus sign (–) to manipulate the values in variables. You'll see how to write methods and pass arguments to methods. You'll also learn how to use selection statements such as *if* and iteration statements such as *while*. Finally, you'll understand how C# uses exceptions to handle errors in a graceful, easy-to-use manner. These topics form the core of C#, and from this solid foundation, you'll progress to more advanced features in Part II through Part IV.

CHAPTER 1

Welcome to C#

After completing this chapter, you will be able to

- Use the Microsoft Visual Studio 2012 programming environment.
- Create a C# console application.
- Explain the purpose of namespaces.
- Create a simple graphical C# application.

This chapter provides an introduction to Visual Studio 2012, the programming environment and toolset designed to help you build applications for Microsoft Windows. Visual Studio 2012 is the ideal tool for writing C# code, and it provides many features that you will learn about as you progress through this book. In this chapter, you will use Visual Studio 2012 to build some simple C# applications and get started on the path to building highly functional solutions for Windows.

Beginning Programming with the Visual Studio 2012 Environment

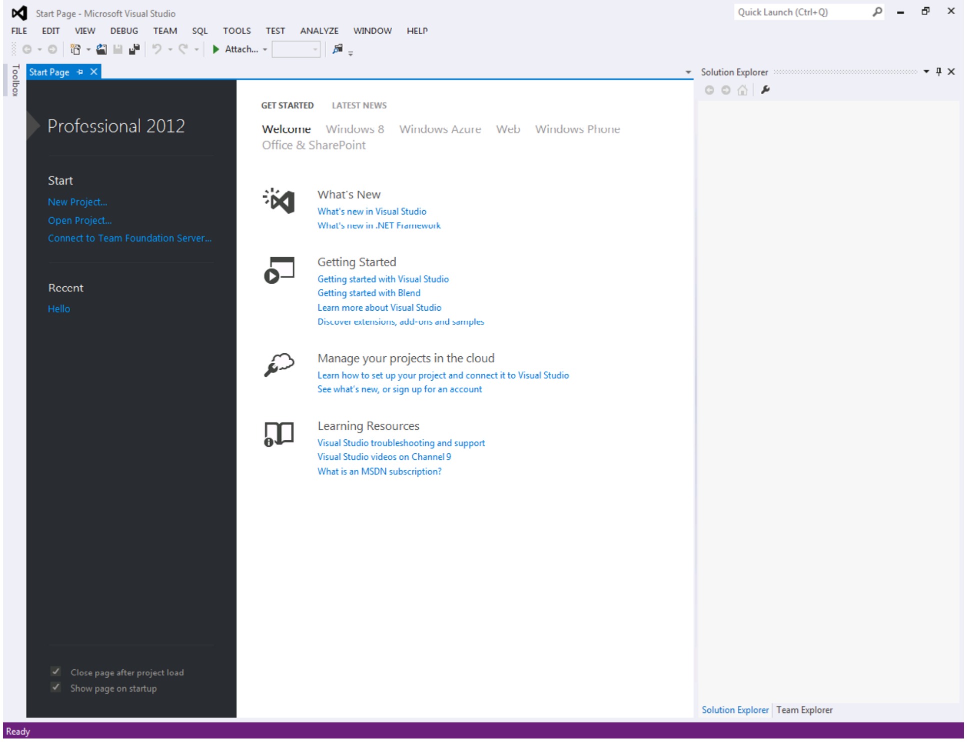
Visual Studio 2012 is a tool-rich programming environment containing the functionality that you need to create large or small C# projects running on Windows 7 and Windows 8. You can even construct projects that seamlessly combine modules written in different programming languages such as C++, Visual Basic, and F#. In the first exercise, you will open the Visual Studio 2012 programming environment and learn how to create a console application.

Note A console application is an application that runs in a command prompt window rather than providing a graphical user interface (GUI).

Create a console application in Visual Studio 2012

- If you are using Windows 8, on the Start screen click the Visual Studio 2012 tile.

  Visual Studio 2012 starts and displays the Start page, like this (your Start page may be different, depending on the edition of Visual Studio 2012 you are using):



Professional 2012

Start

New Project...

Open Project...

Connect to Team Foundation Server...

Recent

Hello

☑ Close page after project load

☑ Show page on startup

GET STARTED LATEST NEWS

Welcome Windows 8 Windows Azure Web Windows Phone Office & SharePoint

What's New

What's new in Visual Studio

What's new in .NET Framework

Getting Started

Getting started with Visual Studio

Getting started with Blend

Learn more about Visual Studio

Discover extensions, add-ons and samples

Manage your projects in the cloud

Learn how to set up your project and connect it to Visual Studio

See what's new, or sign up for an account

Learning Resources

Visual Studio troubleshooting and support

Visual Studio videos on Channel 9

What is an MSDN subscription?

Note If this is the first time you have run Visual Studio 2012, you might see a dialog box prompting you to choose your default development environment settings. Visual Studio 2012 can tailor itself according to your preferred development language. The various dialog boxes and tools in the integrated development environment (IDE) will have their default selections set for the language you choose. Select Visual C# Development Settings from the list, and then click the Start Visual Studio button. After a short delay, the Visual Studio 2012 IDE appears.

- If you are using Windows 7, perform the following operations to start Visual Studio 2012:

  a. On the Microsoft Windows taskbar, click the Start button, point to All Programs, and then click the Microsoft Visual Studio 2012 program group.

  b. In the Microsoft Visual Studio 2012 program group, click Visual Studio 2012.

  Visual Studio 2012 starts and displays the Start page.

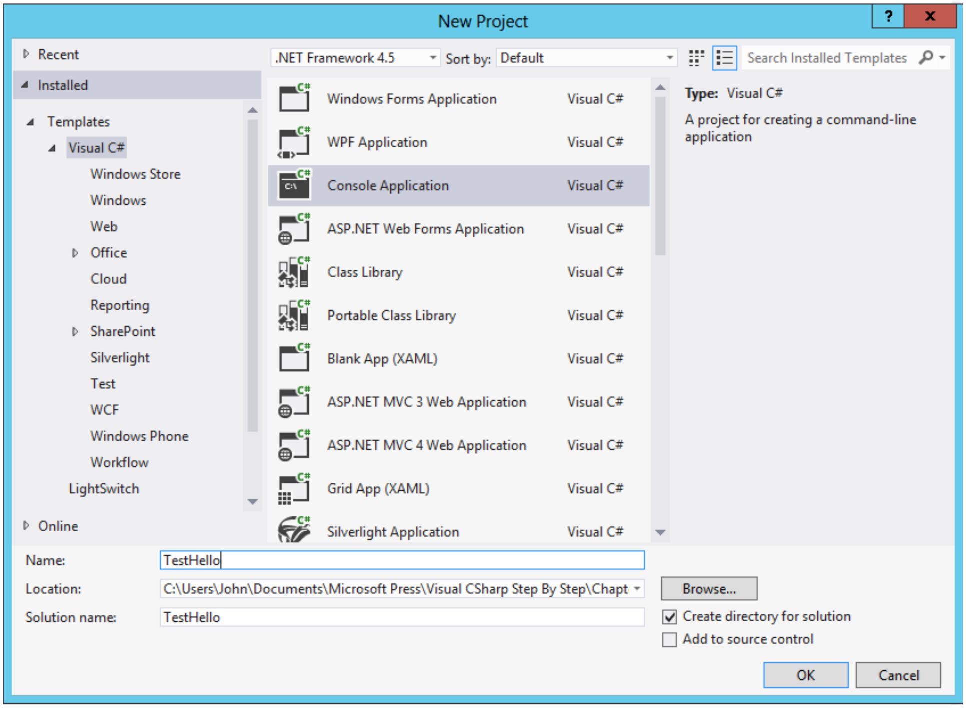
Note To avoid repetition, throughout this book, I will simply state "Start Visual Studio" when you need to open Visual Studio 2012, regardless of the operating system you are using.

- Perform the following tasks to create a new console application:

  a. On the FILE menu, point to New, and then click Project.

  The New Project dialog box opens. This dialog box lists the templates that you can use as a starting point for building an application. The dialog box categorizes templates according to the programming language you are using and the type of application.

  b. In the left pane, under Templates, click Visual C#. In the middle pane, verify that the combo box at the top of the pane displays the text .NET Framework 4.5, and then click the Console Application icon.



.NET Framework 4.5 Sort by: Default

Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
ASP.NET Web Forms Application	Visual C#
Class Library	Visual C#
Portable Class Library	Visual C#
Blank App (XAML)	Visual C#
ASP.NET MVC 3 Web Application	Visual C#
ASP.NET MVC 4 Web Application	Visual C#
Grid App (XAML)	Visual C#
Silverlight Application	Visual C#

Type: Visual C#

A project for creating a command-line application

Name: TestHello

Location: C:\Users\John\Documents\Microsoft Press\Visual CSharp Step By Step\Chapt

Solution name: TestHello

☑ Create directory for solution

□ Add to source control

  c. In the Location field, type **C:\Users\*YourName*\Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 1**. Replace the text *YourName* in this path with your Windows username.

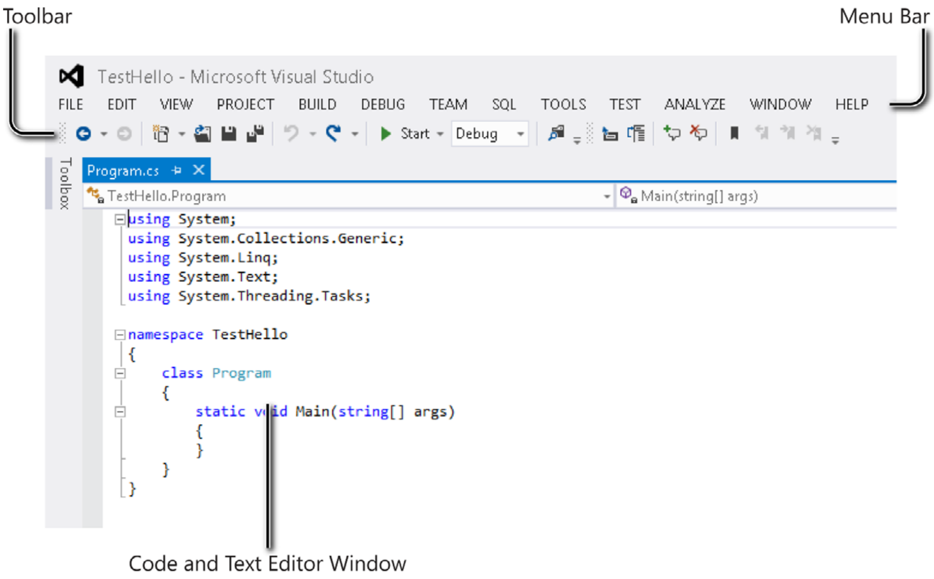
> **Note** To save space throughout the rest of this book, I will simply refer to the path C:\Users\YourName\Documents as your Documents folder.

Tip If the folder you specify does not exist, Visual Studio 2012 creates it for you.

d. In the Name field, type **TestHello** (overtype the existing name, ConsoleApplication1).

e. Ensure that the Create Directory for Solution check box is selected, and then click OK.

Visual Studio creates the project using the Console Application template and displays the starter code for the project, like this:



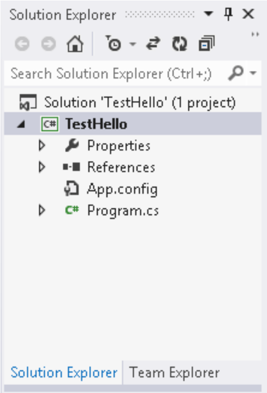
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestHello
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

The menu bar at the top of the screen provides access to the features you'll use in the programming environment. You can use the keyboard or the mouse to access the menus and commands exactly as you can in all Windows-based programs. The toolbar is located beneath the menu bar and provides button shortcuts to run the most frequently used commands.

The Code and Text Editor window occupying the main part of the screen displays the contents of source files. In a multifile project, when you edit more than one file, each source file has its own tab labeled with the name of the source file. You can click the tab to bring the named source file to the foreground in the Code and Text Editor window.

The Solution Explorer pane appears on the right side of the dialog box:



Solution Explorer displays the names of the files associated with the project, among other items. You can also double-click a file name in the Solution Explorer pane to bring that source file to the foreground in the Code and Text Editor window.

Before writing the code, examine the files listed in Solution Explorer, which Visual Studio 2012 has created as part of your project:

- **Solution 'TestHello'** This is the top-level solution file. Each application contains a single solution file. A solution can contain one or more projects, and Visual Studio 2012 creates the solution file to help organize these projects. If you use Windows Explorer to look at your Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 1\TestHello folder, you'll see that the actual name of this file is TestHello.sln.

- **TestHello** This is the C# project file. Each project file references one or more files containing the source code and other artifacts for the project, such as graphics images. All the source code in a single project must be written in the same programming language. In Windows Explorer, this file is actually called TestHello.csproj, and it is stored in the \Microsoft Press\Visual CSharp Step By Step\Chapter 1\TestHello\TestHello folder under your Documents folder.

- **Properties** This is a folder in the TestHello project. If you expand it (click the arrow next to the text Properties), you will see that it contains a file called AssemblyInfo.cs. AssemblyInfo.cs is a special file that you can use to add attributes to a program, such as the name of the author, the date the program was written, and so on. You can specify additional attributes to modify the way in which the program runs. Explaining how to use these attributes is beyond the scope of this book.

- **References** This folder contains references to libraries of compiled code that your application can use. When your C# code is compiled, it is converted into a library and given a unique name. In the .NET Framework, these libraries are called *assemblies*. Developers use assemblies to package useful functionality that they have written so they can distribute it to other developers who might want to use these features in their own applications. If you expand the References folder, you can see the default set of references that Visual Studio 2012 adds to your project. These assemblies provide access to many of the commonly used features of the .NET Framework and are provided by Microsoft with Visual Studio 2012. You will learn about many of these assemblies as you progress through the exercises in this book.

- **App.config** This is the application configuration file. It is optional, and it may not always be present. You can specify settings that your application can use at runtime to modify its behavior, such as the version of the .NET Framework to use to run the application. You will learn more about this file in later chapters of this book.
- **Program.cs** This is a C# source file, and it is displayed in the Code and Text Editor window when the project is first created. You will write your code for the console application in this file. It also contains some code that Visual Studio 2012 provides automatically, which you will examine shortly.

Writing Your First Program

The Program.cs file defines a class called *Program* that contains a method called *Main*. In C#, all executable code must be defined inside a method, and all methods must belong to a class or a struct. You will learn more about classes in Chapter 7, "Creating and Managing Classes and Objects," and you will learn about structs in Chapter 9, "Creating Value Types with Enumerations and Structures."

The *Main* method designates the program's entry point. This method should be defined in the manner specified in the *Program* class, as a static method, otherwise the .NET Framework may not recognize it as the starting point for your application when you run it. (You will look at methods in detail in Chapter 3, "Writing Methods and Applying Scope," and Chapter 7 provides more information on static methods.)

Important C# is a case-sensitive language. You must spell *Main* with an uppercase *M*.

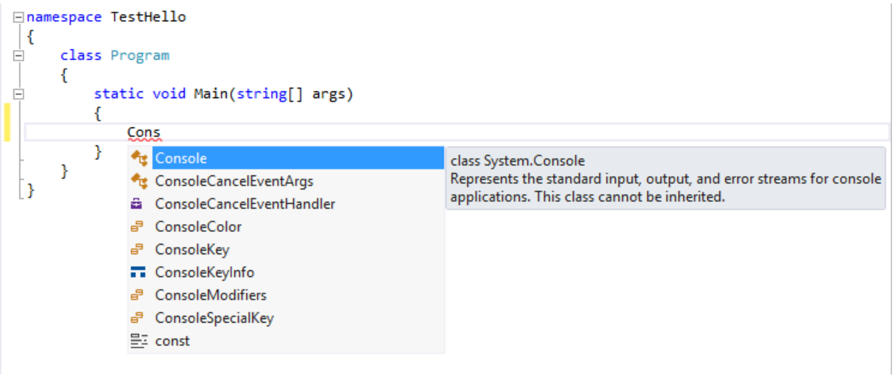
In the following exercises, you write the code to display the message "Hello World!" to the console window; you build and run your Hello World console application; and you learn how namespaces are used to partition code elements.

Write the code by using Microsoft IntelliSense

1. In the Code and Text Editor window displaying the Program.cs file, place the cursor in the *Main* method immediately after the opening brace, {, and then press Enter to create a new line.

2. On the new line, type the word **Console**; this is the name of another class provided by the assemblies referenced by your application. It provides methods for displaying messages in the console window and reading input from the keyboard.

   As you type the letter **C** at the start of the word *Console*, an IntelliSense list appears.



```
namespace TestHello
{
    class Program
    {
        static void Main(string[] args)
        {
            Cons
        }
    }
}
```

class System.Console
Represents the standard input, output, and error streams for console applications. This class cannot be inherited.

   This list contains all of the C# keywords and data types that are valid in this context. You can either continue typing or scroll through the list and double-click the Console item with the mouse. Alternatively, after you have typed **Cons**, the IntelliSense list automatically homes in on the *Console* item, and you can press the Tab or Enter key to select it.

   Main should look like this:

```
static void Main(string[] args)
{
    Console
}
```

Note *Console* is a built-in class.

3. Type a period immediately after *Console*. Another IntelliSense list appears, displaying the methods, properties, and fields of the *Console* class.

4. Scroll down through the list, select WriteLine, and then press Enter. Alternatively, you can continue typing the characters **W**, **r**, **i**, **t**, **e**, **L** until WriteLine is selected, and then press Enter.

   The IntelliSense list closes, and the word *WriteLine* is added to the source file. *Main* should now look like this:

```
static void Main(string[] args)
{
    Console.WriteLine
}
```

5. Type an opening parenthesis, (. Another IntelliSense tip appears.

   This tip displays the parameters that the *WriteLine* method can take. In fact, *WriteLine* is an *overloaded method*, meaning that the *Console* class contains more than one method named *WriteLine*—it actually provides 19 different versions of this method. Each version of the *WriteLine* method can be used to output different types of data. (Chapter 3 describes overloaded methods in more detail.) *Main* should now look like this:

   ```
   static void Main(string[] args)
   {
       Console.WriteLine(
   }
   ```

> **Tip** You can click the up and down arrows in the tip to scroll through the different overloads of *WriteLine*.

6. Type a closing parenthesis,), followed by a semicolon, ;.

   Main should now look like this:

   ```
   static void Main(string[] args)
   {
       Console.WriteLine();
   }
   ```

7. Move the cursor, and type the string **"Hello World!"**, including the quotation marks, between the left and right parentheses following the *WriteLine* method.

   Main should now look like this:

   ```
   static void Main(string[] args)
   {
       Console.WriteLine("Hello World!");
   }
   ```

> **Tip** Get into the habit of typing matched character pairs, such as (and) and { and }, before filling in their contents. It's easy to forget the closing character if you wait until after you've entered the contents.

IntelliSense Icons

When you type a period after the name of a class, IntelliSense displays the name of every member of that class. To the left of each member name is an icon that depicts the type of member. Common icons and their types include the following:

Icon	Meaning
⬡	Method (discussed in Chapter 3)
🔧	Property (discussed in Chapter 15)
⧉	Class (discussed in Chapter 7)
▀	Struct (discussed in Chapter 9)
▭	Enum (discussed in Chapter 9)
⬡↓	Extension method (discussed in Chapter 12)
•-o	Interface (discussed in Chapter 13)
▣	Delegate (discussed in Chapter 17)
⚡	Event (discussed in Chapter 17)
{ }	Namespace (discussed in the next section of this chapter)

You will also see other IntelliSense icons appear as you type code in different contexts.

You will frequently see lines of code containing two forward slashes, //, followed by ordinary text. These are comments. They are ignored by the compiler but are very useful for developers because they help document what a program is actually doing. For example:

```
Console.ReadLine(); // Wait for the user to press the Enter key
```

The compiler skips all text from the two slashes to the end of the line. You can also add multiline comments that start with a forward slash followed by an asterisk (/*). The compiler skips everything until it finds an asterisk followed by a forward slash sequence (*/), which could be many lines lower down. You are actively encouraged to document your code with as many meaningful comments as necessary.

Build and run the console application

1. On the BUILD menu, click Build Solution.

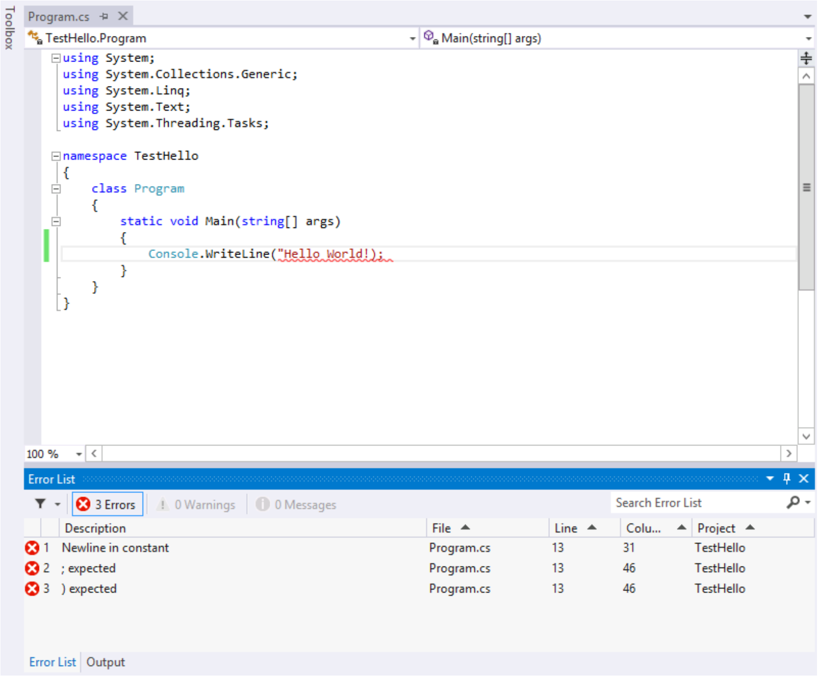
   This action compiles the C# code, resulting in a program that you can run. The Output window appears below the Code and Text Editor window.

   > **Tip** If the Output window does not appear, on the VIEW menu, click Output to display it.

   In the Output window, you should see messages similar to the following indicating how the program is being compiled:

```
1>------ Build started: Project: TestHello, Configuration: Debug Any CPU ------
1> TestHello -> C:\Users\John\Documents\Microsoft Press\Visual CSharp Step By Step\
Chapter 1\TestHello\TestHello\bin\Debug\TestHello.exe
 ========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

   If you have made any mistakes, they will be reported in the Error List window. The following image shows what happens if you forget to type the closing quotation marks after the text *Hello World* in the *WriteLine* statement. Notice that a single mistake can sometimes cause multiple compiler errors.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestHello
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!);
        }
    }
}
```

	Description	File	Line	Colu...	Project
1	Newline in constant	Program.cs	13	31	TestHello
2	; expected	Program.cs	13	46	TestHello
3	) expected	Program.cs	13	46	TestHello

Tip You can double-click an item in the Error List window, and the cursor will be placed on the line that caused the error. You should also notice that Visual Studio displays a wavy red line under any lines of code that will not compile when you enter them.

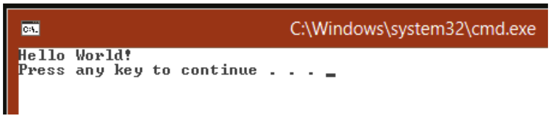
If you have followed the previous instructions carefully, there should be no errors or warnings, and the program should build successfully.

Tip There is no need to save the file explicitly before building because the Build Solution command automatically saves the file.

An asterisk after the file name in the tab above the Code and Text Editor window indicates that the file has been changed since it was last saved.

2. On the DEBUG menu, click Start Without Debugging.

   A command window opens, and the program runs. The message "Hello World!" appears, and then the program waits for you to press any key, as shown in the following graphic:



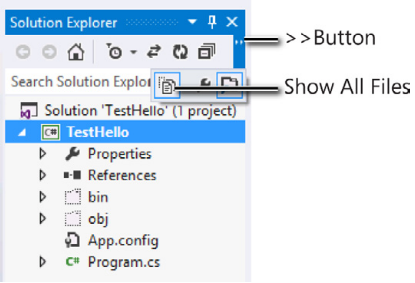
```
Hello World!
Press any key to continue . . . _
```

Note The prompt "Press any key to continue . . ." is generated by Visual Studio; you did not write any code to do this. If you run the program by using the Start Debugging command on the DEBUG menu, the application runs, but the command window closes immediately without waiting for you to press a key.

3. Ensure that the command window displaying the program's output has the focus, and then press Enter.

   The command window closes, and you return to the Visual Studio 2012 programming environment.

4. In Solution Explorer, click the TestHello project (not the solution), and then click the Show All Files toolbar button on the Solution Explorer toolbar. Note that you may need to click the >> button on the right edge of the Solution Explorer toolbar to make this button appear.



Entries named bin and obj appear above the Program.cs file. These entries correspond directly to folders named bin and obj in the project folder (Microsoft Press\Visual CSharp Step By Step\Chapter 1\TestHello\TestHello). Visual Studio creates these folders when you build your application, and they contain the executable version of the program together with some other files used to build and debug the application.

5. In Solution Explorer, expand the bin entry.

   Another folder named Debug appears.

> **Note** You might also see a folder called Release.

6. In Solution Explorer, expand the Debug folder.

   Several more items appear, including a file named TestHello.exe. This is the compiled program, and it is this file that runs when you click Start Without Debugging on the DEBUG menu. The other files contain information that is used by Visual Studio 2012 if you run your program in debug mode (when you click Start Debugging on the DEBUG menu).

Using Namespaces

The example you have seen so far is a very small program. However, small programs can soon grow into much bigger programs. As a program grows, two issues arise. First, it is harder to understand and maintain big programs than it is to understand and maintain smaller ones. Second, more code usually means more classes, with more methods, requiring you to keep track of more names. As the number of names increases, so does the likelihood of the project build failing because two or more names clash; for example, you might try and create two classes with the same name. The situation becomes more complicated when a program references assemblies written by other developers who have also used a variety of names.

In the past, programmers tried to solve the name-clashing problem by prefixing names with some sort of qualifier (or set of qualifiers). This is not a good solution because it's not scalable; names become longer, and you spend less time writing software and more time typing (there is a difference), and reading and rereading incomprehensibly long names.

Namespaces help solve this problem by creating a container for items such as classes. Two classes with the same name will not be confused with each other if they live in different namespaces. You can create a class named *Greeting* inside the namespace named *TestHello* by using the *namespace* keyword like this:

```
namespace TestHello
{
   class Greeting
   {
     ...
   }
}
```

You can then refer to the *Greeting* class as *TestHello.Greeting* in your programs. If another developer also creates a *Greeting* class in a different namespace, such as *NewNamespace*, and you install the assembly that contains this class on your computer, your programs will still work as expected because they are using the *TestHello.Greeting* class. If you want to refer to the other developer's *Greeting* class, you must specify it as *NewNamespace.Greeting*.

It is good practice to define all your classes in namespaces, and the Visual Studio 2012 environment follows this recommendation by using the name of your project as the top-level namespace. The .NET Framework class library also adheres to this recommendation; every class in the .NET Framework lives inside a namespace. For example, the *Console* class lives inside the *System* namespace. This means that its full name is actually *System.Console*.

Of course, if you had to write the full name of a class every time you used it, the situation would be no better than prefixing qualifiers or even just naming the class with some globally unique name such *SystemConsole*. Fortunately, you can solve this problem with a *using* directive in your programs. If you return to the TestHello program in Visual Studio 2012 and look at the file Program.cs in the Code and Text Editor window, you will notice the following lines at the top of the file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

These lines are *using* directives. A *using* directive brings a namespace into scope. In subsequent code in the same file, you no longer have to explicitly qualify objects with the namespace to which they belong. The five namespaces shown contain classes that are used so often that Visual Studio 2012 automatically adds these *using* statements every time you create a new project. You can add further *using* directives to the top of a source file if you need to reference other namespaces.

The following exercise demonstrates the concept of namespaces in more depth.

Try longhand names

1. In the Code and Text Editor window displaying the Program.cs file, comment out the first *using* directive at the top of the file, like this:

```
//using System;
```

2. On the BUILD menu, click Build Solution.

   The build fails, and the Error List window displays the following error message:

```
The name 'Console' does not exist in the current context.
```

3. In the Error List window, double-click the error message.

   The identifier that caused the error is highlighted in the Program.cs source file.

4. In the Code and Text Editor window, edit the *Main* method to use the fully qualified name *System.Console*.

   Main should look like this:

```
static void Main(string[] args)
{
    System.Console.WriteLine("Hello World!");
}
```

> **Note** When you type the period after *System*, the names of all the items in the *System* namespace are displayed by IntelliSense.

5. On the BUILD menu, click Build Solution.

   The project should build successfully this time. If it doesn't, make sure that *Main* is exactly as it appears in the preceding code, and then try building again.

6. Run the application to make sure it still works by clicking Start Without Debugging on the DEBUG menu.

7. When the program runs and displays "Hello World!", press Enter in the console window to return to Visual Studio 2012.

Namespaces and Assemblies

A *using* directive simply brings the items in a namespace into scope and frees you from having to fully qualify the names of classes in your code. Classes are compiled into *assemblies*. An assembly is a file that usually has the .dll file name extension, although strictly speaking, executable programs with the .exe file name extension are also assemblies.

An assembly can contain many classes. The classes that the .NET Framework class library comprises, such as *System.Console*, are provided in assemblies that are installed on your computer together with Visual Studio. You will find that the .NET Framework class library contains thousands of classes. If they were all held in the same assembly, the assembly would be huge and difficult to maintain. (If Microsoft updated a single method in a single class, it would have to distribute the entire class library to all developers!)

For this reason, the .NET Framework class library is split into a number of assemblies, partitioned by the functional area to which the classes they contain relate. For example, a "core" assembly (actually called *mscorlib.dll*) contains all the common classes, such as *System.Console*, and further assemblies contain classes for manipulating databases, accessing web services, building GUIs, and so on. If you want to make use of a class in an assembly, you must add to your project a reference to that assembly. You can then add *using* statements to your code that bring the items in namespaces in that assembly into scope.

You should note that there is not necessarily a 1:1 equivalence between an assembly and a namespace. A single assembly can contain classes defined in many namespaces, and a single namespace can span multiple assemblies. For example, the classes and items in the *System* namespace are actually implemented by several assemblies, including *mscorlib.dll*, *System.dll*, and *System.Core.dll*, among others. This all sounds very confusing at first, but you will soon get used to it.

When you use Visual Studio to create an application, the template you select automatically includes references to the appropriate assemblies. For example, in Solution Explorer for the TestHello project, expand the References folder. You will see that a console application automatically contains references to assemblies called *Microsoft.CSharp, System, System.Core, System.Data, System.Data.DataExtensions, System.Xml*, and *System.Xml.Linq*. You may be surprised to see that *mscorlib.dll* is not included in this list; this is because all .NET Framework applications must use this assembly, as it contains fundamental runtime functionality. The References folder lists only the optional assemblies; you can add or remove assemblies from this folder as necessary.

You can add references for additional assemblies to a project by right-clicking the References folder and clicking Add Reference—you will perform this task in later exercises. You can remove an assembly by right-clicking the assembly in the References folder and then clicking Remove.

Creating a Graphical Application

So far, you have used Visual Studio 2012 to create and run a basic console application. The Visual Studio 2012 programming environment also contains everything you need to create graphical applications for Windows 7 and Windows 8. You can design the user interface of a Windows application interactively. Visual Studio 2012 then generates the program statements to implement the user interface you've designed.

Visual Studio 2012 provides you with two views of a graphical application: the *design view* and the *code view*. You use the Code and Text Editor window to modify and maintain the code and program logic for a graphical application, and you use the Design View window to lay out your user interface. You can switch between the two views whenever you want.

In the following set of exercises, you'll learn how to create a graphical application using Visual Studio 2012. This program will display a simple form containing a text box where you can enter your name and a button that when clicked displays a personalized greeting.

Important In Windows 7, Visual Studio 2012 provides two templates for building graphical applications: the Windows Forms Application template and the WPF Application template. Windows Forms is a technology that first appeared with the .NET Framework version 1.0. WPF, or Windows Presentation Foundation, is an enhanced technology that first appeared with the .NET Framework version 3.0. It provides many additional features and capabilities over Windows Forms, and you should consider using WPF instead of Windows Forms for all new Windows 7 development.

You can also build Windows Forms and WPF applications in Windows 8. However, Windows 8 provides a new style of user interface, referred to as the Windows Store style, and applications that use this style of user interface are called Windows Store applications (or *apps*). Windows 8 has been designed to operate on a variety of hardware, including computers with touch-sensitive screens and tablet computers or slates. These computers enable users to interact with applications by using touch-based gestures—for example, users can swipe applications with their fingers to move them around the screen and rotate them, or "pinch" and "stretch" applications to zoom out and back in again. Additionally, many slates include sensors that can detect the orientation of the device, and Windows 8 can pass this information to an application, which can then dynamically adjust the user interface to match the orientation (it can switch from landscape to portrait mode, for example). If you have installed Visual Studio 2012 on a Windows 8 computer, you are provided with an additional set of templates for building Windows Store apps.

To cater to both Windows 7 and Windows 8 developers, I have provided instructions in many of the exercises for using the WPF templates if you are running Windows 7, or Windows 8 if you want to use the Windows Store style of user interface. Of course, you can follow the Windows 7 and WPF instructions on Windows 8 if you prefer.

If you want more information about the specifics of writing Windows 8 applications, the chapters in Part IV of this book provide more detail and guidance.

Create a graphical application in Visual Studio 2012

- If you are using Windows 8, perform the following operations to create a new graphical application:

  a. Start Visual Studio 2012 if it is not already running.

  b. On the FILE menu, point to New, and then click Project.

  The New Project dialog box opens.

  c. In the left pane, under Installed Templates, expand the Visual C# folder if it is not already expanded, and then click the Windows Store folder.

  d. In the middle pane, click the Blank App (XAML) icon.

> **Note** XAML stands for Extensible Application Markup Language, the language that Windows Store apps use to define the layout for the GUI of an application. You will learn more about XAML as you progress through the exercises in this book.

  e. Ensure that the Location field refers to the \Microsoft Press\Visual CSharp Step By Step\Chapter 1 folder under your Documents folder.

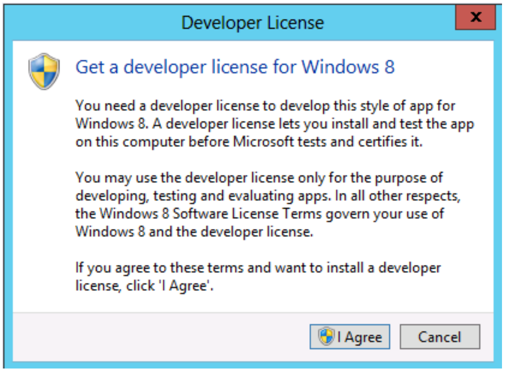
  f. In the Name field, type **Hello**.

  g. In the Solution field, ensure that Create New Solution is selected.

  This action creates a new solution for holding the project. The alternative, Add to Solution, adds the project to the TestHello solution, which is not what you want for this exercise.

  h. Click OK.

  If this is the first time that you have created a Windows Store app, you will be prompted to apply for a developer license. You must agree to the terms and conditions indicated in the dialog box before you can continue to build Windows Store apps. If you concur with these conditions, click I Agree. You will be prompted to sign into Windows Live (you can create a new account at this point if necessary), and a developer license will be created and allocated to you.



Get a developer license for Windows 8

You need a developer license to develop this style of app for Windows 8. A developer license lets you install and test the app on this computer before Microsoft tests and certifies it.

You may use the developer license only for the purpose of developing, testing and evaluating apps. In all other respects, the Windows 8 Software License Terms govern your use of Windows 8 and the developer license.

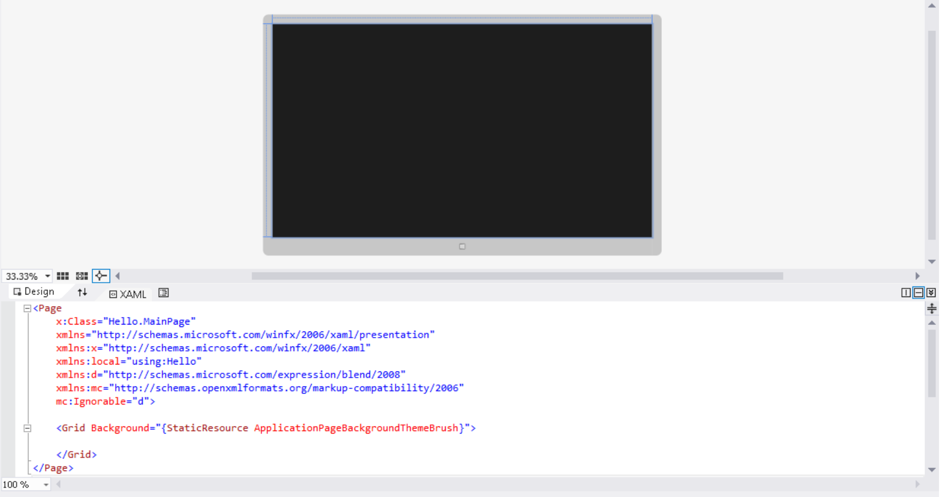
If you agree to these terms and want to install a developer license, click 'I Agree'.

i. After the application has been created, look in the Solution Explorer window.

Don't be fooled by the name of the application template—although it is called Blank App, this template actually provides a number of files and contains a significant amount of code. For example, if you expand the Common folder in Solution Explorer, you will find a file named StandardStyles.xaml. This file contains XAML code defining styles that you can use to format and present data for display. Part IV, "Building Professional Windows 8 Applications with C#," describes the purpose of these styles in more detail, so don't worry about them for now. Similarly, if you expand the MainPage.xaml folder, you will find a C# file named MainPage.xaml.cs. This file is where you add the code that runs when the user interface defined by the MainPage.xaml file is displayed.

j. In Solution Explorer, double-click MainPage.xaml.

This file contains the layout of the user interface. The Design View window shows two representations of this file:



```
<Page
    x:Class="Hello.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Hello"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    </Grid>
</Page>
```

At the top is a graphical view depicting the screen of a tablet computer. The lower pane contains a description of the contents of this screen using XAML. XAML is an XML-like language used by Windows Store apps and WPF applications to define the layout of a form and its contents. If you have knowledge of XML, XAML should look familiar.

In the next exercise, you will use the Design View window to lay out the user interface for the application, and you will examine the XAML code that this layout generates.

- If you are using Windows 7, perform the following tasks:

  a. Start Visual Studio 2012 if it is not already running.

  b. On the FILE menu, point to New, and then click Project.

  The New Project dialog box opens.

  c. In the left pane, under Installed Templates, expand the Visual C# folder if it is not already expanded, and then click the Windows folder.

  d. In the middle pane, click the WPF Application icon.

  e. Ensure that the Location field refers to the \Microsoft Press\Visual CSharp Step By Step\Chapter 1 folder under your Documents folder.

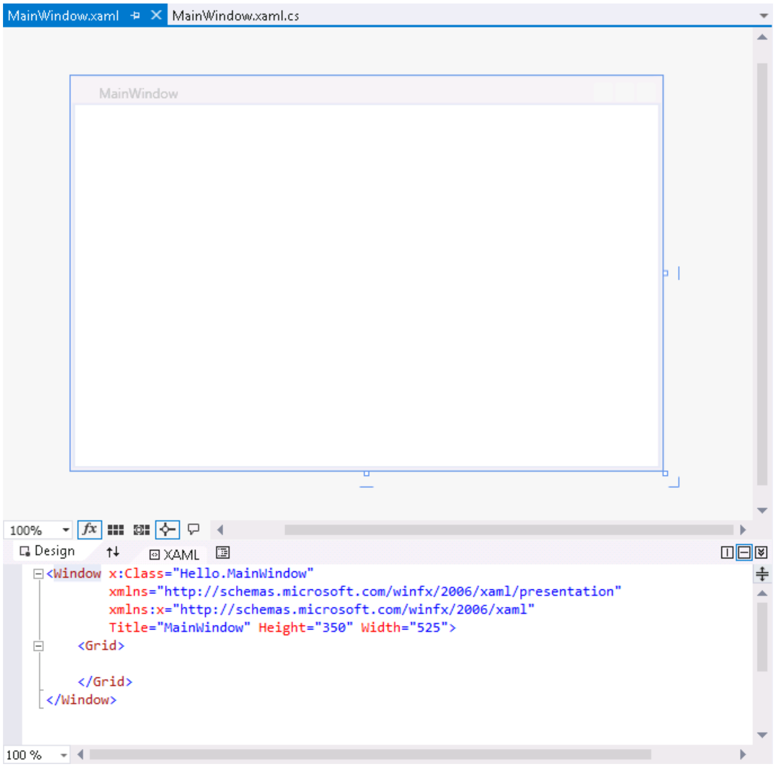
  f. In the Name field, type **Hello**.

  g. In the Solution field, ensure that Create New Solution is selected.

  h. Click OK.

  The WPF Application template generates fewer items than the Windows Store Blank App template; it contains none of the styles generated by the Blank App template as the functionality that these styles embody is specific to Windows 8. However, the WPF Application template does generate a default window for your application. Like a Windows Store app, this window is defined by using XAML, but in this case it is called MainWindow.xaml by default.

  i. In Solution Explorer, double-click MainWindow.xaml to display the contents of this file in the Design View window.



```
<Window x:Class="Hello.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Tip Close the Output and Error List windows to provide more space for displaying the Design View window.

Note Before going further, it is worth explaining some terminology. In a typical WPF application, the user interface consists of one or more *windows*, but in a Windows Store app the corresponding items are referred to as *pages* (strictly speaking, a WPF application can also contain pages, but I don't want to confuse matters further at this point). To avoid repeating the rather verbose phrase "WPF window or Windows Store app page" repeatedly throughout this book, I will simply refer to both items by using the blanket term *form*. However, I will continue to use the word *window* to refer to items in the Visual Studio 2012 IDE, such as the Design View window.

In the following exercises, you use the Design View window to add three controls to the form displayed by your application, and you examine some of the C# code automatically generated by Visual Studio 2012 to implement these controls.

Note The steps in the following exercises are common to Windows 7 and Windows 8, except where any differences are explicitly called out.

Create the user interface

1. Click the Toolbox tab that appears to the left of the form in the Design View window.

   The Toolbox appears, partially obscuring the form, and displays the various components and controls that you can place on a form.

2. If you are using Windows 8, expand the Common XAML Controls section.

   If you are using Windows 7, expand the Common WPF Controls section.

   This section displays a list of controls that are used by most graphical applications.

   Tip The All XAML Controls section (Windows 8) or All WPF Controls section (Windows 7) displays a more extensive list of controls.

3. In the Common XAML Controls section or Common WPF Controls section, click TextBlock, and then drag the *TextBlock* control onto the form displayed in the Design View window.

   Tip Make sure you select the *TextBlock* control and not the *TextBox* control. If you accidentally place the wrong control on a form, you can easily remove it by clicking the item on the form and then pressing Delete.

   A *TextBlock* control is added to the form (you will move it to its correct location in a moment), and the Toolbox disappears from view.

   Tip If you want the Toolbox to remain visible but not to hide any part of the form, click the Auto Hide button to the right in the Toolbox title bar. (It looks like a pin.) The Toolbox appears permanently on the left side of the Visual Studio 2012 window, and the Design View window shrinks to accommodate it. (You might lose a lot of space if you have a low-resolution screen.) Clicking the Auto Hide button once more causes the Toolbox to disappear again.

4. The *TextBlock* control on the form is probably not exactly where you want it. You can click and drag the controls you have added to a form to reposition them. Using this technique, move the *TextBlock* control so that it is positioned toward the upper-left corner of the form. (The exact placement is not critical for this application.) Notice that you may need to click away from the control and then click it again before you are able to move it in the Design View window.

   The XAML description of the form in the lower pane now includes the *TextBlock* control, together with properties such as its location on the form, governed by the *Margin* property, the default text displayed by this control in the *Text* property, the alignment of text displayed by this control specified by the *HorizontalAlignment* and *VerticalAlignment* properties, and whether text should wrap if it exceeds the width of the control.

   If you are using Windows 8, the XAML code for the *TextBlock* will look similar to this (your values for the *Margin* property may be slightly different, depending on where you have positioned the *TextBlock* control on the form):

   ```
   <TextBlock HorizontalAlignment="Left" Margin="400,200,0,0" TextWrapping="Wrap"
   Text="TextBlock" VerticalAlignment="Top"/>
   ```

   If you are using Windows 7, the XAML code will be much the same, except that the units used by the *Margin* property operate on a different scale due to the finer resolution of Windows 8 devices.

   The XAML pane and the Design View window have a two-way relationship with each other. You can edit the values in the XAML pane, and the changes will be reflected in the Design View window. For example, you can change the location of the *TextBlock* control by modifying the values in the *Margin* property.

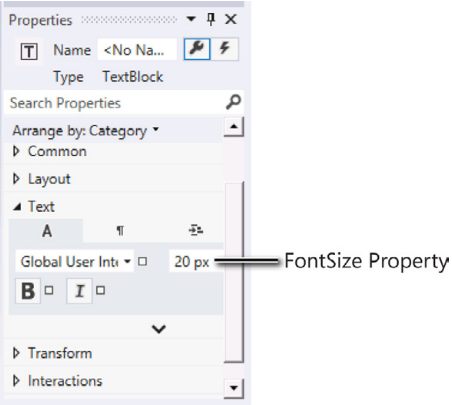
5. On the VIEW menu, click Properties Window.

   If it was not already displayed, the Properties window appears at the lower-right of the screen, under Solution Explorer. You can specify the properties of controls by using the XAML pane under the Design View window, but the Properties window provides a more convenient way for you to modify the properties for items on a form, as well as other items in a project.

   The Properties window is context sensitive in that it displays the properties for the currently selected item. If you click the form displayed in the Design View window, outside of the *TextBlock* control, you can see that the Properties window displays the properties for a *Grid* element. If you look at the XAML pane, you should see that the *TextBlock* control is contained within a *Grid* element. All forms contain a *Grid* element that controls the layout of displayed items; you can define tabular layouts by adding rows and columns to the *Grid*, for example.

6. Click the *TextBlock* control in the Design View window. The Properties window displays the properties for the *TextBlock* control again.

7. In the Properties window, expand the *Text* property. Change the *FontSize* property to **20 px** and then press Enter. This property is located next to the drop-down list box containing the name of the font, which will be different for Windows 8 (Global User Interface) and Windows 7 (Segoe UI):



Name <No Na...
Type TextBlock
Global User Int
20 px

Note The suffix *px* indicates that the font size is measured in pixels.

8. In the XAML pane below the Design View window, examine the text that defines the *TextBlock* control. If you scroll to the end of the line, you should see the text FontSize="20". Any changes that you make using the Properties window are automatically reflected in the XAML definitions and vice versa.

   Overtype the value of the *FontSize* property in the XAML pane, and change it to **24**. The font size of the text for the *TextBlock* control in the Design View window and the Properties window changes.

9. In the Properties window, examine the other properties of the *TextBlock* control. Feel free to experiment by changing them to see their effects.

   Notice that as you change the values of properties, these properties are added to the definition of the *TextBlock* control in the XAML pane. Each control that you add to a form has a default set of property values, and these values are not displayed in the XAML pane unless you change them.

10. Change the value of the *Text* property of the *TextBlock* control from TextBlock to **Please enter your name**. You can do this either by editing the *Text* element in the XAML pane or by changing the value in the Properties window (this property is located in the Common section in the Properties window).

    Notice that the text displayed in the *TextBlock* control in the Design View window changes.

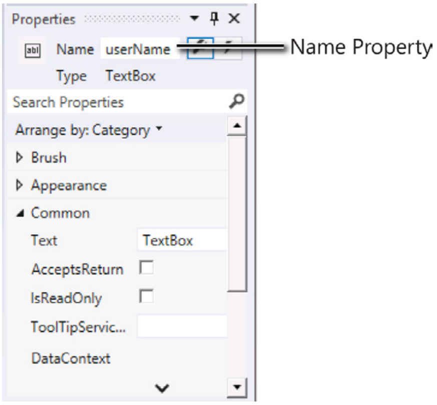
11. Click the form in the Design View window, and then display the Toolbox again.

12. In the Toolbox, click and drag the *TextBox* control onto the form. Move the *TextBox* control so that it is directly underneath the *TextBlock* control.

> **Tip** When you drag a control on a form, alignment indicators appear automatically when the control becomes aligned vertically or horizontally with other controls. This gives you a quick visual cue for making sure that controls are lined up neatly.

13. In the Design View window, place the mouse over the right edge of the *TextBox* control. The mouse pointer should change to a double-headed arrow to indicate that you can resize the control. Click the mouse and drag the right edge of the *TextBox* control until it is aligned with the right edge of the *TextBlock* control above; a guide should appear when the two edges are correctly aligned.

14. While the *TextBox* control is selected, change the value of the *Name* property displayed at the top of the Properties window from <No Name> to **userName**:



Properties

Name userName

Type TextBox

Search Properties

Arrange by: Category

Brush

Appearance

Common

Text TextBox

AcceptsReturn □

IsReadOnly □

ToolTipServic...

DataContext

Name Property

> **Note** You will learn more about naming conventions for controls and variables in Chapter 2, "Working with Variables, Operators, and Expressions."

15. Display the Toolbox again, and then click and drag a *Button* control onto the form. Place the *Button* control to the right of the *TextBox* control on the form so that the bottom of the button is aligned horizontally with the bottom of the text box.

16. Using the Properties window, change the *Name* property of the *Button* control to **ok** and change the *Content* property (in the Common section) from Button to **OK** and press Enter. Verify that the caption of the *Button* control on the form changes to display the text *OK*.

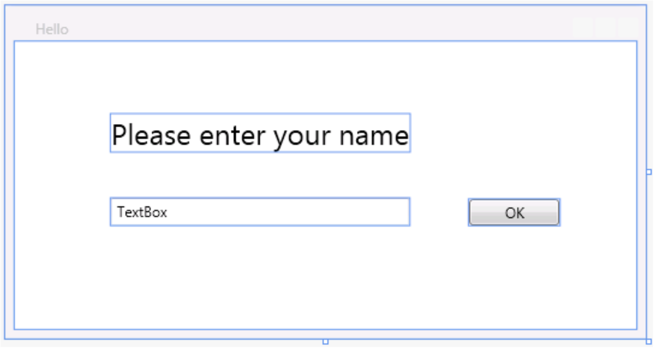
17. If you are using Windows 7, click the title bar of the form in the Design View window. In the Properties window, change the *Title* property (in the Common section again) from MainWindow to **Hello**.

Note Windows Store apps do not have a title bar.

18. If you are using Windows 7, in the Design View window, click the title bar of the Hello form. Notice that a resize handle (a small square) appears in the lower-right corner of the Hello form. Move the mouse pointer over the resize handle. When the pointer changes to a diagonal double-headed arrow, click and drag the pointer to resize the form. Stop dragging and release the mouse button when the spacing around the controls is roughly equal.

Important Click the title bar of the Hello form and not the outline of the grid inside the Hello form before resizing it. If you select the grid, you will modify the layout of the controls on the form but not the size of the form itself.

The Hello form should now look similar to the following figure.

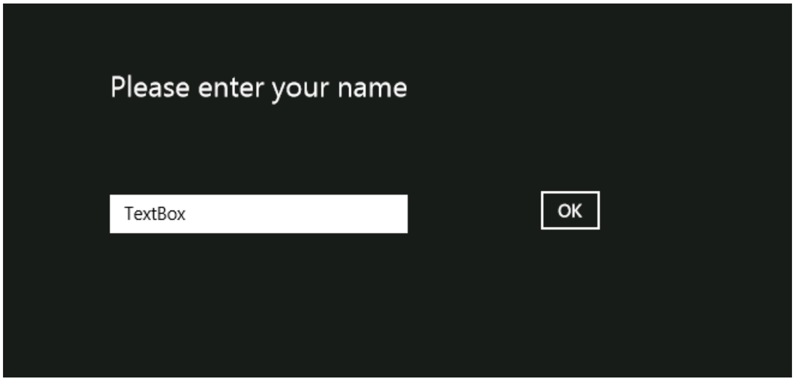


Note Pages in Windows Store apps cannot be resized in the same way as Windows 7 forms; when they run, they automatically occupy the full screen of the device. However, they can adapt themselves to different screen resolutions and device orientation, and present different views when they are "snapped." You can easily see what your application looks like on a different device by clicking Device Window on the DESIGN menu and then selecting from the different screen resolutions available in the Display drop-down list. You can also see how your application appears in portrait mode or when snapped by selecting the Portrait orientation or Snapped view from the list of available views.

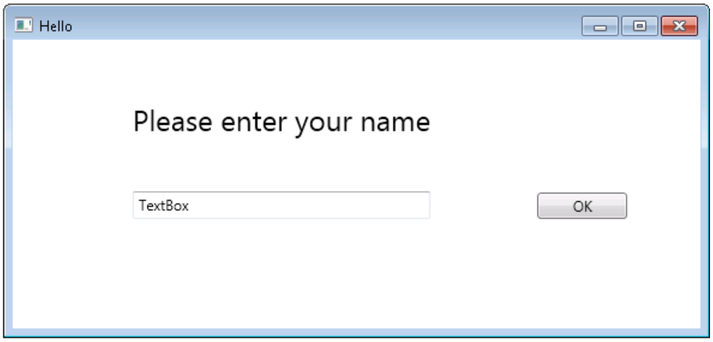
19. On the BUILD menu, click Build Solution, and verify that the project builds successfully.

20. On the DEBUG menu, click Start Debugging.

    The application should run and display your form. If you are using Windows 8, the form occupies the entire screen and looks like this:

    If you are using Windows 7, the form looks like this:

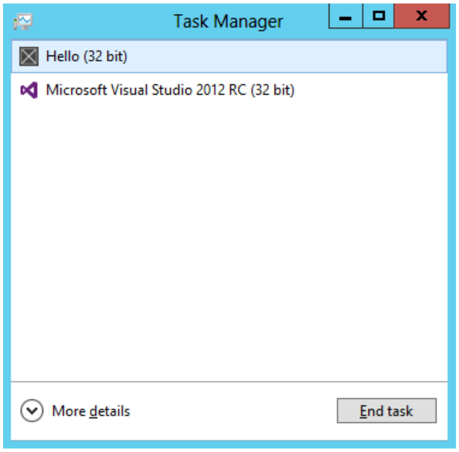
    You can delete the text *TextBox*, type your name in the text box, and click OK, but nothing happens yet. You need to add some code to indicate what should happen when the user clicks the OK button, which is what you will do next.

21. Return to Visual Studio 2012, and on the DEBUG menu click Stop Debugging. Alternatively, if you are using Windows 7, click the close button (the *X* in the upper-right corner of the form) to close the form and return to Visual Studio.

Closing a Windows Store App

If you are using Windows 8 and you clicked Start Without Debugging on the DEBUG menu to run the application, you will need to forcibly close it. This is because unlike console applications, the lifetime of a Windows Store app is managed by the operating system rather than the user. Windows 8 suspends an application when it is not currently displayed, and it will terminate the application when the operating system needs to free the resources occupied by the application. The most reliable way to forcibly stop the Hello application is to click (or place your finger if you have a touch-sensitive screen) at the top of the screen, and then click and drag (or swipe) the application to the bottom of the screen. This action closes the application and returns you to the Windows Start screen, where you can switch back to Visual Studio. Alternatively, you can perform the following tasks:

1. Click, or place your finger, in the top-right corner of the screen and then drag the image of Visual Studio to the middle of the screen (or press the Windows key and the B key at the same time).
2. Right-click the Windows taskbar at the bottom of the desktop and then click Task Manager.
3. In the Task Manager window, click the Hello application, and then click End Task.



4. Close the Task Manager window.

You have managed to create a graphical application without writing a single line of C# code. It does not do much yet (you will have to write some code soon), but Visual Studio 2012 actually generates a lot of code for you that handles routine tasks that all graphical applications must perform, such as starting up and displaying a window. Before adding your own code to the application, it helps to have an understanding of what Visual Studio has produced for you. The structure is slightly different between a Windows Store app and a Windows 7 WPF application, and the following sections summarize these application styles separately.

Examining the Windows Store App

If you are using Windows 8, in Solution Explorer, click the arrow adjacent to the MainPage.xaml file to expand the node. The file MainPage.xaml.cs appears; double-click this file. The following code for the form is displayed in the Code and Text Editor window:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?LinkId=234238

namespace Hello
{
  /// <summary>
  /// An empty page that can be used on its own or navigated to within a Frame.
  /// </summary>
  public sealed partial class MainPage : Page
  {
    public MainPage()
    {
      this.InitializeComponent();
    }

    /// <summary>
    /// Invoked when this page is about to be displayed in a Frame.
    /// </summary>
    /// <param name="e">Event data that describes how this page was reached.  The Parameter
    /// property is typically used to configure the page.</param>
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
    }
  }
}
```

In addition to a good number of *using* directives bringing into scope some namespaces that most Windows Store apps use, the file contains the definition of a class called *MainPage* but not much else. There is a little bit of code for the *MainPage* class known as a constructor that calls a method called *InitializeComponent*. A *constructor* is a special method with the same name as the class. It is executed when an instance of the class is created and can contain code to initialize the instance. You will learn about constructors in Chapter 7.

The class also contains a method called *OnNavigatedTo*. This is an example of a method that is invoked by an event, and the code in this method runs when the window is displayed. You can add your own code to this method to configure the display if necessary. You will learn more about events in Chapter 17, "Introducing Generics," and Chapter 25, "Implementing the User Interface for a Windows Store App," provides more information about the *NavigatedTo* event.

The class actually contains a lot more code than the few lines shown in the MainPage.xaml.cs file, but much of it is generated automatically based on the XAML description of the form, and it is hidden from you. This hidden code performs operations such as creating and displaying the form, and creating and positioning the various controls on the form.

Tip You can also display the C# code file for a page in a Windows Store app by clicking Code on the VIEW menu when the Design View window is displayed.

At this point, you might be wondering where the *Main* method is and how the form gets displayed when the application runs. Remember that in a console application, *Main* defines the point at which the program starts. A graphical application is slightly different.

In Solution Explorer, you should notice another source file called App.xaml. If you expand the node for this file, you will see another file called App.xaml.cs. In a Windows Store app, the App.xaml file provides the entry point when the application starts running. If you double-click App.xaml.cs in Solution Explorer, you should see some code that looks similar to this:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Application template is documented at http://go.microsoft.com/fwlink/?LinkId=234227

namespace Hello
{
    /// <summary>
    /// Provides application-specific behavior to supplement the default Application class.
    /// </summary>
    sealed partial class App : Application
    {
        /// <summary>
        /// Initializes the singleton application object.  This is the first line of authored
        /// executed, and as such is the logical equivalent of main() or WinMain().
        /// </summary>
        public App()
        {
            this.InitializeComponent();
            this.Suspending += OnSuspending;
        }

        /// <summary>
        /// Invoked when the application is launched normally by the end user.  Other entry
        /// will be used when the application is launched to open a specific file, to display
        /// search results, and so forth.
        /// </summary>
        /// <param name="args">Details about the launch request and process.</param>
        protected override void OnLaunched(LaunchActivatedEventArgs args)
        {
            Frame rootFrame = Window.Current.Content as Frame;

            // Do not repeat app initialization when the Window already has content,
            // just ensure that the window is active
            if (rootFrame == null)
            {
                // Create a Frame to act as the navigation context and navigate to the first
                rootFrame = new Frame();

                if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
                {
                    //TODO: Load state from previously suspended application
                }

                // Place the frame in the current Window
                Window.Current.Content = rootFrame;
            }

            if (rootFrame.Content == null)
            {
                // When the navigation stack isn't restored navigate to the first page,
                // configuring the new page by passing required information as a navigation
                // parameter
                if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
                {
                    throw new Exception("Failed to create initial page");
                }
            }
            // Ensure the current window is active
            Window.Current.Activate();
        }

        /// <summary>
        /// Invoked when application execution is being suspended.  Application state is saved
        /// without knowing whether the application will be terminated or resumed with the
        /// of memory still intact.
        /// </summary>
        /// <param name="sender">The source of the suspend request.</param>
        /// <param name="e">Details about the suspend request.</param>
        private void OnSuspending(object sender, SuspendingEventArgs e)
        {
            var deferral = e.SuspendingOperation.GetDeferral();
            //TODO: Save application state and stop any background activity
            deferral.Complete();
        }
    }
}
```

Much of this code consists of comments (the lines beginning *"///"*) and other statements that you don't need to understand just yet, but the key elements are located in the *OnLaunched* method, highlighted in bold. This method runs when the application starts, and the code in this method causes the application to create a new *Frame* object, display the MainPage form in this frame, and then activate it. It is not necessary at this stage to fully comprehend how this code works or the syntax of any of these statements, but simply appreciate that this is how the application displays the form when it starts running.

Examining the WPF Application

If you are using Windows 7, in Solution Explorer, click the arrow adjacent to the MainWindow.xaml file to expand the node. The file MainWindow.xaml.cs appears; double-click this file. The code for the form is displayed in the Code and Text Editor window. It looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
namespace Hello
{
   /// <summary>
   /// Interaction logic for MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window
   {
      public MainWindow()
      {
          InitializeComponent();
      }
   }
}
```

This code looks similar to that for the Windows Store app, but there are some significant differences. First, there is no *OnNavigatedTo* method; this is because the way in which a WPF application moves between forms is different from a Windows Store app. Second, many of the namespaces referenced by the *using* directives at the top of the file are different. For example, WPF applications make use of objects defined in namespaces that begin with the prefix *System.Windows*, whereas Windows Store apps use objects defined in namespaces that start with *Windows.UI*. This difference is not just cosmetic. These namespaces are implemented by different assemblies, and the controls and functionality that these assemblies provide are different between WPF and Windows Store apps, although they may have similar names. Going back to the earlier exercise, you added *TextBlock*, *TextBox*, and *Button* controls to the WPF form and the Windows Store app. Although these controls have the same name in each style of application, they are defined in different assemblies: *Windows.UI.Xaml.Controls* for Windows Store apps and *System.Windows.Controls* for WPF applications. The controls for Windows Store apps have been specifically designed and optimized for touch interfaces, whereas the WPF controls are intended primarily for use in mouse-driven systems.

As with the code in the Windows Store app, the constructor in the *MainWindow* class initializes the WPF form by calling the *InitializeComponent* method. Again, as before, the code for this method is hidden from you, and it performs operations such as creating and displaying the form, and creating and positioning the various controls on the form.

The way in which a WPF application specifies the initial form to be displayed is different from that of a Windows Store app. Like a Windows Store app, it defines an *App* object defined in the App.xaml file to provide the entry point for the application, but the form to display is specified declaratively as part of the XAML code rather than programmatically. If you double-click the App.xaml file in Solution Explorer (not App.xaml.cs), you can examine the XAML description. One property in the XAML code is called *StartupUri*, and it refers to the MainWindow.xaml file, as shown in bold in the following code example:

```
<Application x:Class="Hello.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com.winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

In a WPF application, the *StartupUri* property of the *App* object indicates which form to display.

Adding Code to the Graphical Application

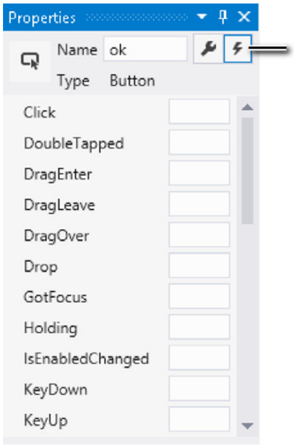
Now that you know a little bit about the structure of a graphical application, the time has come to write some code to make your application actually do something.

Write the code for the OK button

1. Open the MainPage.xaml file (Windows 8) or MainWindow.xaml file (Windows 7) in the Design View window (double-click MainPage.xaml or MainWindow.xaml in Solution Explorer).

2. In the Design View window, click the OK button on the form to select it.

3. In the Properties window, click the Event Handlers for Selected Element button. This button displays an icon that looks like a bolt of lightning.

   Name ok
   Type Button

   Event Handlers for Selected Element

   The Properties window displays a list of event names for the *Button* control. An event indicates a significant action that usually requires a response, and you can write your own code to perform this response.

4. In the box adjacent to the *Click* event, type **okClick** and then press Enter.

   The MainPage.xaml.cs file (Windows 8) or MainWindow.xaml.cs file (Windows 7) appears in the Code and Text Editor window, and a new method called *okClick* is added to the *MainPage* or *MainWindow* class. The method looks like this:

   ```
   private void okClick(object sender, RoutedEventArgs e)
   {

   }
   ```

   Do not worry too much about the syntax of this code just yet—you will learn all about methods in Chapter 3.

5. If you are using Windows 8, perform the following tasks:

   a. Add the following *using* directive shown in bold to the list at the top of the file (the ellipsis character, ..., indicates statements that have been omitted for brevity):

```
using System;
...
using Windows.UI.Xaml.Navigation;
using Windows.UI.Popups;
```

b. Add the following code shown in bold to the *okClick* method:

```
void okClick(object sender, RoutedEventArgs e)
{
  MessageDialog msg = new MessageDialog("Hello " + userName.Text);
  msg.ShowAsync();
}
```

This code will run when the user clicks the OK button. Again, do not worry too much about the syntax, just make sure you copy it exactly as shown; you will find out what these statements mean in the next few chapters. The key things to understand are that the first statement creates a *MessageDialog* object with the message "Hello <*YourName*>", where <*YourName*> is the name that you type into the *TextBox* on the form. The second statement displays the *MessageDialog*, causing it to appear on the screen. The *MessageDialog* class is defined in the *Windows.UI.Popups* namespace, which is why you added it in step a.

6. If you are using Windows 7, just add the single statement shown in bold to the *okClick* method:

```
void okClick(object sender, RoutedEventArgs e)
{
  MessageBox.Show("Hello " + userName.Text);
}
```

This code performs a similar function to that of the Windows Store app, except that it uses a different class called *MessageBox*. This class is defined in the *System.Windows* namespace, which is already referenced by the existing *using* directives at the top of the file, so you don't need to add it yourself.

7. Click the MainPage.xaml tab or the MainWindow.xaml tab above the Code and Text Editor window to display the form in the Design View window again.

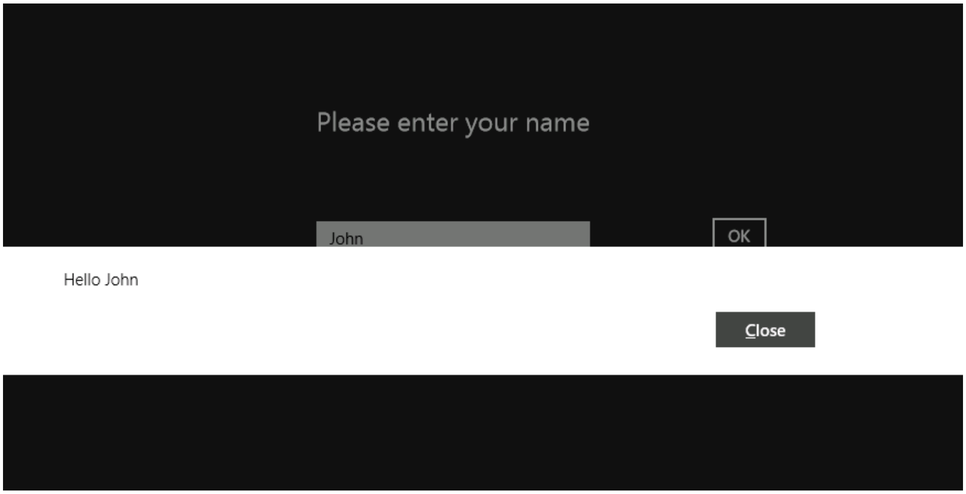
8. In the lower pane displaying the XAML description of the form, examine the *Button* element, but be careful not to change anything. Notice that it now contains an element called *Click* that refers to the *okClick* method:

```
<Button x:Name="ok" ... Click="okClick" />
```

9. On the DEBUG menu, click Start Debugging.

10. When the form appears, in the text box overtype the text *TextBox* with your name and then click OK.

If you are using Windows 8, a message dialog appears across the middle of the screen, welcoming you by name:

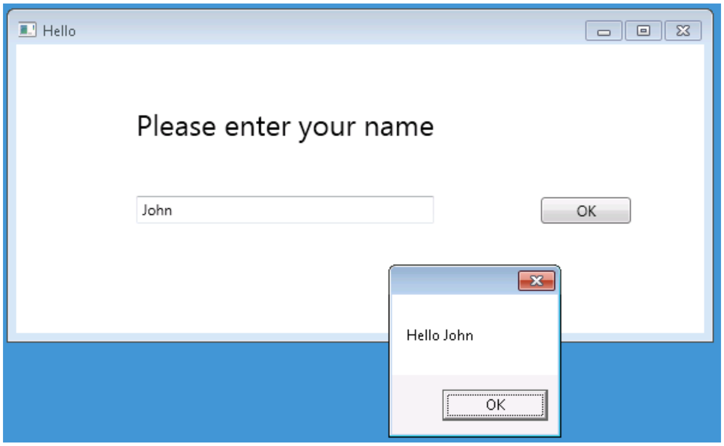


Please enter your name

John

Hello John

If you are using Windows 7, a message box appears displaying the following greeting:



Please enter your name

John

Hello John

11. Click Close in the message dialog (Windows 8) or OK (Windows 7) in the message box.

12. Return to Visual Studio 2012, and on the DEBUG menu click Stop Debugging.

Summary

In this chapter, you have seen how to use Visual Studio 2012 to create, build, and run applications. You have created a console application that displays its output in a console window, and you have created a WPF application with a simple GUI.

- If you want to continue to the next chapter

  Keep Visual Studio 2012 running, and turn to Chapter 2.

- If you want to exit Visual Studio 2012 now

  On the FILE menu, click Exit. If you see a Save dialog box, click Yes and save the project.

Chapter 1 Quick Reference

To	Do this
Create a new console application using Visual Studio 2012	On the FILE menu, point to New, and then click Project to open the New Project dialog box. In the left pane, under Installed Templates, click Visual C#. In the middle pane, click Console Application. Specify a directory for the project files in the Location box. Type a name for the project. Click OK.
Create a new Windows Store blank graphical application for Windows 8 using Visual Studio 2012	On the FILE menu, point to New, and then click Project to open the New Project dialog box. In the left pane, under Installed Templates, expand Visual C# and then click Windows Store. In the middle pane, click Blank App (XAML). Specify a directory for the project files in the Location box. Type a name for the project. Click OK.
Create a new WPF graphical application for Windows 7 using Visual Studio 2012	On the FILE menu, point to New, and then click Project to open the New Project dialog box. In the left pane, under Installed Templates, expand Visual C# and then click Windows. In the middle pane, click WPF Application. Specify a directory for the project files in the Location box. Type a name for the project. Click OK.
Build the application	On the BUILD menu, click Build Solution.
Run the application in debug mode	On the DEBUG menu, click Start Debugging.
Run the application without debugging	On the DEBUG menu, click Start Without Debugging.

CHAPTER 2

Working with Variables, Operators, and Expressions

After completing this chapter, you will be able to

- Understand statements, identifiers, and keywords.
- Use variables to store information.
- Work with primitive data types.
- Use arithmetic operators such as the plus sign (+) and the minus sign (–).
- Increment and decrement variables.

In Chapter 1, "Welcome to C#," you learned how to use the Microsoft Visual Studio 2012 programming environment to build and run a console program and a graphical application. This chapter introduces you to the elements of Microsoft Visual C# syntax and semantics, including statements, keywords, and identifiers. You'll study the primitive types that are built into the C# language and the characteristics of the values that each type holds. You'll also see how to declare and use local variables (variables that exist only in a method or other small section of code), learn about the arithmetic operators that C# provides, find out how to use operators to manipulate values, and learn how to control expressions containing two or more operators.

Understanding Statements

A *statement* is a command that performs an action, such as calculating a value and storing the result, or displaying a message to a user. You combine statements to create methods. You'll learn more about methods in Chapter 3, "Writing Methods and Applying Scope," but for now, think of a method as a named sequence of statements. *Main*, which was introduced in the previous chapter, is an example of a method.

Statements in C# follow a well-defined set of rules describing their format and construction. These rules are collectively known as *syntax*. (In contrast, the specification of what statements *do* is collectively known as *semantics*.) One of the simplest and most important C# syntax rules states that you must terminate all statements with a semicolon. For example, you saw in Chapter 1 that without its terminating semicolon, the following statement won't compile:

```
Console.WriteLine("Hello World!");
```

> **Tip** C# is a "free format" language, which means that white space, such as a space character or a new line, is not significant except as a separator. In other words, you are free to lay out your statements in any style you choose. However, you should adopt a simple, consistent layout style to make your programs easier to read and understand.

The trick to programming well in any language is learning the syntax and semantics of the language and then using the language in a natural and idiomatic way. This approach makes your programs more easily maintainable. As you progress through this book, you'll see examples of the most important C# statements.

Using Identifiers

Identifiers are the names you use to identify the elements in your programs, such as namespaces, classes, methods, and variables. (You will learn about variables shortly.) In C#, you must adhere to the following syntax rules when choosing identifiers:

- You can use only letters (uppercase and lowercase), digits, and underscore characters.
- An identifier must start with a letter or an underscore.

For example, *result, _score, footballTeam*, and *plan9* are all valid identifiers, whereas *result%, footballTeam$*, and *9plan* are not.

> **Important** C# is a case-sensitive language: *footballTeam* and *FootballTeam* are not the same identifier.

Identifying Keywords

The C# language reserves 77 identifiers for its own use, and you cannot reuse these identifiers for your own purposes. These identifiers are called *keywords*, and each has a particular meaning. Examples of keywords are *class, namespace*, and *using*. You'll learn the meaning of most of the C# keywords as you proceed through this book. The following table lists the keywords:

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

Tip In the Visual Studio 2012 Code and Text Editor window, keywords are colored blue when you type them.

C# also uses the following identifiers. These identifiers are not reserved by C#, which means that you can use these names as identifiers for your own methods, variables, and classes, but you should avoid doing so if at all possible.

add	get	remove
alias	global	select
ascending	group	set
async	into	value
await	join	var
descending	let	where
dynamic	orderby	yield
from	partial	

Using Variables

A *variable* is a storage location that holds a value. You can think of a variable as a box in the computer's memory holding temporary information. You must give each variable in a program an unambiguous name that uniquely identifies it in the context in which it is used. You use a variable's name to refer to the value it holds. For example, if you want to store the value of the cost of an item in a store, you might create a variable simply called *cost* and store the item's cost in this variable. Later on, if you refer to the *cost* variable, the value retrieved will be the item's cost that you stored there earlier.

Naming Variables

You should adopt a naming convention for variables that helps you avoid confusion concerning the variables you have defined. This is especially important if you are part of a project team with several developers working on different parts of an application; a consistent naming convention helps to avoid confusion and can reduce the scope for bugs. The following list contains some general recommendations:

- Don't start an identifier with an underscore. Although this is legal in C#, it can limit the interoperability of your code with applications built by using other languages, such as Microsoft Visual Basic.
- Don't create identifiers that differ only by case. For example, do not create one variable named *myVariable* and another named *MyVariable* for use at the same time, because it is too easy to get them confused. Also, defining identifiers that differ only by case can limit the ability to reuse classes in applications developed by using other languages that are not case sensitive, such as Microsoft Visual Basic.
- Start the name with a lowercase letter.
- In a multiword identifier, start the second and each subsequent word with an uppercase letter. (This is called *camelCase notation*.)
- Don't use Hungarian notation. (Microsoft Visual C++ developers reading this book are probably familiar with Hungarian notation. If you don't know what Hungarian notation is, don't worry about it!)

For example, *score, footballTeam*, *_score*, and *FootballTeam* are all valid variable names, but only the first two are recommended.

Declaring Variables

Variables hold values. C# has many different types of values that it can store and process—integers, floating-point numbers, and strings of characters, to name three. When you declare a variable, you must specify the type of data it will hold.

You declare the type and name of a variable in a declaration statement. For example, the following statement declares that the variable named *age* holds *int* (integer) values. As always, the statement must be terminated with a semicolon.

```
int age;
```

The variable type *int* is the name of one of the *primitive* C# types, *integer*, which is a whole number. (You'll learn about several primitive data types later in this chapter.)

> **Note** Microsoft Visual Basic programmers should note that C# does not allow implicit variable declarations. You must explicitly declare all variables before you use them.

After you've declared your variable, you can assign it a value. The following statement assigns *age* the value 42. Again, you'll see that the semicolon is required.

```
age = 42;
```

The equal sign (=) is the *assignment* operator, which assigns the value on its right to the variable on its left. After this assignment, you can use the *age* variable in your code to refer to the value it holds. The next statement writes the value of the *age* variable, 42, to the console:

```
Console.WriteLine(age);
```

> **Tip** If you leave the mouse pointer over a variable in the Visual Studio 2012 Code and Text Editor window, a ScreenTip appears, telling you the type of the variable.

Working with Primitive Data Types

C# has a number of built-in types called *primitive data types*. The following table lists the most commonly used primitive data types in C# and the range of values that you can store in each.

Data type	Description	Size (bits)	Range	Sample usage

int	Whole numbers (integers)	32	-2^{31} through $2^{31} - 1$	int count;
				count = 42;

long	Whole numbers (bigger range)	64	-2^{63} through $2^{63} - 1$	long wait;
				wait = 42L;

float	Floating-point numbers	32	±1.5 x 10^{-45} through ±3.4 x 10^{38}	float away;
				away = 0.42F;

double	Double-precision (more accurate) floating-point numbers	64	±5.0 x 10^{-324} through ±1.7 x 10^{308}	double trouble;
				trouble = 0.42;

Data type	Description	Size (bits)	Range	Sample usage

decimal	Monetary values	128	28 significant figures	decimal coin;
				coin = 0.42M;

string	Sequence of characters	16 bits per character	Not applicable	string vest;
				vest = "fortytwo";

char	Single character	16	0 through $2^{16} - 1$	char grill;
				grill = 'x';

bool	Boolean	8	true or false	bool teeth;
				teeth = false;

Unassigned Local Variables

When you declare a variable, it contains a random value until you assign a value to it. This behavior was a rich source of bugs in C and C++ programs that created a variable and accidentally used it as a source of information before giving it a value. C# does not allow you to use an unassigned variable. You must assign a value to a variable before you can use it, otherwise your program will not compile. This requirement is called the *definite assignment rule*. For example, the following statements generate the compile-time error message "Use of unassigned local variable 'age'" because the *Console.WriteLine* statement attempts to display the value of an uninitialized variable:

```
int age;
Console.WriteLine(age); // compile-time error
```

Displaying Primitive Data Type Values

In the following exercise, you use a C# program named PrimitiveDataTypes to demonstrate how several primitive data types work.

Display primitive data type values

1. Start Visual Studio 2012 if it is not already running.

2. On the FILE menu, point to Open, and then click Project/Solution.

   The Open Project dialog box appears.

3. If you are using Windows 8, move to the \Microsoft Press\Visual CSharp Step By Step\Chapter 2\Windows 8\PrimitiveDataTypes folder in your Documents folder. If you are using Windows 7, move to the \Microsoft Press\Visual CSharp Step By Step\Chapter 2\Windows 7\PrimitiveDataTypes folder in your Documents folder.

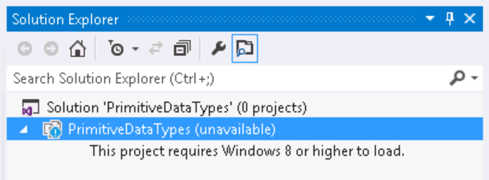
Note To avoid repetition and to save space, in subsequent exercises I will simply refer to solution paths by using a phrase of the form \Microsoft Press\Visual CSharp Step By Step\Chapter 2\Windows *X*\PrimitiveDataTypes, where *X* is either 7 or 8, depending on the operating system you are using.

4. Select the PrimitiveDataTypes solution file, and then click Open.

   The solution loads, and Solution Explorer displays the PrimitiveDataTypes project.

Note Solution file names have the .sln suffix, such as PrimitiveDataTypes.sln. A solution can contain one or more projects. Project files have the .csproj suffix. If you open a project rather than a solution, Visual Studio 2012 automatically creates a new solution file for it. This situation can be confusing if you are not aware of this feature, as it can result in you accidentally generating multiple solutions for the same project.

Tip Be sure to open the solution file in the correct folder for your operating system. If you attempt to open a solution for a Windows Store app by using Visual Studio 2012 on Windows 7, the project will fail to load. Solution Explorer will mark the project as unavailable and display the message "This project requires Windows 8 or higher to load" if you expand the project node, as shown in the following image:



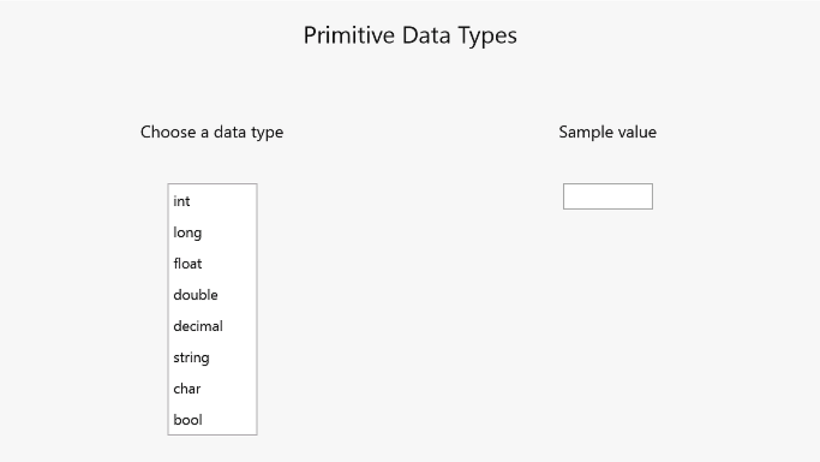
Solution 'PrimitiveDataTypes' (0 projects)
PrimitiveDataTypes (unavailable)
This project requires Windows 8 or higher to load.

If this happens, close the solution and open the version in the correct folder.

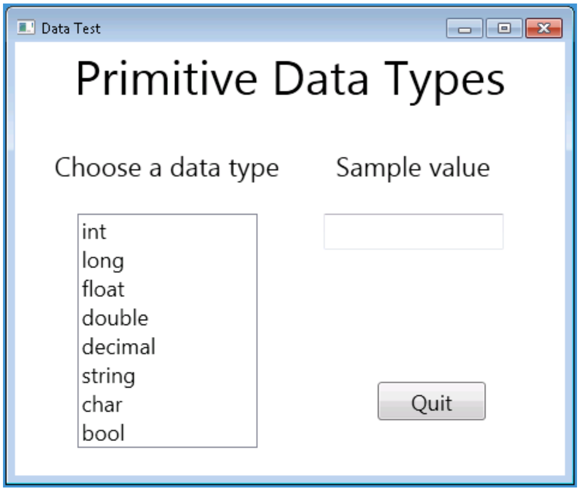
5. On the DEBUG menu, click Start Debugging.

   You might see some warnings in Visual Studio. You can safely ignore them. (You will correct them in the next exercise.)

   If you are using Windows 8, the following page is displayed:



If you are using Windows 7, the following window appears:



6. In the Choose a Data Type list, click the *string* type.

   The value "forty two" appears in the Sample Value box.

7. Click the *int* type in the list.

   The value "to do" appears in the Sample Value box, indicating that the statements to display an *int* value still need to be written.

8. Click each data type in the list. Confirm that the code for the *double* and *bool* types is not yet implemented.

9. Return to Visual Studio 2012, and on the DEBUG menu click Stop Debugging.

Note If you are using Windows 7, you can also click Quit to close the window and stop the program.

Use primitive data types in code

1. In Solution Explorer, expand the PrimitiveDataTypes project (if it is not already expanded) and then double-click MainWindow.xaml.

   Note To keep the exercise instructions simple, the forms in the Windows 8 and Windows 7 applications have the same names from now on.

   The form for the application appears in the Design View window.

   Tip If your screen is not big enough to display the entire form, you can zoom in and out in the Design View window by using Ctrl+Alt+= and Ctrl+Alt+- or by selecting the size from the zoom drop-down list in the bottom-left corner of the Design View window.

2. In the XAML pane, scroll down to locate the markup for the *ListBox* control. This control displays the list of data types in the left part of the form, and it looks like this (some of the properties have been removed from this text):

```
<ListBox x:Name="type" ... SelectionChanged="typeSelectionChanged">
  <ListBoxItem>int</ListBoxItem>
  <ListBoxItem>long</ListBoxItem>
  <ListBoxItem>float</ListBoxItem>
  <ListBoxItem>double</ListBoxItem>
  <ListBoxItem>decimal</ListBoxItem>
  <ListBoxItem>string</ListBoxItem>
  <ListBoxItem>char</ListBoxItem>
  <ListBoxItem>bool</ListBoxItem>
</ListBox>
```

   The *ListBox* control displays each data type as a separate *ListBoxItem*. When the application is running, if a user clicks an item in the list, the *SelectionChanged* event occurs (this is a little bit like the *Clicked* event that occurs when the user clicks a button, which you saw in Chapter 1). You can see that in this case, the *ListBox* invokes the *typeSelectionChanged* method. This method is defined in the MainWindow.xaml.cs file.

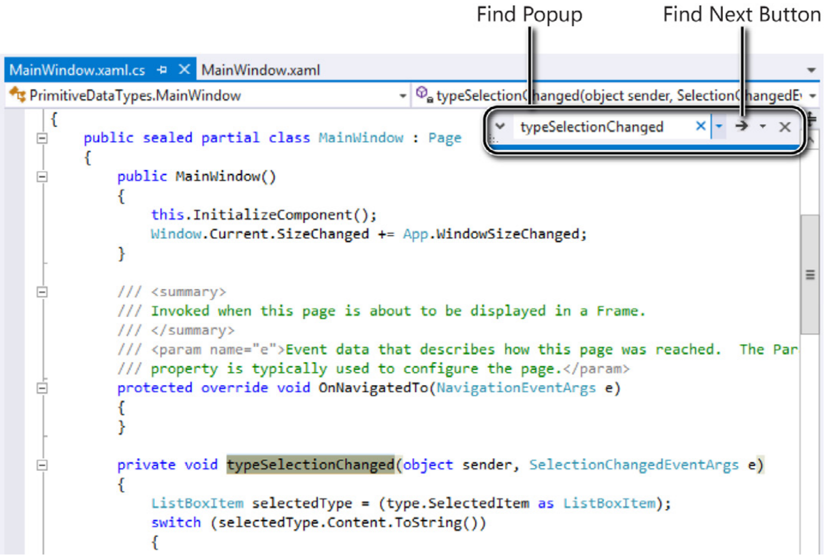
3. On the VIEW menu, click Code.

   The Code and Text Editor window opens, displaying the MainWindow.xaml.cs file.

> **Note** Remember that you can also use Solution Explorer to access the code. Click the arrow to the left of the MainWindow.xaml file to expand the node, and then double-click MainWindow.xaml.cs.

4. In the Code and Text Editor window, find the *typeSelectionChanged* method.

> **Tip** To locate an item in your project, on the EDIT menu, point to Find and Replace, and then click Quick Find. A menu opens in the top-right corner of the Code and Text Editor window. In the text box in this menu, type the name of the item you're looking for, and then click Find Next (this is the button with the right-arrow symbol next to the text box):


Find Popup
Find Next Button
MainWindow.xaml.cs
MainWindow.xaml
PrimitiveDataTypes.MainWindow
typeSelectionChanged(object sender, SelectionChangedE
typeSelectionChanged

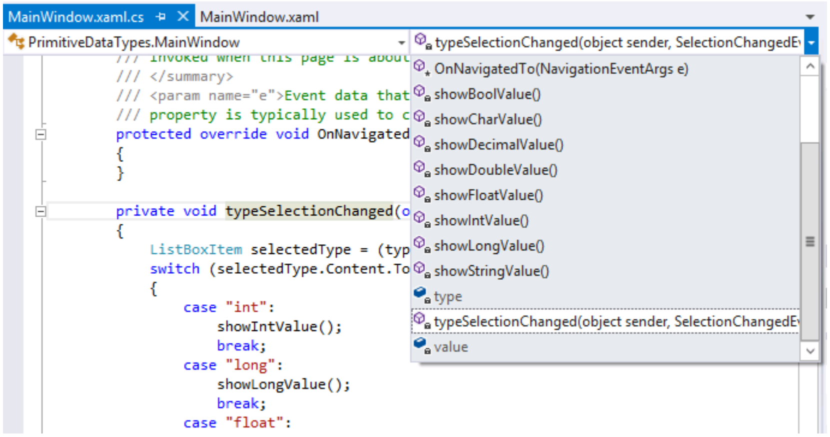
```
{
    public sealed partial class MainWindow : Page
    {
        public MainWindow()
        {
            this.InitializeComponent();
            Window.Current.SizeChanged += App.WindowSizeChanged;
        }

        /// <summary>
        /// Invoked when this page is about to be displayed in a Frame.
        /// </summary>
        /// <param name="e">Event data that describes how this page was reached.  The Par
        /// property is typically used to configure the page.</param>
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
        }

        private void typeSelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            ListBoxItem selectedType = (type.SelectedItem as ListBoxItem);
            switch (selectedType.Content.ToString())
            {
```

> By default, the search is not case sensitive. If you want to perform a case-sensitive search, click the down arrow next to the text to search for, click the drop-down arrow to the right of the text box in the shortcut menu to display additional options, and then select the Match Case check box. If you have time, you can experiment with the other options as well.
>
> You can also press Ctrl+F to display the Quick Find dialog box rather than using the EDIT menu. Similarly, you can press Ctrl+H to display the Quick Replace dialog box.

As an alternative to using the Quick Find functionality, you can also locate the methods in a class by using the class members drop-down list box above the Code and Text Editor window, on the right.


MainWindow.xaml.cs
MainWindow.xaml
PrimitiveDataTypes.MainWindow
typeSelectionChanged(object sender, SelectionChangedE
OnNavigatedTo(NavigationEventArgs e)
showBoolValue()
showCharValue()
showDecimalValue()
showDoubleValue()
showFloatValue()
showIntValue()
showLongValue()
showStringValue()
type
typeSelectionChanged(object sender, SelectionChangedE
value

```
        /// invoked when this page is about
        /// </summary>
        /// <param name="e">Event data that
        /// property is typically used to c
        protected override void OnNavigated
        {
        }

        private void typeSelectionChanged(o
        {
            ListBoxItem selectedType = (typ
            switch (selectedType.Content.To
            {
                case "int":
                    showIntValue();
                    break;
                case "long":
                    showLongValue();
                    break;
                case "float":
```

The class members drop-down list box displays all the methods in the class, together with the variables and other items that the class contains. (You will learn more about these items in later chapters.) In the drop-down list, click the *typeSelectionChanged* method, and the cursor will move directly to the *typeSelectionChanged* method in the class.

If you have programmed using another language, you can probably guess how the *typeSelectionChanged* method works; if not, then Chapter 4, "Using Decision Statements," will make this code clear. At present, all you need to understand is that when the user clicks an item in the *ListBox* control, the value of the item is passed to this method, which then uses this value to determine what happens next. For example, if the user clicks the *float* value, then this method calls another method named *showFloatValue*.

5. Scroll down through the code and find the *showFloatValue* method, which looks like this:

```
private void showFloatValue()
{
  float floatVar;
  floatVar = 0.42F;
  value.Text = floatVar.ToString();
}
```

The body of this method contains three statements. The first statement declares a variable named *floatVar* of type *float*.

The second statement assigns *floatVar* the value 0.42F. (The *F* is a type suffix specifying that 0.42 should be treated as a *float* value. If you forget the *F*, the value 0.42 is treated as a *double* and your program will not compile, because you cannot assign a value of one type to a variable of a different type without writing additional code—C# is very strict in this respect.)

The third statement displays the value of this variable in the value text box on the form. This statement requires your attention. If you remember from Chapter 1, the way in which you display an item in a text box is to set its *Text* property (you did this by using XAML in Chapter 1). You can also perform this task programmatically, which is what is going on here. Notice that you access the property of an object by using the same dot notation that you saw for running a method. (Remember *Console.WriteLine* from Chapter 1?) Also, the data that you put in the *Text* property must be a string and not a number. If you try to assign a number to the *Text* property, your program will not compile. Fortunately, the .NET Framework provides some help in the form of the *ToString* method.

Every data type in the .NET Framework has a *ToString* method. The purpose of *ToString* is to convert an object to its string representation. The *showFloatValue* method uses the *ToString* method of the *float* variable *floatVar* object to generate a string version of the value of this variable. This string can then be safely assigned to the *Text* property of the value text box. When you create your own data types and classes, you can define your own implementation of the *ToString* method to specify how your class should be represented as a string. You learn more about creating your own classes in Chapter 7, "Creating and Managing Classes and Objects."

6. In the Code and Text Editor window, locate the *showIntValue* method:

```
private void showIntValue()
{
    value.Text = "to do";
}
```

The *showIntValue* method is called when you click the *int* type in the list box.

7. Type the following two statements at the start of the *showIntValue* method, on a new line after the opening brace, as shown in bold type in the following code:

```
private void showIntValue()
{
    int intVar;
    intVar = 42;
    value.Text = "to do";
}
```

The first statement creates a variable called *intVar* that can hold an *int* value. The second statement assigns the value 42 to this variable.

8. In the original statement in this method, change the string *"to do"* to *intVar.ToString()*.

   The method should now look exactly like this:

```
private void showIntValue()
{
    int intVar;
    intVar = 42;
    value.Text = intVar.ToString();
}
```

9. On the DEBUG menu, click Start Debugging.

   The form appears again.

10. Select the *int* type in the Choose a Data Type list. Confirm that the value 42 is displayed in the Sample Value text box.

11. Return to Visual Studio, and on the DEBUG menu click Stop Debugging.

12. In the Code and Text Editor window, find the *showDoubleValue* method.

13. Edit the *showDoubleValue* method exactly as shown in bold type in the following code:

```
private void showDoubleValue()
{
    double doubleVar;
    doubleVar = 0.42;
    value.Text = doubleVar.ToString();
}
```

   This code is similar to the *showIntValue* method, except that it creates a variable called *doubleVar* that holds double values, and it is assigned the value 0.42.

14. In the Code and Text Editor window, locate the *showBoolValue* method.

15. Edit the *showBoolValue* method exactly as follows:

```
private void showBoolValue()
{
    bool boolVar;
    boolVar = false;
    value.Text = boolVar.ToString();
}
```

   Again, this code is similar to the previous examples, except that *boolVar* can only hold a Boolean value, true or false.

16. On the DEBUG menu, click Start Debugging.

17. In the Choose a Data Type list, select the *int*, *double*, and *bool* types. In each case, verify that the correct value is displayed in the Sample Value text box.
18. Return to Visual Studio, and on the DEBUG menu click Stop Debugging.

Using Arithmetic Operators

C# supports the regular arithmetic operations you learned in your childhood: the plus sign (+) for addition, the minus sign (–) for subtraction, the asterisk (*) for multiplication, and the forward slash (/) for division. The symbols +, –, *, and / are called *operators* because they "operate" on values to create new values. In the following example, the variable *moneyPaidToConsultant* ends up holding the product of 750 (the daily rate) and 20 (the number of days the consultant was employed):

```
long moneyPaidToConsultant;
moneyPaidToConsultant = 750 * 20;
```

> **Note** The values that an operator operates on are called *operands*. In the expression 750 * 20, the * is the operator, and 750 and 20 are the operands.

Operators and Types

Not all operators are applicable to all data types. The operators that you can use on a value depend on the value's type. For example, you can use all the arithmetic operators on values of type *char, int, long, float, double*, or *decimal*. However, with the exception of the plus operator, +, you can't use the arithmetic operators on values of type *string*, and you cannot use any of them with values of type *bool*. So the following statement is not allowed, because the *string* type does not support the minus operator (subtracting one string from another is meaningless):

```
// compile-time error
Console.WriteLine("Gillingham" - "Forest Green Rovers");
```

You can use the + operator to concatenate string values. You need to be careful because this can have unexpected results. For example, the following statement writes "431" (not "44") to the console:

```
Console.WriteLine("43" + "1");
```

> **Tip** The .NET Framework provides a method called *Int32.Parse* that you can use to convert a string value to an integer if you need to perform arithmetic computations on values held as strings.

You should also be aware that the type of the result of an arithmetic operation depends on the type of the operands used. For example, the value of the expression 5.0/2.0 is 2.5; the type of both operands is *double*, so the type of the result is also *double*. (In C#, literal numbers with decimal points are always *double*, not *float*, to maintain as much accuracy as possible.) However, the value of the expression 5/2 is 2. In this case, the type of both operands is *int*, so the type of the result is also *int*. C# always rounds toward zero in circumstances like this. The situation gets a little more complicated if you mix the types of the operands. For example, the expression 5/2.0 consists of an *int* and a *double*. The C# compiler detects the mismatch and generates code that converts the *int* into a *double* before performing the operation. The result of the operation is therefore a *double* (2.5). However, although this works, it is considered poor practice to mix types in this way.

C# also supports one less-familiar arithmetic operator: the *remainder*, or *modulus*, operator, which is represented by the percent sign (%). The result of *x % y* is the remainder after dividing the value *x* by the value *y*. So, for example, 9 % 2 is 1 because 9 divided by 2 is 4, remainder 1.

Note If you are familiar with C or C++, you know that you can't use the remainder operator on *float* or *double* values in these languages. However, C# relaxes this rule. The remainder operator is valid with all numeric types, and the result is not necessarily an integer. For example, the result of the expression 7.0 % 2.4 is 2.2.

Numeric Types and Infinite Values

There are one or two other features of numbers in C# that you should be aware of. For example, the result of dividing any number by zero is infinity, which is outside the range of the *int, long*, and *decimal* types; consequently, evaluating an expression such as 5/0 results in an error. However, the *double* and *float* types actually have a special value that can represent infinity, and the value of the expression 5.0/0.0 is *Infinity*. The one exception to this rule is the value of the expression 0.0/0.0. Usually, if you divide zero by anything, the result is zero, but if you divide anything by zero the result is infinity. The expression 0.0/0.0 results in a paradox—the value must be zero and infinity at the same time. C# has another special value for this situation called *NaN*, which stands for "not a number." So if you evaluate 0.0/0.0, the result is *NaN*.

NaN and *Infinity* propagate through expressions. If you evaluate 10 + *NaN*, the result is *NaN*, and if you evaluate 10 + *Infinity*, the result is *Infinity*. The one exception to this rule is the case when you multiply *Infinity* by 0. The value of the expression *Infinity* * 0 is 0, although the value of *NaN* * 0 is *NaN*.

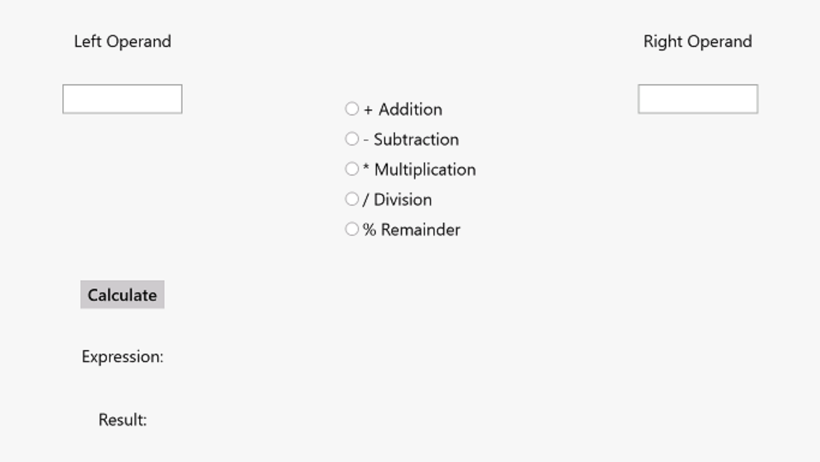
Examining Arithmetic Operators

The following exercise demonstrates how to use the arithmetic operators on *int* values.

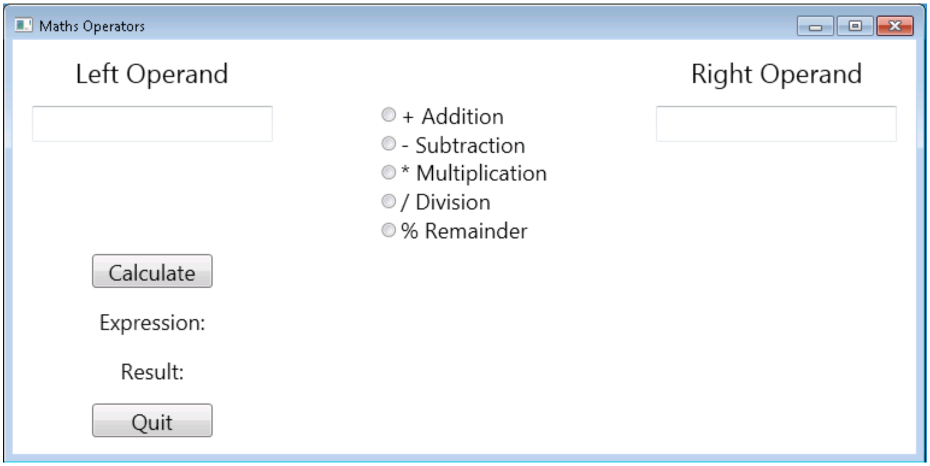
Run the MathsOperators project

1. Start Visual Studio 2012 if it is not already running.
2. Open the MathsOperators project, located in the \Microsoft Press\Visual CSharp Step By Step\Chapter 2\Windows *X*\MathsOperators folder in your Documents folder.
3. On the DEBUG menu, click Start Debugging.

   If you are using Windows 8, the following page appears:



If you are using Windows 7, the following form displays:



4. Type **54** in the Left Operand text box.

5. Type **13** in the Right Operand text box.

   You can now apply any of the operators to the values in the text boxes.

6. Click the – Subtraction button, and then click Calculate.

   The text in the Expression text box changes to 54 – 13, but the value 0 appears in the Result box; this is clearly wrong.

7. Click the / Division button, and then click Calculate.

   The text in the Expression text box changes to 54/13, and again the value 0 appears in the Result text box.

8. Click the % Remainder button, and then click Calculate.

   The text in the Expression text box changes to 54 % 13, but once again the value 0 appears in the Result text box. Test the other combinations of numbers and operators; you will find that they all currently yield the value 0.

> **Note** If you type a noninteger value into either of the operand text boxes, the application detects an error and displays the message "Input string was not in a correct format." You will learn more about how to catch and handle errors and exceptions in Chapter 6, "Managing Errors and Exceptions."

9. When you have finished, return to Visual Studio, and on the DEBUG menu click Stop Debugging (if you are using Windows 7, you can also click Quit on the MathsOperators form).

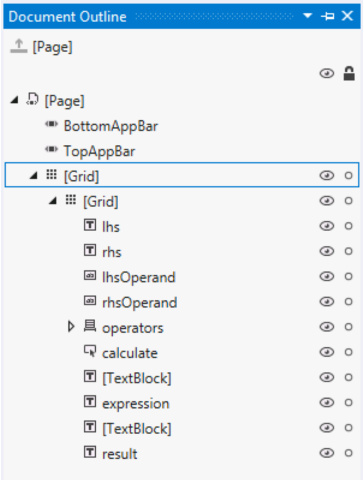
As you may have guessed, none of the calculations is currently implemented by the MathsOperators application. In the next exercise, you will correct this.

Perform calculations in the MathsOperators application

1. Display the MainWindow.xaml form in the Design View window. (Double-click the file MainWindow.xaml in the MathsOperators project in Solution Explorer.)

2. On the VIEW menu, point to Other Windows, and then click Document Outline.

   The Document Outline window appears, showing the names and types of the controls on the form. The Document Outline window provides a simple way to locate and select controls on a complex form. The controls are arranged in a hierarchy, starting with the *Page* (Windows 8) or *Window* (Windows 7) that constitutes the form. As mentioned in the previous chapter, a Windows Store app page or a WPF form contains a *Grid* control, and the other controls are placed in this *Grid*. If you expand the Grid node in the Document Outline window, the other controls appear, starting with another *Grid* (the outer *Grid* acts as a frame, and the inner *Grid* contains the controls that you see on the form). If you expand the inner *Grid*, you can see each of the controls on the form.



If you click any of these controls, the corresponding element is highlighted in the Design View window. Similarly, if you select a control in the Design View window, the corresponding control is selected in the Document Outline window (pin the Document Outline window in place by deselecting the Auto Hide button in the top-right corner of the Document Outline window to see this in action.)

3. On the form, click the two *TextBox* controls in which the user types numbers. In the Document Outline window, verify that they are named *lhsOperand* and *rhsOperand*.

   When the form runs, the *Text* property of each of these controls holds the values that the user enters.

4. Toward the bottom of the form, verify that the *TextBlock* control used to display the expression being evaluated is named *expression* and that the *TextBlock* control used to display the result of the calculation is named *result*.

5. Close the Document Outline window.

6. On the VIEW menu, click Code to display the code for the MainWindow.xaml.cs file in the Code and Text Editor window.

7. In the Code and Text Editor window, locate the *addValues* method. It looks like this:

```
private void addValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome;
    // TODO: Add rhs to lhs and store the result in outcome
    expression.Text = lhsOperand.Text + " + " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
```

The first statement in this method declares an *int* variable called *lhs* and initializes it with the integer corresponding to the value typed by the user in the *lhsOperand* text box. Remember that the *Text* property of a *TextBox* control contains a string, but *lhs* is an *int*, so you must convert this string to an integer before you can assign it to *lhs*. The *int* data type provides the *int.Parse* method, which does precisely this.

The second statement declares an *int* variable called *rhs* and initializes it to the value in the *rhsOperand* text box after converting it to an *int*.

The third statement declares an *int* variable called *outcome*.

A comment stating that you need to add *rhs* to *lhs* and store the result in *outcome* follows. This is the missing bit of code that you need to implement, which you will do in the next step.

The fifth statement concatenates three strings indicating the calculation being performed (using the plus operator, +) and assigns the result to the *expression.Text* property. This causes the string to appear in the Expression text box on the form.

The final statement displays the result of the calculation by assigning it to the *Text* property of the Result text box. Remember that the *Text* property is a string, and the result of the calculation is an *int*, so you must convert the *int* to a string before assigning it to the *Text* property. Recall that this is what the *ToString* method of the *int* type does.

8. Underneath the comment in the middle of the *addValues* method, add the statement shown below in bold:

```
private void addValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome;
    // TODO: Add rhs to lhs and store the result in outcome
    outcome = lhs + rhs;
    expression.Text = lhsOperand.Text + " + " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
```

This statement evaluates the expression *lhs* + *rhs* and stores the result in *outcome*.

9. Examine the *subtractValues* method. You should see that it follows a similar pattern, and you need to add the statement to calculate the result of subtracting *rhs* from *lhs* and storing it in *outcome*. Add the statement shown below in bold to this method:

```
private void subtractValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome;
    // TODO: Subtract rhs from lhs and store the result in outcome
    outcome = lhs - rhs;
    expression.Text = lhsOperand.Text + " - " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
```

10. Examine the *mutiplyValues*, *divideValues*, and *remainderValues* methods. Again, they all have the crucial statement that performs the specified calculation missing. Add the appropriate statements to these methods (shown below in bold).

```
private void multiplyValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Multiply lhs by rhs and store the result in outcome
    outcome = lhs * rhs;
    expression.Text = lhsOperand.Text + " * " + rhsOperand.Text;
    result.Text = outcome.ToString();
}

private void divideValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Divide lhs by rhs and store the result in outcome
    outcome = lhs / rhs;
    expression.Text = lhsOperand.Text + " / " + rhsOperand.Text;
    result.Text = outcome.ToString();
}

private void remainderValues()
{
    int lhs = int.Parse(lhsOperand.Text);
    int rhs = int.Parse(rhsOperand.Text);
    int outcome = 0;
    // TODO: Work out the remainder after dividing lhs by rhs and store the result
    outcome = lhs % rhs;
    expression.Text = lhsOperand.Text + " % " + rhsOperand.Text;
    result.Text = outcome.ToString();
}
```

Test the MathsOperators application

1. On the DEBUG menu, click Start Debugging to build and run the application.
2. Type **54** in the Left Operand text box, type **13** in the Right Operand text box, click the + Addition button, and then click Calculate.

   The value 67 should appear in the Result text box.
3. Click the – Subtraction button and then click Calculate. Verify that the result is now 41.
4. Click the * Multiplication button and then click Calculate. Verify that the result is now 702.
5. Click the / Division button and then click Calculate. Verify that the result is now 4.

   In real life, 54/13 is 4.153846 recurring, but this is not real life—this is C# performing integer division. When you divide one integer by another integer, the answer you get back is an integer, as explained earlier.
6. Click the % Remainder button and then click Calculate. Verify that the result is now 2.

   When dealing with integers, the remainder after dividing 54 by 13 is 2; (54 – ((54/13) * 13)) is 2. This is because the calculation rounds down to an integer at each stage. (My school mathematics teacher would be horrified to be told that (54/13) * 13 does not equal 54!)
7. Return to Visual Studio and stop debugging (or click Quit if you are using Windows 7).

Controlling Precedence

Precedence governs the order in which an expression's operators are evaluated. Consider the following expression, which uses the + and * operators:

```
2 + 3 * 4
```

This expression is potentially ambiguous: do you perform the addition first or the multiplication? The order of the operations matters because it changes the result:

- If you perform the addition first, followed by the multiplication, the result of the addition (2 + 3) forms the left operand of the * operator, and the result of the whole expression is 5 * 4, which is 20.
- If you perform the multiplication first, followed by the addition, the result of the multiplication (3 * 4) forms the right operand of the + operator, and the result of the whole expression is 2 + 12, which is 14.

In C#, the multiplicative operators (*, /, and %) have precedence over the additive operators (+ and –), so in expressions such as 2 + 3 * 4, the multiplication is performed first, followed by the addition. The answer to 2 + 3 * 4 is therefore 14.

You can use parentheses to override precedence and force operands to bind to operators in a different way. For example, in the following expression, the parentheses force the 2 and the 3 to bind to the + operator (making 5), and the result of this addition forms the left operand of the * operator to produce the value 20:

```
(2 + 3) * 4
```

Note The term *parentheses* or *round brackets* refers to (). The term *braces* or *curly brackets* refers to { }. The term *square brackets* refers to [ ].

Using Associativity to Evaluate Expressions

Operator precedence is only half the story. What happens when an expression contains different operators that have the same precedence? This is where associativity becomes important. *Associativity* is the direction (left or right) in which the operands of an operator are evaluated. Consider the following expression that uses the / and * operators:

```
4 / 2 * 6
```

At first glance, this expression is potentially ambiguous. Do you perform the division first or the multiplication? The precedence of both operators is the same (they are both multiplicative), but the order in which the operators in the expression are applied is important because you can get two different results:

- If you perform the division first, the result of the division (4/2) forms the left operand of the * operator, and the result of the whole expression is (4/2) * 6, or 12.
- If you perform the multiplication first, the result of the multiplication (2 * 6) forms the right operand of the / operator, and the result of the whole expression is 4/(2 * 6), or 4/12.

In this case, the associativity of the operators determines how the expression is evaluated. The * and / operators are both left-associative, which means that the operands are evaluated from left to right. In this case, 4/2 will be evaluated before multiplying by 6, giving the result 12.

Associativity and the Assignment Operator

In C#, the equal sign, =, is an operator. All operators return a value based on their operands. The assignment operator = is no different. It takes two operands: the operand on its right side is evaluated and then stored in the operand on its left side. The value of the assignment operator is the value that was assigned to the left operand. For example, in the following assignment statement, the value returned by the assignment operator is 10, which is also the value assigned to the variable *myInt*:

```
int myInt;
myInt = 10; // value of assignment expression is 10
```

At this point, you might be thinking that this is all very nice and esoteric, but so what? Well, because the assignment operator returns a value, you can use this same value with another occurrence of the assignment statement, like this:

```
int myInt;
int myInt2;
myInt2 = myInt = 10;
```

The value assigned to the variable *myInt2* is the value that was assigned to *myInt*. The assignment statement assigns the same value to both variables. This technique is useful if you want to initialize several variables to the same value. It makes it very clear to anyone reading your code that all the variables must have the same value:

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

From this discussion, you can probably deduce that the assignment operator associates from right to left. The rightmost assignment occurs first, and the value assigned propagates through the variables from right to left. If any of the variables previously had a value, it is overwritten by the value being assigned.

You should treat this construct with caution, however. One frequent mistake that new C# programmers make is to try to combine this use of the assignment operator with variable declarations. For example, you might expect the following code to create and initialize three variables with the same value (10):

```
int myInt, myInt2, myInt3 = 10;
```

This is legal C# code (because it compiles). What it does is declare the variables *myInt, myInt2*, and *myInt3*, and initialize *myInt3* with the value 10. However, it does not initialize *myInt* or *myInt2*. If you try to use *myInt* or *myInt2* in an expression such as this:

```
myInt3 = myInt / myInt2;
```

the compiler generates the following errors:

```
Use of unassigned local variable 'myInt'
Use of unassigned local variable 'myInt2'
```

Incrementing and Decrementing Variables

If you want to add 1 to a variable, you can use the + operator:

```
count = count + 1;
```

However, adding 1 to a variable is so common that C# provides its own operator just for this purpose: the ++ operator. To increment the variable *count* by 1, you can write the following statement:

```
count++;
```

Similarly, C# provides the –– operator that you can use to subtract 1 from a variable, like this:

```
count--;
```

The ++ and –– operators are *unary* operators, meaning that they take only a single operand. They share the same precedence and are both left-associative.

Prefix and Postfix

The increment, ++, and decrement, ––, operators are unusual in that you can place them either before or after the variable. Placing the operator symbol before the variable is called the *prefix form* of the operator, and using the operator symbol after the variable is called the *postfix form*. Here are examples:

```
count++; // postfix increment
++count; // prefix increment
count--; // postfix decrement
--count; // prefix decrement
```

Whether you use the prefix or postfix form of the ++ or –– operator makes no difference to the variable being incremented or decremented. For example, if you write *count*++, the value of *count* increases by 1, and if you write ++*count*, the value of *count* also increases by 1. Knowing this, you're probably wondering why there are two ways to write the same thing. To understand the answer, you must remember that ++ and –– are operators and that all operators are used to evaluate an expression that has a value. The value returned by *count*++ is the value of *count* before the increment takes place, whereas the value returned by ++*count* is the value of *count* after the increment takes place. Here is an example:

```
int x;
x = 42;
Console.WriteLine(x++); // x is now 43, 42 written out
x = 42;
Console.WriteLine(++x); // x is now 43, 43 written out
```

The way to remember which operand does what is to look at the order of the elements (the operand and the operator) in a prefix or postfix expression. In the expression *x*++, the variable *x* occurs first, so its value is used as the value of the expression before *x* is incremented. In the expression ++*x*, the operator occurs first, so its operation is performed before the value of *x* is evaluated as the result.

These operators are most commonly used in *while* and *do* statements, which are presented in Chapter 5, "Using Compound Assignment and Iteration Statements." If you are using the increment and decrement operators in isolation, stick to the postfix form and be consistent.

Declaring Implicitly Typed Local Variables

Earlier in this chapter, you saw that you declare a variable by specifying a data type and an identifier, like this:

```
int myInt;
```

It was also mentioned that you should assign a value to a variable before you attempt to use it. You can declare and initialize a variable in the same statement, like this:

```
int myInt = 99;
```

Or you can even do it like this, assuming that *myOtherInt* is an initialized integer variable:

```
int myInt = myOtherInt * 99;
```

Now, remember that the value you assign to a variable must be of the same type as the variable. For example, you can assign an *int* value only to an *int* variable. The C# compiler can quickly work out the type of an expression used to initialize a variable and indicate if it does not match the type of the variable. You can also ask the C# compiler to infer the type of a variable from an expression and use this type when declaring the variable by using the *var* keyword in place of the type, like this:

```
var myVariable = 99;
var myOtherVariable = "Hello";
```

The variables *myVariable* and *myOtherVariable* are referred to as *implicitly typed* variables. The *var* keyword causes the compiler to deduce the type of the variables from the types of the expressions used to initialize them. In these examples, *myVariable* is an *int*, and *myOtherVariable* is a *string*. However, it is important for you to understand that this is a convenience for declaring variables only, and that after a variable has been declared you can assign only values of the inferred type to it—you cannot assign *float*, *double*, or *string* values to *myVariable* at a later point in your program, for example. You should also understand that you can use the *var* keyword only when you supply an expression to initialize a variable. The following declaration is illegal and causes a compilation error:

```
var yetAnotherVariable; // Error - compiler cannot infer type
```

> **Important** If you have programmed with Visual Basic in the past, you might be familiar with the *Variant* type, which you can use to store any type of value in a variable. I emphasize here and now that you should forget everything you ever learned when programming with Visual Basic about *Variant* variables. Although the keywords look similar, *var* and *Variant* mean totally different things. When you declare a variable in C# using the *var* keyword, the type of values that you assign to the variable *cannot change* from that used to initialize the variable.

If you are a purist, you are probably gritting your teeth at this point and wondering why on earth the designers of a neat language such as C# should allow a feature such as *var* to creep in. After all, it sounds like an excuse for extreme laziness on the part of programmers and can make it more difficult to understand what a program is doing or track down bugs (and it can even easily introduce new bugs into your code). However, trust me that *var* has a very valid place in C#, as you will see when you work through many of the following chapters. However, for the time being, we will stick to using explicitly typed variables except for when implicit typing becomes a necessity.

Summary

In this chapter, you have seen how to create and use variables, and you have learned about some of the common data types available for variables in C#. You have learned about identifiers. In addition, you have used a number of operators to build expressions, and you have learned how the precedence and associativity of operators determine how expressions are evaluated.

- If you want to continue to the next chapter

  Keep Visual Studio 2012 running, and turn to Chapter 3.

- If you want to exit Visual Studio 2012 now

  On the FILE menu, click Exit. If you see a Save dialog box, click Yes and save the project.

Chapter 2 Quick Reference

To	Do this
Declare a variable	Write the name of the data type, followed by the name of the variable, followed by a semicolon. For example: `int outcome;`
Declare a variable and give it an initial value	Write the name of the data type, followed by the name of the variable, followed by the assignment operator and the initial value. Finish with a semicolon. For example: `int outcome = 99;`
Change the value of a variable	Write the name of the variable on the left, followed by the assignment operator, followed by the expression calculating the new value, followed by a semicolon. For example: `outcome = 42;`

Generate a string representation of the value in a variable	Call the ToString method of the variable. For example:
	`int intVar = 42;`
	`string stringVar = intVar.ToString();`

Convert a string to an int	Call the System.Int32.Parse method. For example:
	`string stringVar = "42";`
	`int intVar = System.Int32.Parse(stringVar);`

Override the precedence of an operator	Use parentheses in the expression to force the order of evaluation. For example: `(3 + 4) * 5`
Assign the same value to several variables	Use an assignment statement that lists all the variables. For example: `myInt4 = myInt3 = myInt2 = myInt = 10;`
Increment or decrement a variable	Use the ++ or -- operator. For example: `count++;`

Index

About the Author

JOHN SHARP is a principal technologist working for Content Master Ltd in the United Kingdom. He gained an honors degree in Computing from Imperial College, London. He has been developing software and writing training courses, guides, and books for over 25 years. John has experience in a wide range of technologies, from database systems and UNIX through to C, C++, and C# applications for the .NET Framework, together with Java and JavaScript development. Apart from six editions of *C# Step By Step*, he has authored several other books, including *Windows Communication Foundation Step By Step* and the *J# Core Reference*.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press