Microsoft

developer

// Step by step

Microsoft
Visual C# 2013

Intermediate

John Sharp

// Step by step

Your hands-on guide to Visual C# fundamentals

Expand your expertise—and teach yourself the fundamentals of Microsoft Visual C# 2013. If you have previous programming experience but are new to Visual C# 2013, this tutorial delivers the step-by-step guidance and coding exercises you need to master core topics and techniques.

Discover how to:

- Create and debug C# applications in Visual Studio 2013
- Work with variables, identifiers, statements, operators, and methods
- Create interfaces and define abstract classes
- Write robust code that can catch and handle exceptions
- Display and edit data using data binding with the Microsoft ADO.NET Entity Framework
- Respond to user input and touchscreen gestures
- Handle events arising from multiple sources
- Develop your first Windows 8.1 apps

microsoft.com/mspress

9 0 0 0 0

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

Programming/Microsoft Visual C#

9 780735 681835

About the Author

John Sharp, a principal technologist at an industry research company, has extensive experience with a range of programming technologies. He is an expert on developing applications with the Microsoft .NET Framework, and the author of *Windows Communication Foundation 4 Step by Step*.

Practice Files + Code
Available at:
http://aka.ms/VC2013SbS/files

Companion eBook
See the instruction page at the back of the book

Microsoft Press

Celebrating 30 years!

Microsoft Visual C# 2013 Step by Step

John Sharp

Contents at a glance

Contents

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our
books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

PART II   UNDERSTANDING THE C# OBJECT MODEL

Chapter 22 Operator overloading   511

PART IV   BUILDING PROFESSIONAL WINDOWS 8.1 APPLICATIONS WITH C#

Chapter 23 Improving throughput by using tasks   537

Chapter 24 Improving response time by performing asynchronous operations   581

Chapter 25 Implementing the user interface for a Windows Store app   623

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our
books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

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 improve 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 made 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.

After listening to feedback from developers, Microsoft modified some aspects of the way in which the user interface works and released a technical preview of Windows 8.1 containing these changes. At the same time, Microsoft released a preview edition of Visual Studio 2013, containing incremental changes to Visual Studio 2012 and adding new features that help to further improve programmer productivity. Although many of the updates to Visual Studio are small, and there have been no changes to the C# language in this release, we felt that the modifications to the way in which Windows 8.1 handles the user interface would make it beneficial to perform a similar incremental update to this book. The result is this volume.

Note This book is based on the Technical Preview of Visual Studio 2013. Consequently, some features of the IDE might change in the final release of the software.

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 2013 and the .NET Framework version 4.5.1. By the time you complete this book, you will have a thorough understanding of C# and will have used it to build responsive and scalable applications that can run by using the Windows operating system.

You can build and run C# 5.0 applications on Windows 7, Windows 8, and Windows 8.1, although the user interfaces provided by Windows 7 and Windows 8 have some significant differences. Additionally, Windows 8.1 has modified some parts of the user interface model, and applications designed to take advantage of these changes might not run on Windows 8. Consequently, Parts I to III of this book provide exercises and working examples that run using Windows 7, Windows 8, and Windows 8.1. Part IV focuses on the application development model used by Windows 8.1, 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#, but not completely new to programming. As such, it concentrates primarily on the C# language. This book is not intended to provide detailed coverage of the multitude of technologies available for building enterprise-level applications for Windows, such as ADO.NET, ASP.NET, Windows Communication Foundation, or 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* by George Shepherd, *Microsoft ADO.NET 4 Step By Step* by Tim Patrick, and *Microsoft Windows Communication Foundation 4 Step By Step* by John Sharp.

Organization of this book

This book is divided into four sections:

■ Part I, "Introducing Microsoft Visual C# and Microsoft Visual Studio 2013," 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 Windows 8.1 applications with C#," describes the Windows 8.1 programming model, and how you can use C# to build interactive applications for this new model.

Note Although Part IV is aimed at Windows 8.1, many of the concepts described in Chapters 23 and 24 are also applicable to Windows 8 and 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.

If you are	Follow these steps
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 2013, 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 2013, and then concentrate on Chapters 7 through 22. 3. For information about building scalable Windows 8.1 applications, read Part IV.
Switching from Visual Basic to C#	1. Install the practice files as described in the upcoming section, "Code Samples." 2. Work through the chapters in Parts I, II, and III sequentially. 3. For information about building Windows 8.1 applications, read Part IV. 4. Read the Quick Reference sections at the end of the chapters for information about specific C# and Visual Studio 2013 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 by using conventions designed to make the information readable and easy to follow.

- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.

- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.

- Text that you type (apart from code blocks) appears in bold.

- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.

- 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 8.1 (x86 and x64), Windows Server 2008 R2 SP1 (x64), Windows Server 2012 (x64), or Windows Server 2012 R2 (x64).

> **!** **Important** The Windows Store templates for Visual Studio 2013 are not available on Windows 8, Windows 7, Windows Server 2012, or Windows Server 2008 R2. If you want to use these templates or perform the exercises that build Windows Store apps, you must be running Windows 8.1 or Windows Server 2012 R2.

- Visual Studio 2013 (any edition except Visual Studio Express for Windows 8.1).

> **!** **Important** You can use Visual Studio Express 2013 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 8.1, 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 2013.

Code samples

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

http://aka.ms/VC2013SbS/files

Follow the instructions to download the 9780735681835_files.zip file.

Note In addition to the code samples, your system should have Visual Studio 2013 installed. If available, install the latest service packs for Windows and Visual Studio.

Installing the code samples

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

1. Unzip the C#_SBS.zip file that you downloaded from the book's website into your Documents folder.

2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the C#_SBS.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 2013 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 8.1, 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".

> **Important** If you are using Windows 8, Windows Server 2012 or Windows Server 2008 R2, follow the instructions for Windows 7. If you are using Windows Server 2012 R2, follow the instructions for Windows 8.1.

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 (+ − * / %).
Chapter 3	
Methods	In this project, you'll re-examine the code in the previous project and investigate how it uses methods to structure the code.

Project	Description
DailyRate	This project walks you through writing your own methods, running the methods, and stepping through the method calls by using the Visual Studio 2013 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 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	

Project	Description
StructsAndEnums	This project defines a *struct* type to represent a calendar date.
Chapter 10	
Cards	This project shows how to use arrays to model hands of cards in a card game.
Chapter 11	
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.

Project	Description
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.
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	

Project	Description
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.
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.

Project	Description
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 so that the user interface can update customer information and send these changes back to the data source.
ViewModel	This version of the Customers project separates the user interface from the logic that accesses the data source by implementing the Model-View-ViewModel pattern.
Search	This project implements the Windows 8.1 Search contract. A user can search for customers by first name or last name.
Chapter 27	
Web Service	This solution includes a web application that provides an ASP.NET Web API web service that the Customers application uses to retrieve customer data from a SQL Server database. The web service uses an entity model created 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

process of getting this edition of the book ready to go to print. Without his efforts you might have been reading this edition just as the next edition of Windows emerged.

Next, Mike Sumsion and Paul Barnes, my esteemed colleagues at Content Master, who performed sterling work reviewing the material for the original versions of 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 very swift technical review of this edition. 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 the editors for correcting my grammar, fixing my spelling, and generally making my material much easier to understand.

Finally, I would like to thank my wife and cricketing companion, Diana, for not frowning too much when I said I was about to start work on an updated edition of this book. She has now become used to my cranky mutterings while I debug code, and the numerous "d'ohs" that I emit when I realize the crass mistakes I have made.

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://aka.ms/VC2013SbS/errata

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

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

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

We want to hear from you

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

http://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*

Welcome to C#

After completing this chapter, you will be able to:

- Use the Microsoft Visual Studio 2013 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 2013, the programming environment, and tool-set designed to help you build applications for Microsoft Windows. Visual Studio 2013 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 2013 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 2013 environment

Visual Studio 2013 is a tool-rich programming environment containing the functionality that you need to create large or small C# projects running on Windows 7, Windows 8, and Windows 8.1. 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 2013 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 2013

- If you are using Windows 8.1 or Windows 8, on the Start screen, type **Visual Studio**, and then, in the Search results pane, click Visual Studio 2013.

Visual Studio 2013 starts and displays the Start page, similar to the following (your Start page might be different, depending on the edition of Visual Studio 2013 you are using).
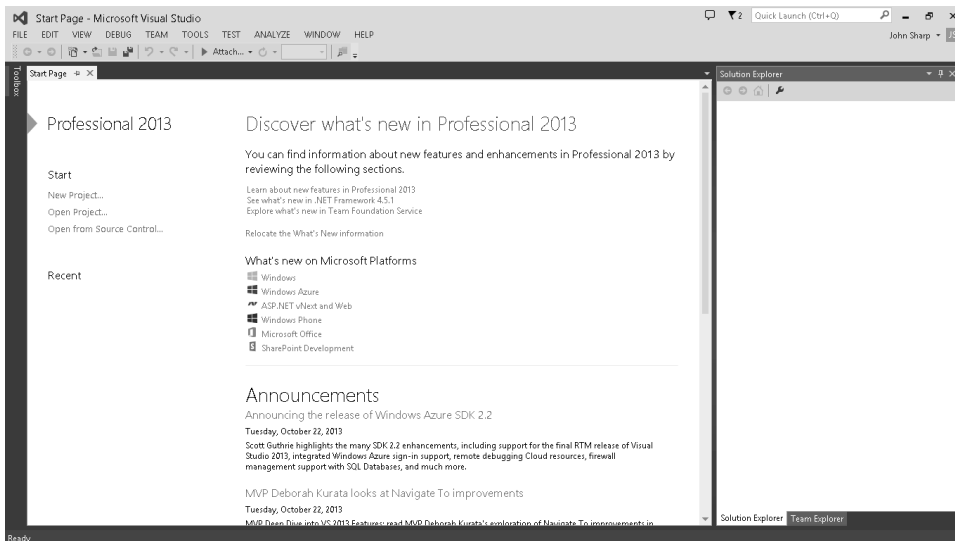
- If you are using Windows 7, perform the following operations to start Visual Studio 2013:

  a. On the Windows taskbar, click the Start button, click All Programs, and then click the Microsoft Visual Studio 2013 program group.

  b. In the Microsoft Visual Studio 2013 program group, click Visual Studio 2013.

   Visual Studio 2013 starts and displays the Start page.

> 📝 **Note** To avoid repetition and save space, throughout this book, I will simply state "Start Visual Studio" when you need to open Visual Studio 2013, 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, in the Templates section, 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.
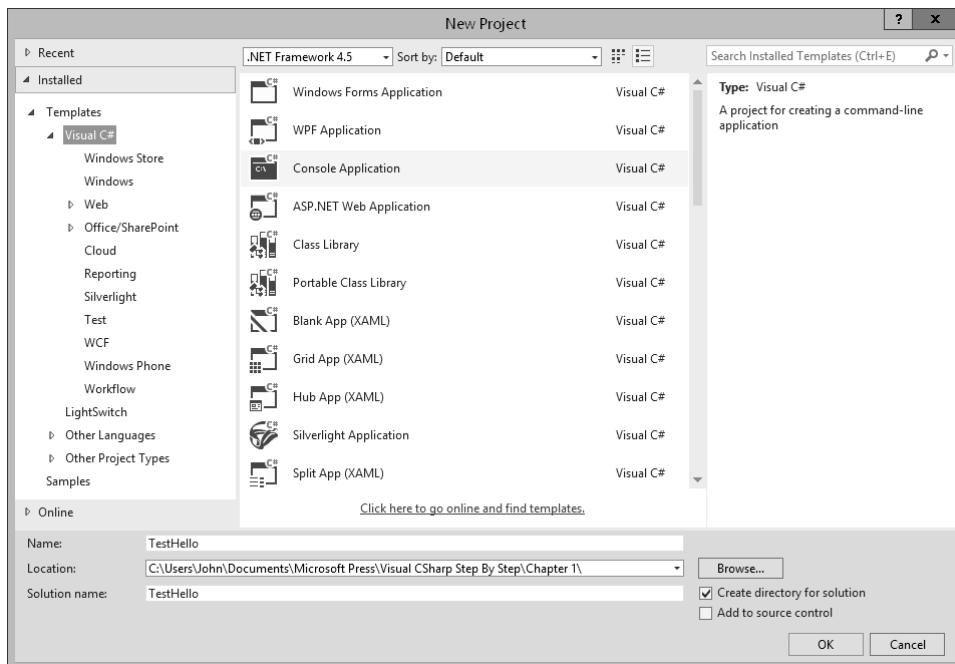
c. In the Location box, type **C:\Users*YourName*\Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 1**. Replace the text *YourName* in this path with your Windows user name.

Note To avoid repetition and save space, throughout the rest of this book, I will re-fer to the path C:\Users*YourName*\Documents simply as your Documents folder.

Tip If the folder you specify does not exist, Visual Studio 2013 creates it for you.

d. In the Name box, type **TestHello** (type over 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:


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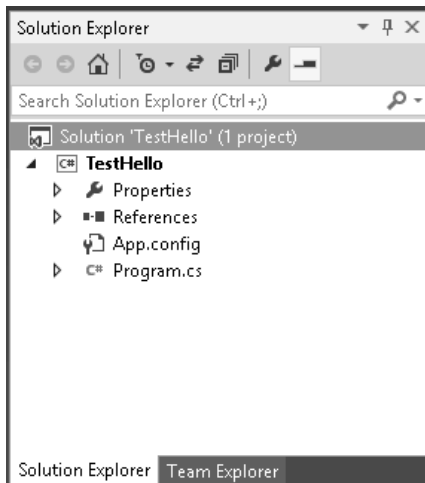
Solution Explorer displays the names of the files associated with the project, among other items. You can also double-click a file name in the Solution Explorer pane to bring that source file to the foreground in the Code and Text Editor window.

Before writing the code, examine the files listed in Solution Explorer, which Visual Studio 2013 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 2013 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. You must write all the source code in a single project 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 in your Documents folder.

- **Properties**   This is a folder in the TestHello project. If you expand it (click the arrow next to 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 Microsoft .NET Framework, these libraries are called *assemblies*. Developers use assemblies to package useful functionality that they have written so that 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 will see the default set of references that Visual Studio 2013 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 2013. 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 might not always be present. You can specify settings that your application can use at run time 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 2013 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 within 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 might 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.

```
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)
        {
            Cons
        }                   Console              class System.Console
    }                       ConsoleCancelEventArgs   Represents the standard input, output, and error streams for console applications. This class cannot be
}                           ConsoleCancelEventHandler  inherited.
                            ConsoleColor
                            ConsoleKey
                            ConsoleKeyInfo
                            ConsoleModifiers
                            ConsoleSpecialKey
                            const
```

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 following *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. You can use each version of the *WriteLine* method 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 parentheses, (and), and curly brackets, { 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, "Implementing properties to access fields")
	Class (discussed in Chapter 7)
	Struct (discussed in Chapter 9)
	Enum (discussed in Chapter 9)
	Extension method (discussed in Chapter 12)
	Interface (discussed in Chapter 13, "Creating interfaces and defining abstract classes")
	Delegate (discussed in Chapter 17, "Introducing generics")
	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, and they are ignored by the compiler but are very useful for developers because they help document what a program is actually doing. Take for instance the following 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.

Tip To go directly to the line that caused the error, you can double-click an item in the Error List window. 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 it.

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; the program waits for you to press any key, as shown in the following graphic:

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 (meaning that it's the window that's currently active), and then press Enter.

The command window closes, and you return to the Visual Studio 2013 programming environment.

4.   In Solution Explorer, click the *TestHello* project (not the solution), and then, on the Solution Explorer toolbar, click the Show All Files button. Be aware that you might need to click the double-arrow button on the right edge of the Solution Explorer toolbar to make this button appear.
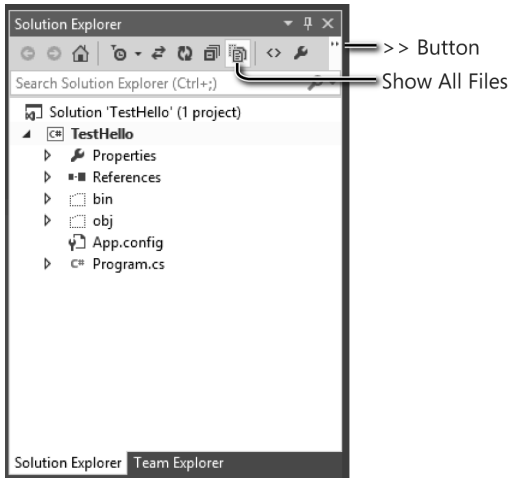
Entries named *bin* and *obj* appear above the *Program.cs* file. These entries correspond directly to folders named bin and obj in the project folder (Microsoft Press\Visual CSharp Step By Step\Chapter 1\TestHello\TestHello). Visual Studio creates these folders when you build your application; they 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 named Release.

6. In Solution Explorer, expand the Debug folder.

Several more items appear, including a file named *TestHello.exe*. This is the compiled program, which is the file that runs when you click Start Without Debugging on the Debug menu. The other files contain information that is used by Visual Studio 2013 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 2013 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 within a namespace. For example, the *Console* class lives within 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 2013 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 need 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 2013 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*, IntelliSense displays the names of all the items in the *System* namespace.

5. On the Build menu, click Build Solution.

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

6. Run the application to ensure that it still works by clicking Start Without Debugging on the Debug menu.

7. When the program runs and displays "Hello World!", in the console window, press Enter to return to Visual Studio 2013.

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 were to update 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 might be surprised to see that *mscorlib.dll* is not included in this list. The reason for this is that all .NET Framework applications must use this assembly because it contains fundamental runtime functionality. The References folder lists only the optional assemblies; you can add or remove assemblies from this folder as necessary.

To add references for additional assemblies to a project, right-click the References folder and then, in the shortcut menu that appears, click 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 2013 to create and run a basic console application. The Visual Studio 2013 programming environment also contains everything you need to create graphical applications for Windows 7, Windows 8, and Windows 8.1. You can design the user interface (UI) of a Windows application interactively. Visual Studio 2013 then generates the program statements to implement the user interface you've designed.

Visual Studio 2013 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 UI. 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 by using Visual Studio 2013. This program displays 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 and Windows 8, Visual Studio 2013 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 8.1. However, Windows 8 and Windows 8.1 provide a new flavor of UI, referred to as the "Windows Store" style. Applications that use this style of UI 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 tablets 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 UI to match the orientation (it can switch from landscape to portrait mode, for example). If you have installed Visual Studio 2013 on a Windows 8.1 computer, you are provided with an additional set of templates for building Windows Store apps. *However, these templates are dependent on features provided by Windows 8.1, so if you are running Windows 8, the Windows Store templates are not available.*

To cater to Windows 7, Windows 8, and Windows 8.1 developers, I have provided instructions in many of the exercises for using the WPF templates. If you are running Windows 7 or Windows 8 you should follow the Windows 7 instructions. If you want to use the Windows Store style UI, you should follow the Windows 8.1 instructions. Of course, you can follow the Windows 7 and Windows 8 instructions to use the WPF templates on Windows 8.1 if you prefer.

If you want more information about the specifics of writing Windows 8.1 applications, the final few chapters in Part IV of this book provide more detail and guidance.

Create a graphical application in Visual Studio 2013

- If you are using Windows 8.1, perform the following operations to create a new graphical application:

  a. Start Visual Studio 2013 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, in the Installed Templates section, 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, which is 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 in your Documents folder.

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

  g. In the Solution box, 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, as depicted in the illustration that follows. 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.

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 some code. For example, if you expand the MainPage.xaml folder, you will find a C# file named Main-Page.xaml.cs. This file is where you add the code that runs when the UI defined by the MainPage.xaml file is displayed.

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

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

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 UI for the application, and you will examine the XAML code that this layout generates.

```
51.43% ▼  ▦ ▦ ✛ ◀
☐ Design   ↑↓   ☒ XAML  ▤
⊟ <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>
100 %  ▼ ◀
```
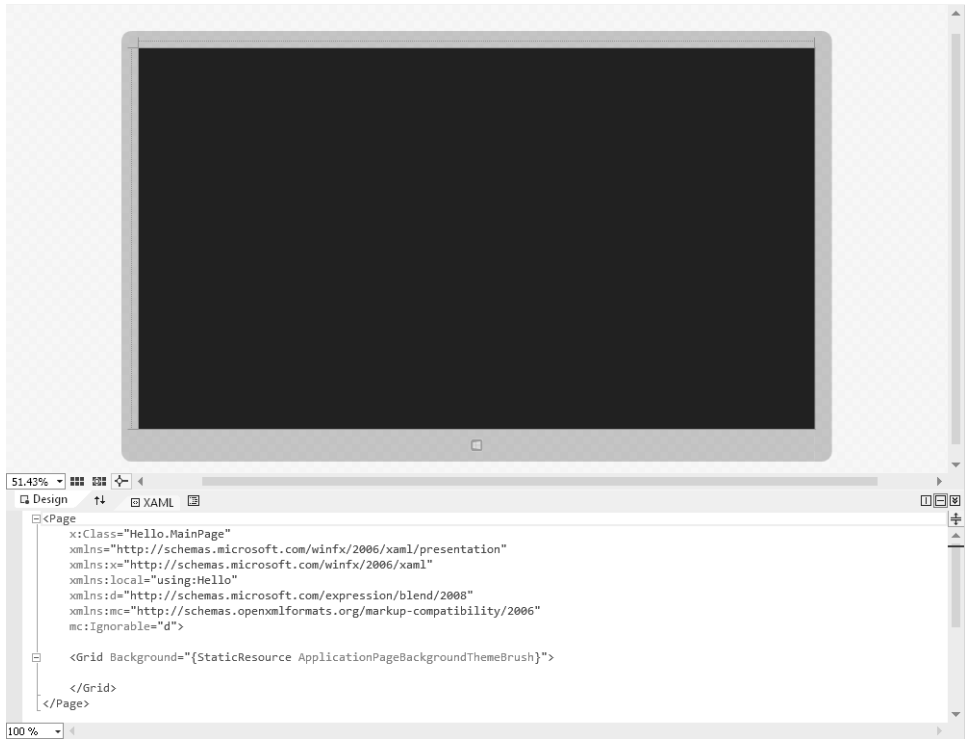
■ If you are using Windows 8 or Windows 7, perform the following tasks:

a. Start Visual Studio 2013 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, in the Installed Templates section, 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 box refers to the \Microsoft Press\Visual CSharp Step By Step\ Chapter 1 folder in your Documents folder.

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

g. In the Solution box, ensure that Create New Solution is selected, and then 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 because the functionality that these styles embody is specific to Windows 8.1. 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.

h. In Solution Explorer, double-click MainWindow.xaml to display the contents of this file in the Design View window.
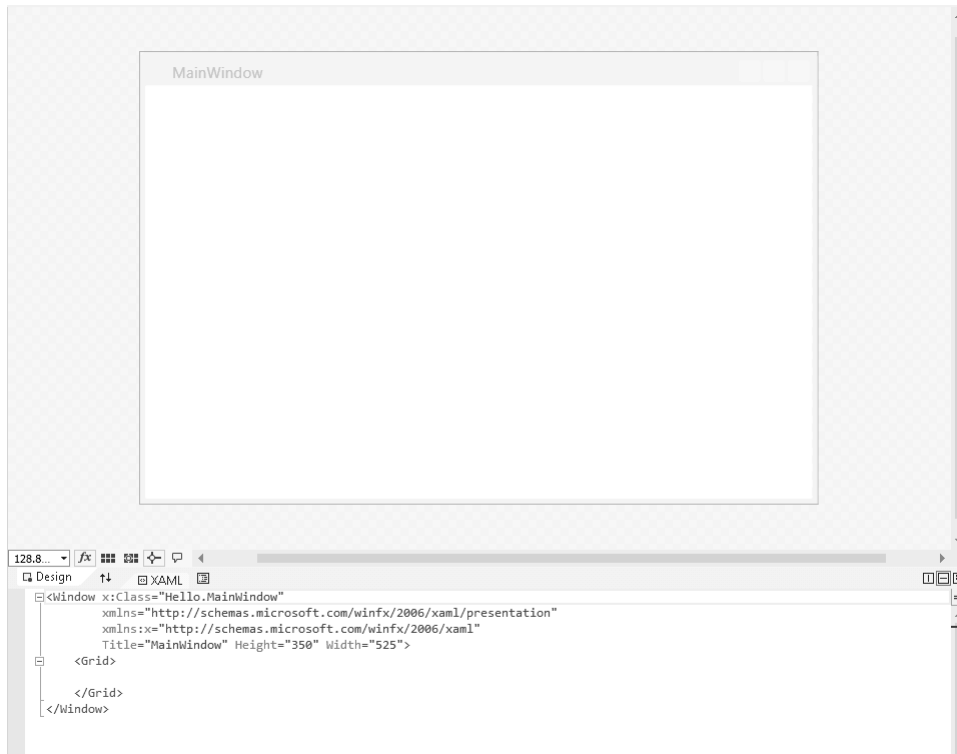
💡 **Tip** Close the Output and Error List windows to provide more space for displaying the Design View window.

Note Before going further, it is worth explaining some terminology. In a typical WPF application, the UI 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 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 2013 IDE, such as the Design View window.

In the following exercises, you will use the Design View window to add three controls to the form displayed by your application, and you will examine some of the C# code automatically generated by Visual Studio 2013 to implement these controls.

Note The steps in the following exercises are common to Windows 7, Windows 8, and Windows 8.1, except where any differences are explicitly called out.

Create the UI

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 8.1, expand the Common XAML Controls section.

   If you are using Windows 7 or Windows 8, expand the Common WPF Controls section.

   This section displays a list of controls that most graphical applications use.

Tip The All XAML Controls section (Windows 8.1) or All WPF Controls section (Windows 7 and Windows 8) 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 Ensure that 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 hide any part of the form, at the right end of the Toolbox title bar, click the Auto Hide button (it looks like a pin). The Toolbox appears permanently on the left side of the Visual Studio 2013 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 might 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 8.1, the XAML code for the *TextBlock* will look similar to this (your values for the *Margin* property might 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 or Windows 8, 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 8.1 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; for example, you can define tabular layouts by adding rows and columns to the *Grid*.

6. In the Design View window, click the *TextBlock* control. 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 8.1 (Global User Interface) and Windows 7 or Windows 8 (Segoe UI):

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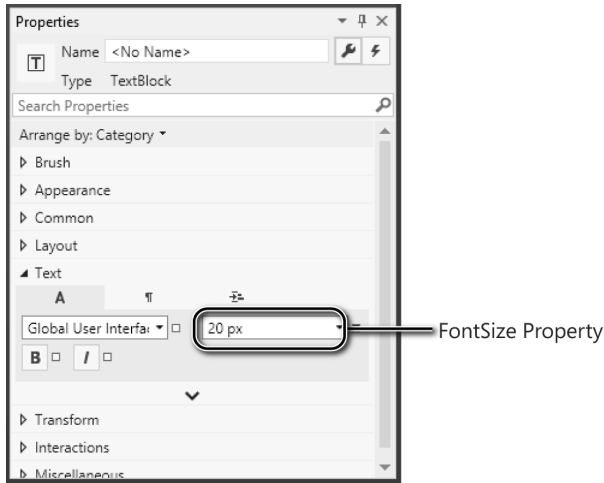
> **Note** The suffix *px* indicates that the font size is measured in pixels.

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.

Type over the value of the *FontSize* property in the XAML pane, changing 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 below 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 to ensure 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, indicating that you can resize the control. 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, at the top of the Properties window, change the value of the *Name* property from <No Name> to **userName**, as illustrated here:
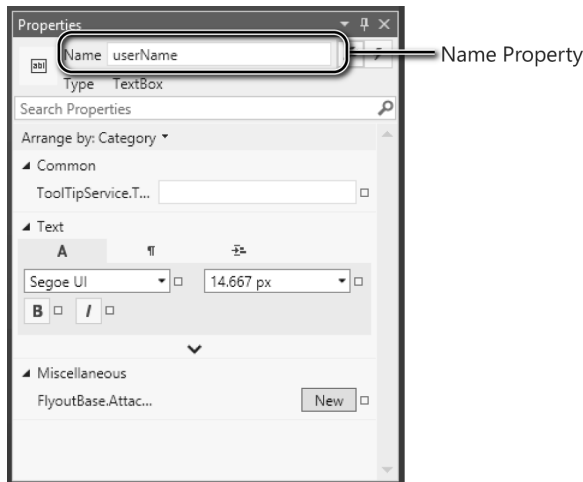
> **Note** You will learn more about naming conventions for controls and variables in Chapter 2, "Working with variables, operators, and expressions."

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 or Windows 8, 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**.

18. If you are using Windows 7 or Windows 8, 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, 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:
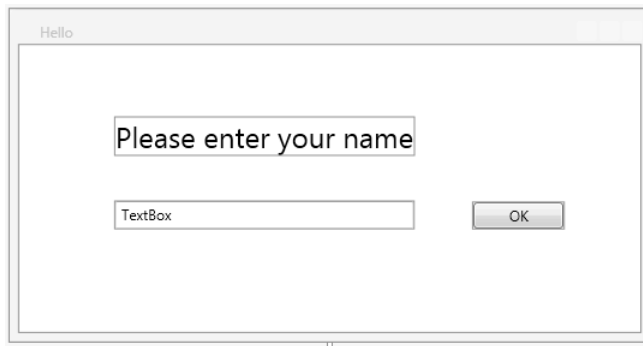
Note Pages in Windows Store apps cannot be resized in the same way as WPF forms; when they run, they automatically occupy the full screen of the device. However, they can adapt themselves to different screen resolutions and device orientation, and present different views when they are "snapped." You 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 then 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 8.1, the form occupies the entire screen and looks like this:

Please enter your name

TextBox   OK
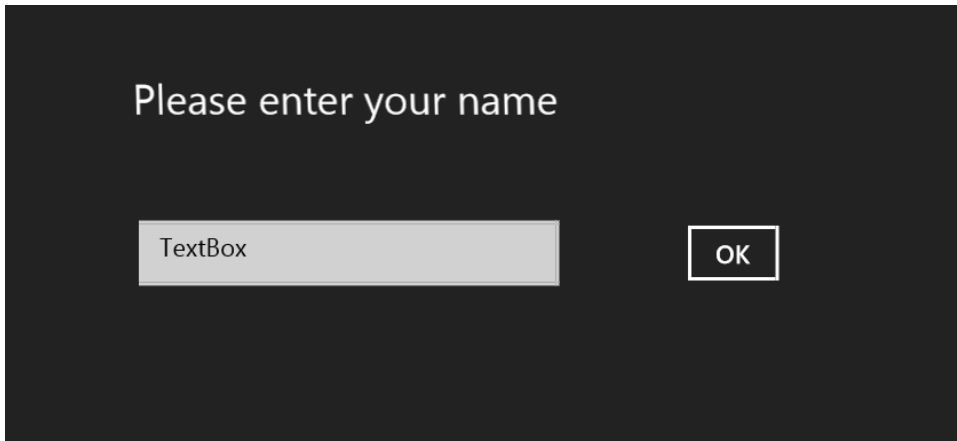
Note When you run a Windows Store App in Debug mode in Windows 8.1, two pairs of numbers appear in the upper-left and upper-right corner of the screen. These numbers track the frame rate, and developers can use them to determine when an application starts to become less responsive than it should be (possibly an indication of performance issues). They appear only when an application runs in Debug mode. A full description of what these numbers mean is beyond the scope of this book, so you can ignore them for now.

If you are using Windows 7 or Windows 8, the form looks like this:
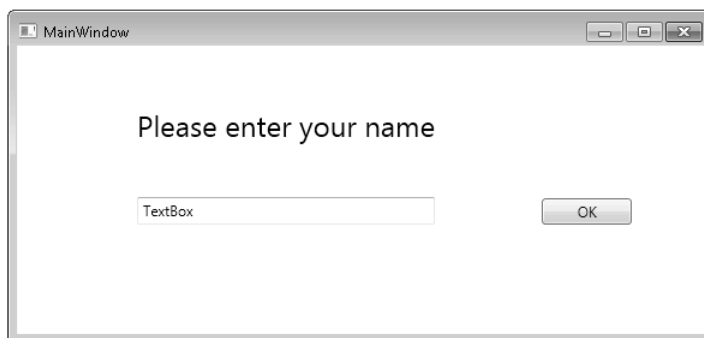
In the text box, you can type over what is there, type your name, and then click OK, but nothing happens yet. You need to add some code to indicate what should happen when the user clicks the OK button, which is what you will do next.

21. Return to Visual Studio 2013. On the DEBUG menu, click Stop Debugging.

   • If you are using Windows 8.1, press the Windows key + B. This should return you to the Windows Desktop running Visual Studio, from which you can access the Debug menu.

- If you are using Windows 7 or Windows 8, you can switch directly to Visual Studio. You can also click the close button (the X in the upper-right corner of the form) to close the form, stop debugging, and return to Visual Studio.

Closing a Windows Store app

If you are using Windows 8.1 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 8.1 suspends an application when it is not currently displayed, and it will terminate the application when the operating system needs to free the resources consumed by that 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 and hold it until the image of the application flips around (if you release the application before the image flips, the application will continue to run in the background). 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 upper-right corner of the screen and then drag the image of Visual Studio to the middle of the screen (or press Windows key + B)

2. At the bottom of the desktop, right-click the Windows taskbar, and then click Task Manager.

3. In the Task Manager window, click the Hello application, and then click End Task.
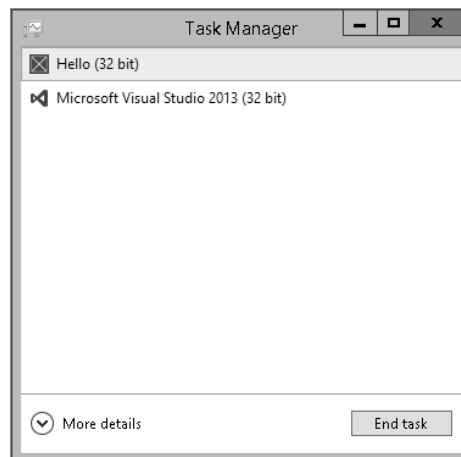
4. Close the Task Manager window.

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 2013 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 WPF application, and the following sections summarize these application styles separately.

Examining the Windows Store app

If you are using Windows 8.1, 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();
        }
    }
}
```

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 runs 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 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 at which 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
code
        /// executed, and as such is the logical equivalent of main() or WinMain().
        /// </summary>
        public App()
        {
            this.InitializeComponent();
            this.Suspending += OnSuspending;
```

```csharp
        /// <summary>
        /// Invoked when the application is launched normally by the end user.  Other entry
points
        /// 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 e)
        {

#if DEBUG
            if (System.Diagnostics.Debugger.IsAttached)
            {
                this.DebugSettings.EnableFrameRateCounter = true;
            }
#endif

            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
page
                rootFrame = new Frame();

                if (e.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), e.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
contents
        /// 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, high-lighted 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 it's helpful that you 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 or Windows 8, in Solution Explorer, click the arrow adjacent to the Main-Window.xaml file to expand the node. The file MainWindow.xaml.cs appears; double-click this file. The code for the form displays in the Code and Text Editor window, as shown here:

```
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; 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 might 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. In the Design View window, open the MainPage.xaml file (Windows 8.1) or MainWindow.xaml file (Windows 7 or Windows 8) (double-click MainPage.xaml or MainWindow.xaml in Solution Explorer).

2. Still 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 the Selected Element button.

   This button displays an icon that looks like a bolt of lightning, as demonstrated here:
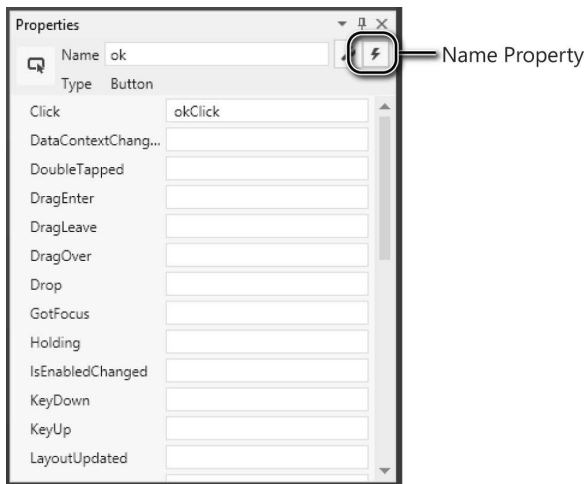
   The Properties window displays a list of event names for the *Button* control. An event indicates a significant action that usually requires a response, and you can write your own code to perform this response.

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

   The MainPage.xaml.cs file (Windows 8.1) or MainWindow.xaml.cs file (Windows 7 or Windows 8) 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 8.1, 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:

```
{
  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 ensure that 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 *<Your-Name>* 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.

This code will display a warning concerning the use of an asynchronous method when it is compiled. You do not have to be concerned about this warning. Asynchronous methods are explained more fully in Chapter 24

6. If you are using Windows 7 or Windows 8, just add the following 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 type your name over the existing text, and then click OK.

If you are using Windows 8.1, a message dialog appears across the middle of the screen, welcoming you by name.

Please enter your name

John   OK
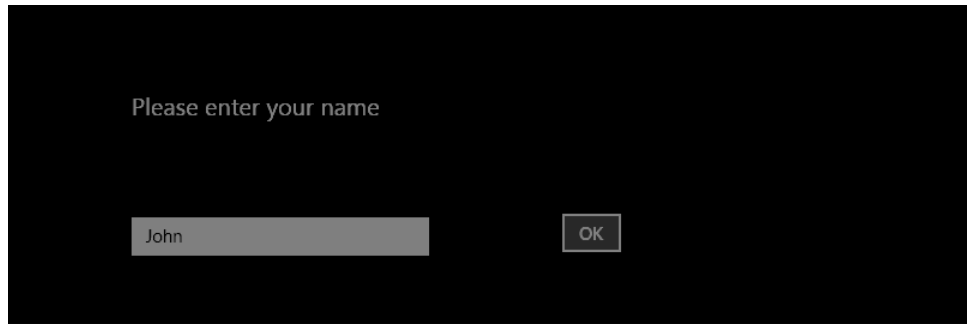
Hello John

Close

If you are using Windows 7 or Windows 8, a message box appears displaying the following greeting:

11. Click Close in the message dialog (Windows 8.1) or OK (Windows 7 or Windows 8) in the message box.

12. Return to Visual Studio 2013 and then, on the Debug menu, click Stop Debugging.

Summary

In this chapter, you saw how to use Visual Studio 2013 to create, build, and run applications. You created a console application that displays its output in a console window, and you created a WPF application with a simple GUI.

■ If you want to continue to the next chapter, keep Visual Studio 2013 running, and turn to Chapter 2.

■ If you want to exit Visual Studio 2013 now, on the File menu, click Exit. If you see a Save dialog box, click Yes to save the project.

Quick Reference

To	Do this
Create a new console application using Visual Studio 2013	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. in the Location box, specify a directory for the project files. Type a name for the project and then click OK.
Create a new Windows Store blank graphical application for Windows 8.1 using Visual Studio 2013	On the File menu, point to New, and then click Project to open the New Project dialog box. In the left pane, in the Installed Templates section, expand Visual C#, and then click Windows Store. In the middle pane, click Blank App (XAML). In the Location box, specify a directory for the project files. Type a name for the project and then click OK.
Create a new WPF graphical application for Windows 7 or Windows 8 using Visual Studio 2013	On the File menu, point to New, and then click Project to open the New Project dialog box. In the left pane, in the Installed Templates section, 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 and then 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.

Using arrays

After completing this chapter, you will be able to:

- Declare array variables.

- Populate an array with a set of data items.

- Access the data items held in an array.

- Iterate through the data items in an array.

You have already seen how to create and use variables of many different types. However, all the examples of variables you have seen so far have one thing in common—they hold information about a single item (an *int*, a *float*, a *Circle*, a *Date*, and so on). What happens if you need to manipulate a set of items? One solution is to create a variable for each item in the set, but this leads to a number of further questions: How many variables do you need? How should you name them? If you need to perform the same operation on each item in the set (such as increment each variable in a set of integers), how would you avoid very repetitive code? This solution assumes that you know, when you write the program, how many items you will need, but how often is this the case? For example, if you are writing an application that reads and processes records from a database, how many records are in the database, and how likely is this number to change?

Arrays provide a mechanism that helps to solve these problems.

Declaring and creating an array

An *array* is an unordered sequence of items. All the items in an array have the same type, unlike the fields in a structure or class, which can have different types. The items in an array live in a contiguous block of memory and are accessed by using an index, unlike fields in a structure or class, which are accessed by name.

Declaring array variables

You declare an array variable by specifying the name of the element type, followed by a pair of square brackets, followed by the variable name. The square brackets signify that the variable is an array. For example, to declare an array of *int* variables named *pins* (for holding a set of personal identification numbers) you can write the following:

```
int[] pins; // Personal Identification Numbers
```

You are not restricted to primitive types as array elements. You can also create arrays of structures, enumerations, and classes. For example, you can create an array of *Date* structures like this:

```
Date[] dates;
```

Tip It is often useful to give array variables plural names, such as *places* (where each element is a *Place*), *people* (where each element is a *Person*), or *times* (where each element is a *Time*).

Creating an array instance

Arrays are reference types, regardless of the type of their elements. This means that an array variable *refers* to a contiguous block of memory holding the array elements on the heap, just as a class variable refers to an object on the heap. (For a description of values and references and the differences between the stack and the heap, see Chapter 8, "Understanding values and references.") This rule applies regardless of the type of the data items in the array. Even if the array contains a value type such as *int*, the memory will still be allocated on the heap; this is the one case where value types are not allocated memory on the stack.

Remember that when you declare a class variable, memory is not allocated for the object until you create the instance by using *new*. Arrays follow the same pattern: when you declare an array variable, you do not declare its size and no memory is allocated (other than to hold the reference on the stack). The array is given memory only when the instance is created, and this is also the point at which you specify the size of the array.

To create an array instance, you use the *new* keyword followed by the element type, followed by the size of the array you're creating between square brackets. Creating an array also initializes its elements by using the now familiar default values (*0*, *null*, or *false*, depending on whether the type is numeric, a reference, or a Boolean, respectively). For example, to create and initialize a new array of four integers for the *pins* variable declared earlier, you write this:

```
pins = new int[4];
```

The following graphic illustrates what happens when you declare an array, and later when you create an instance of the array:
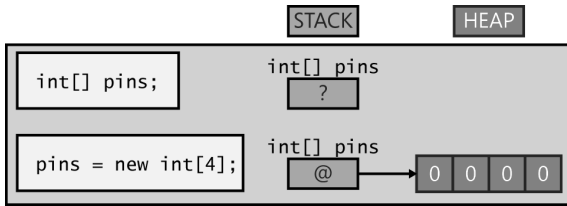
Because the memory for the array instance is allocated dynamically, the size of the array does not have to be a constant; it can be calculated at run time, as shown in this example:

```
int size = int.Parse(Console.ReadLine());
int[] pins = new int[size];
```

You're can also create an array whose size is 0. This might sound bizarre, but it's useful for situations in which the size of the array is determined dynamically and could even be 0. An array of size 0 is not a *null* array; it is an array containing zero elements.

Populating and using an array

When you create an array instance, all the elements of the array are initialized to a default value depending on their type. For example, all numeric values default to *0*, objects are initialized to *null*, *DateTime* values are set to the date and time *"01/01/0001 00:00:00"*, and strings are initialized to *null*. You can modify this behavior and initialize the elements of an array to specific values if you prefer. You achieve this by providing a comma-separated list of values between a pair of braces. For example, to initialize *pins* to an array of four *int* variables whose values are *9, 3, 7,* and *2*, you write this:

```
int[] pins = new int[4]{ 9, 3, 7, 2 };
```

The values between the braces do not have to be constants; they can be values calculated at run time, as shown in the following example, which populates the *pins* array with four random numbers:

```
Random r = new Random();
int[] pins = new int[4]{ r.Next() % 10, r.Next() % 10,
                         r.Next() % 10, r.Next() % 10 };
```

Note The *System.Random* class is a pseudorandom number generator. The *Next* method returns a nonnegative random integer in the range *0* to *Int32.MaxValue* by default. The *Next* method is overloaded, and other versions enable you to specify the minimum value and maximum value of the range. The default constructor for the *Random* class seeds the random number generator with a time-dependent seed value, which reduces the possibility of the class duplicating a sequence of random numbers. Using an overloaded version of the constructor, you can provide your own seed value. That way, you can generate a repeatable sequence of random numbers for testing purposes.

The number of values between the braces must exactly match the size of the array instance being created:

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7 };    // compile-time error
int[] pins = new int[4]{ 9, 3, 7, 2 }; // OK
```

When you're initializing an array variable in this way, you can actually omit the *new* expression and the size of the array. In this case, the compiler calculates the size from the number of initializers and generates code to create the array, such as in the following example:

```
int[] pins = { 9, 3, 7, 2 };
```

If you create an array of structures or objects, you can initialize each structure in the array by calling the structure or class constructor, as shown in this example:

```
Time[] schedule = { new Time(12,30), new Time(5,30) };
```

Creating an implicitly typed array

The element type when you declare an array must match the type of elements that you attempt to store in the array. For example, if you declare *pins* to be an array of *int*, as shown in the preceding examples, you cannot store a *double, string, struct*, or anything that is not an *int* in this array. If you specify a list of initializers when declaring an array, you can let the C# compiler infer the actual type of the elements in the array for you, like this:

```
var names = new[]{"John", "Diana", "James", "Francesca"};
```

In this example, the C# compiler determines that the *names* variable is an array of strings. It is worth pointing out a couple of syntactic quirks in this declaration. First, you omit the square brackets from the type; the *names* variable in this example is declared simply as *var*, not *var[]*. Second, you must specify the *new* operator and square brackets before the initializer list.

If you use this syntax, you must ensure that all the initializers have the same type. This next example causes the compile-time error "No best type found for implicitly typed array":

```
var bad = new[]{"John", "Diana", 99, 100};
```

However, in some cases, the compiler will convert elements to a different type, if doing so makes sense. In the following code, the *numbers* array is an array of *double* because the constants *3.5* and *99.999* are both *double*, and the C# compiler can convert the integer values *1* and *2* to *double* values:

```
var numbers = new[]{1, 2, 3.5, 99.999};
```

Generally, it is best to avoid mixing types and hoping that the compiler will convert them for you.

Implicitly typed arrays are most useful when you are working with anonymous types, as described in Chapter 7, "Creating and managing classes and objects." The following code creates an array of anonymous objects, each containing two fields specifying the name and age of the members of my family:

```
var names = new[] { new { Name = "John", Age = 47 },
                    new { Name = "Diana", Age = 46 },
                    new { Name = "James", Age = 20 },
                    new { Name = "Francesca", Age = 18 } };
```

The fields in the anonymous types must be the same for each element of the array.

Accessing an individual array element

To access an individual array element, you must provide an index indicating which element you require. Array indexes are zero-based; thus, the initial element of an array lives at index 0 and not index 1. An index value of *1* accesses the second element. For example, you can read the contents of element 2 of the *pins* array into an *int* variable by using the following code:

```
int myPin;
myPin = pins[2];
```

Similarly, you can change the contents of an array by assigning a value to an indexed element:

```
myPin = 1645;
pins[2] = myPin;
```

All array element access is bounds-checked. If you specify an index that is less than 0 or greater than or equal to the length of the array, the compiler throws an *IndexOutOfRangeException* exception, as in this example:

```
try
{
    int[] pins = { 9, 3, 7, 2 };
    Console.WriteLine(pins[4]); // error, the 4th and last element is at index 3
}
catch (IndexOutOfRangeException ex)
{
    ...
}
```

Iterating through an array

All arrays are actually instances of the *System.Array* class in the Microsoft .NET Framework, and this class defines a number of useful properties and methods. For example, you can query the *Length* property to discover how many elements an array contains and iterate through all the elements of an array by using a *for* statement. The following sample code writes the array element values of the *pins* array to the console:

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
    int pin = pins[index];
    Console.WriteLine(pin);
}
```

It is common for new programmers to forget that arrays start at element 0 and that the last element is numbered *Length* – 1. C# provides the *foreach* statement with which you can iterate through the elements of an array without worrying about these issues. For example, here's the preceding *for* statement rewritten as an equivalent *foreach* statement:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
    Console.WriteLine(pin);
}
```

The *foreach* statement declares an iteration variable (in this example, *int pin*) that automatically acquires the value of each element in the array. The type of this variable must match the type of the elements in the array. The *foreach* statement is the preferred way to iterate through an array; it expresses the intention of the code directly, and all of the *for* loop scaffolding drops away. However, in a few cases, you'll find that you have to revert to a *for* statement:

- A *foreach* statement always iterates through the entire array. If you want to iterate through only a known portion of an array (for example, the first half) or bypass certain elements (for example, every third element), it's easier to use a *for* statement.

- A *foreach* statement always iterates from index 0 through index *Length* – 1. If you want to iterate backward or in some other sequence, it's easier to use a *for* statement.

- If the body of the loop needs to know the index of the element rather than just the value of the element, you'll have to use a *for* statement.

- If you need to modify the elements of the array, you'll have to use a *for* statement. This is because the iteration variable of the *foreach* statement is a read-only copy of each element of the array.

> **Tip** It's perfectly safe to attempt to iterate through a zero-length array by using a *foreach* statement.

You can declare the iteration variable as a *var* and let the C# compiler work out the type of the variable from the type of the elements in the array. This is especially useful if you don't actually know the type of the elements in the array, such as when the array contains anonymous objects. The following example demonstrates how you can iterate through the array of family members shown earlier:

```
var names = new[] { new { Name = "John", Age = 47 },
                    new { Name = "Diana", Age = 46 },
                    new { Name = "James", Age = 20 },
```

```
                    new { Name = "Francesca", Age = 18 } };
foreach (var familyMember in names)
{
    Console.WriteLine("Name: {0}, Age: {1}", familyMember.Name, familyMember.Age);
}
```

Passing arrays as parameters and return values for a method

You can define methods that take arrays as parameters or pass them back as return values.

The syntax for passing an array as a parameter is much the same as declaring an array. For example, the code sample that follows defines a method called *ProcessData* that takes an array of integers as a parameter. The body of the method iterates through the array and performs some unspecified processing on each element:

```
public void ProcessData(int[] data)
{
    foreach (int i in data)
    {
        ...
    }
}
```

It is important to remember that arrays are reference objects, so if you modify the contents of an array passed as a parameter inside a method such as *ProcessData*, the modification is visible through all references to the array, including the original argument passed as the parameter.

To return an array from a method, you specify the type of the array as the return type. In the method, you create and populate the array. The following example prompts the user for the size of an array, followed by the data for each element. The array created by the method is passed back as the return value:

```
public int[] ReadData()
{
    Console.WriteLine("How many elements?");
    string reply = Console.ReadLine();
    int numElements = int.Parse(reply);

    int[] data = new int[numElements];
    for (int i = 0; i < numElements; i++)
    {
        Console.WriteLine("Enter data for element {0}", i);
        reply = Console.ReadLine();
        int elementData = int.Parse(reply);
        data[i] = elementData;
    }
    return data;
}
```

You can call the *ReadData* method like this:

```
int[] data = ReadData();
```

Copying arrays

Arrays are reference types (remember that an array is an instance of the *System.Array* class). An array variable contains a reference to an array instance. This means that when you copy an array variable, you actually end up with two references to the same array instance, as demonstrated in the following example:

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; //  alias and pins refer to the same array instance
```

In this example, if you modify the value at *pins[1]*, the change will also be visible by reading *alias[1]*.

If you want to make a copy of the array instance (the data on the heap) that an array variable refers to, you have to do two things. First, you create a new array instance of the same type and the same length as the array you are copying. Second, you copy the data element by element from the original array to the new array, as in this example:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
    copy[i] = pins[i];
}
```

Note that this code uses the *Length* property of the original array to specify the size of the new array.

Copying an array is actually a common requirement of many applications—so much so that the *System.Array* class provides some useful methods that you can employ to copy an array rather than writing your own code. For example, the *CopyTo* method copies the contents of one array into another array given a specified starting index. The following example copies all the elements from the *pins* array to the *copy* array starting at element zero:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

Another way to copy the values is to use the *System.Array* static method named *Copy*. As with *CopyTo*, you must initialize the target array before calling *Copy*:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

> **Note** Ensure that you specify a valid value for the length parameter of the *Array. Copy* method. If you provide a negative value, the method throws an *ArgumentOutOf RangeException* exception. If you specify a value that is greater than the number of elements in the source array, the method throws an *ArgumentException* exception.

Yet another alternative is to use the *System.Array* instance method named *Clone*. You can call this method to create an entire array and copy it in one action:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone();
```

> **Note** *Clone* methods are described in Chapter 8. The *Clone* method of the *Array* class returns an *object* rather than *Array*, which is why you must cast it to an array of the appropriate type when you use it. Furthermore, the *Clone*, *CopyTo*, and *Copy* methods all create a *shallow* copy of an array (shallow and deep copying are also described in Chapter 8). If the elements in the array being copied contain references, the *Clone* method simply copies the references rather than the objects being referred to. After copying, both arrays refer to the same set of objects. If you need to create a deep copy of such an array, you must use appropriate code in a *for* loop.

Using multidimensional arrays

The arrays shown so far have contained a single dimension, and you can think of them as simple lists of values. You can create arrays with more than one dimension. For example, to create a two-dimensional array, you specify an array that requires two integer indexes. The following code creates a two-dimensional array of 24 integers called *items*. If it helps, you can think of the array as a table with the first dimension specifying a number of rows, and the second specifying a number of columns.

```
int[,] items = new int[4, 6];
```

To access an element in the array, you provide two index values to specify the "cell" holding the element. (A cell is the intersection of a row and a column.) The following code shows some examples using the *items* array:

```
items[2, 3] = 99;           // set the element at cell(2,3) to 99
items[2, 4] = items [2,3];  // copy the element in cell(2, 3) to cell(2, 4)
items[2, 4]++;              // increment the integer value at cell(2, 4)
```

There is no limit on the number of dimensions that you can specify for an array. The next code example creates and uses an array called *cube* that contains three dimensions. Notice that you must specify three indexes to access each element in the array.

```
int[, ,] cube = new int[5, 5, 5];
cube[1, 2, 1] = 101;
cube[1, 2, 2] = cube[1, 2, 1] * 3;
```

At this point, it is worth giving a word of caution about creating arrays with more than three dimensions. Specifically, arrays can consume a lot of memory. The *cube* array contains 125 elements (5 * 5 * 5). A four-dimensional array for which each dimension has a size of 5 contains 625 elements. If you start to create arrays with three or more dimensions, you can soon run out of memory. Therefore, you should always be prepared to catch and handle *OutOfMemoryException* exceptions when you use multidimensional arrays.

Creating jagged arrays

In C#, ordinary multidimensional arrays are sometimes referred to as *rectangular* arrays. Each dimension has a regular shape. For example, in the following tabular two-dimensional *items* array, every row has a column containing 40 elements, and there are 160 elements in total:

```
int[,] items = new int[4, 40];
```

As mentioned in the previous section, multidimensional arrays can consume a lot of memory. If the application uses only some of the data in each column, allocating memory for unused elements is a waste. In this scenario, you can use a *jagged* array, for which each column has a different length, like this:

```
int[][] items = new int[4][];
int[] columnForRow0 = new int[3];
int[] columnForRow1 = new int[10];
int[] columnForRow2 = new int[40];
int[] columnForRow3 = new int[25];
items[0] = columnForRow0;
items[1] = columnForRow1;
items[2] = columnForRow2;
items[3] = columnForRow3;
...
```

In this example, the application requires only 3 elements in the first column, 10 elements in the second column, 40 elements in the third column, and 25 elements in the final column. This code illustrates an array of arrays—rather than *items* being a two-dimensional array, it has only a single dimension, but the elements in that dimension are themselves arrays. Furthermore, the total size of the *items* array is 78 elements rather than 160; no space is allocated for elements that the application is not going to use.

It is worth highlighting some of the syntax in this example. The following declaration specifies that *items* is an array of arrays of *int*.

```
int[][] items;
```

The following statement initializes *items* to hold four elements, each of which is an array of indeterminate length:

```
items = new int[4][];
```

The arrays *columnForRow0* to *columnForRow3* are all single-dimensional *int* arrays, initialized to hold the required amount of data for each column. Finally, each column array is assigned to the appropriate elements in the *items* array, like this:

```
items[0] = columnForRow0;
```

Recall that arrays are reference objects, so this statement simply adds a reference to *columnForRow0* to the first element in the *items* array; it does not actually copy any data. You can populate data

in this column either by assigning a value to an indexed element in *columnForRow0* or by referencing it through the *items* array. The following statements are equivalent:

```
columnForRow0[1] = 99;
items[0][1] = 99;
```

You can extend this idea further if you want to create arrays of arrays of arrays rather than rectangular three-dimensional arrays, and so on.

> **Note** If you have written code using the Java programming language in the past, you should be familiar with this concept. Java does not have multidimensional arrays; instead, you can create arrays of arrays exactly as just described.

In the following exercise, you will use arrays to implement an application that deals playing cards as part of a card game. The application displays a form with four hands of cards dealt at random from a regular (52-card) pack of playing cards. You will complete the code that deals the cards for each hand.

Use arrays to implement a card game

1. Start Microsoft Visual Studio 2013 if it is not already running.

2. Open the Cards project, which is located in the \Microsoft Press\Visual CSharp Step By Step\ Chapter 10\Windows *X*\Cards folder in your Documents folder.

3. On the Debug menu, click Start Debugging to build and run the application.

   A form appears with the caption Card Game, four text boxes (labeled North, South, East, and West), and a button with the caption Deal.

   If you are using Windows 7, the form looks like this:
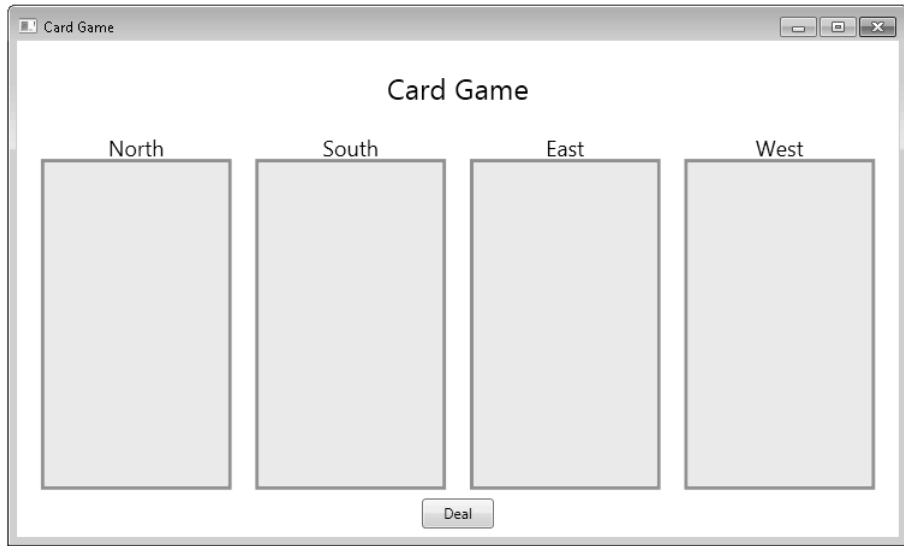
If you are using Windows 8.1, the Deal button is on the app bar rather than on the main form, and the application looks like this:

4. Click Deal.

   Nothing happens. You have not yet implemented the code that deals the cards; this is what you will do in this exercise.

5. Return to Visual Studio 2013. On the Debug menu, click Stop Debugging.

6. In Solution Explorer, locate the Value.cs file. Open this file in the Code and Text Editor window.

   This file contains an enumeration called *Value*, which represents the different values that a card can have, in ascending order:

   ```
   enum Value { Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King,
   Ace }
   ```

7. Open the Suit.cs file in the Code and Text Editor window.

   This file contains an enumeration called *Suit*, which represents the suits of cards in a regular pack:

   ```
   enum Suit { Clubs, Diamonds, Hearts, Spades }
   ```

8. Display the PlayingCard.cs file in the Code and Text Editor window.

   This file contains the *PlayingCard* class. This class models a single playing card.

   ```
   class PlayingCard
   {
       private readonly Suit suit;
       private readonly Value value;

       public PlayingCard(Suit s, Value v)
       {
           this.suit = s;
           this.value = v;
       }

       public override string ToString()
       {
           string result = string.Format("{0} of {1}", this.value, this.suit);
           return result;
       }

       public Suit CardSuit()
       {
           return this.suit;
       }
   ```

```
        public Value CardValue()
        {
            return this.value;
        }
}
```

This class has two *readonly* fields that represent the value and suit of the card. The constructor initializes these fields.

> **Note** A *readonly* field is useful for modeling data that should not change after it has been initialized. You can assign a value to a *readonly* field by using an initializer when you declare it, or in a constructor, but thereafter you cannot change it.

The class contains a pair of methods called *CardValue* and *CardSuit* that return this information, and it overrides the *ToString* method to return a string representation of the card.

> **Note** The *CardValue* and *CardSuit* methods are actually better implemented as properties, which you learn how to do in Chapter 15.

9. Open the Pack.cs file in the Code and Text Editor window.

This file contains the *Pack* class, which models a pack of playing cards. At the top of the *Pack* class are two public *const int* fields called *NumSuits* and *CardsPerSuit*. These two fields specify the number of suits in a pack of cards and the number of cards in each suit. The private *cardPack* variable is a two-dimensional array of *PlayingCard* objects. You will use the first dimension to specify the suit and the second dimension to specify the value of the card in the suit. The *randomCardSelector* variable is a random number generated based on the *Random* class. You will use the *randomCardSelector* variable to help shuffle the cards before they are dealt to each hand.

```
class Pack
{
    public const int NumSuits = 4;
    public const int CardsPerSuit = 13;
    private PlayingCard[,] cardPack;
    private Random randomCardSelector = new Random();
    ...
}
```

10. Locate the default constructor for the *Pack* class. Currently, this constructor is empty apart from a *// TODO:* comment. Delete the comment, and add the following statement shown in bold to instantiate the *cardPack* array with the appropriate values for each dimension:

```
public Pack()
{
    this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
}
```

11. Add the following code shown in bold to the *Pack* constructor. These statements populate the *cardPack* array with a full, sorted deck of cards.

```
public Pack()
{
    this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
    for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
    {
        for (Value value = Value.Two; value <= Value.Ace; value++)
        {
            this.cardPack[(int)suit, (int)value] = new PlayingCard(suit, value);
        }
    }
}
```

The outer *for* loop iterates through the list of values in the *Suit* enumeration, and the inner loop iterates through the values each card can have in each suit. The inner loop creates a new *PlayingCard* object of the specified suit and value, and adds it to the appropriate element in the *cardPack* array.

> **Note** You must use one of the integer types as indexes into an array. The *suit* and *value* variables are enumeration variables. However, enumerations are based on the integer types, so it is safe to cast them to *int* as shown in the code.

12. Find the *DealCardFromPack* method in the *Pack* class. The purpose of this method is to pick a random card from the pack, remove the card from the pack to prevent it from being selected again, and then pass it back as the return value from the method.

The first task in this method is to pick a suit at random. Delete the comment and the statement that throws the *NotImplementedException* exception from this method and replace them with the following statement shown in bold:

```
public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
}
```

This statement uses the *Next* method of the *randomCardSelector* random number generator object to return a random number corresponding to a suit. The parameter to the *Next* method specifies the exclusive upper bound of the range to use; the value selected is between 0 and this value minus 1. Note that the value returned is an *int*, so it has to be cast before you can assign it a *Suit* variable.

There is always the possibility that there are no more cards left of the selected suit. You need to handle this situation and pick another suit if necessary.

13. After the code that selects a suit at random, add the *while* loop that follows (shown in bold).

This loop calls the *IsSuitEmpty* method to determine whether there are any cards of the specified suit left in the pack (you will implement the logic for this method shortly). If not, it picks another suit at random (it might actually pick the same suit again) and checks again. The loop repeats the process until it finds a suit with at least one card left.

```
public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
    while (this.IsSuitEmpty(suit))
    {
        suit = (Suit)randomCardSelector.Next(NumSuits);
    }
}
```

14. You have now selected a suit at random with at least one card left. The next task is to pick a card at random in this suit. You can use the random number generator to select a card value, but as before, there is no guarantee that the card with the chosen value has not already been dealt. However, you can use the same idiom as before: call the *IsCardAlreadyDealt* method (which you will examine and complete later) to determine whether the card has been dealt before, and if so, pick another card at random and try again, repeating the process until a card is found. To do this, add the following statements shown in bold to the *DealCardFromPack* method, after the existing code:

```
public PlayingCard DealCardFromPack()
{
    ...
    Value value = (Value)randomCardSelector.Next(CardsPerSuit);
    while (this.IsCardAlreadyDealt(suit, value))
    {
        value = (Value)randomCardSelector.Next(CardsPerSuit);
    }
}
```

15. You have now selected a random playing card that has not been dealt previously. Add the following code to the end of the *DealCardFromPack* method to return this card and set the corresponding element in the *cardPack* array to *null*:

```
public PlayingCard DealCardFromPack()
{
    ...
    PlayingCard card = this.cardPack[(int)suit, (int)value];
    this.cardPack[(int)suit, (int)value] = null;
    return card;
}
```

16. Locate the *IsSuitEmpty* method. Remember that the purpose of this method is to take a *Suit* parameter and return a Boolean value indicating whether there are any more cards of this suit left in the pack. Delete the comment and the statement that throws the *NotImplementedException* exception from this method, and add the following code shown in bold:

```
private bool IsSuitEmpty(Suit suit)
```

```
{
    bool result = true;
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        if (!IsCardAlreadyDealt(suit, value))
        {
            result = false;
            break;
        }
    }

    return result;
}
```

This code iterates through the possible card values and determines whether there is a card left in the *cardPack* array that has the specified suit and value by using the *IsCardAlreadyDealt* method, which you will complete in the next step. If the loop finds a card, the value in the *result* variable is set to *false* and the *break* statement causes the loop to terminate. If the loop completes without finding a card, the *result* variable remains set to its initial value of *true*. The value of the *result* variable is passed back as the return value of the method.

17. Find the *IsCardAlreadyDealt* method. The purpose of this method is to determine whether the card with the specified suit and value has already been dealt and removed from the pack. You will see later that when the *DealFromPack* method deals a card, it removes the card from the *cardPack* array and sets the corresponding element to *null*. Replace the comment and the statement that throws the *NotImplementedException* exception in this method with the code shown in bold:

```
private bool IsCardAlreadyDealt(Suit suit, Value value)
{
    return (this.cardPack[(int)suit, (int)value] == null);
}
```

This statement returns *true* if the element in the *cardPack* array corresponding to the suit and value is *null*, and it returns *false* otherwise.

18. The next step is to add the selected playing card to a hand. Open the Hand.cs file, and display it in the Code and Text Editor window. This file contains the *Hand* class, which implements a hand of cards (that is, all cards dealt to one player).

This file contains a *public const int* field called *HandSize*, which is set to the size of a hand of cards (13). It also contains an array of *PlayingCard* objects, which is initialized by using the *HandSize* constant. The *playingCardCount* field will be used by your code to keep track of how many cards the hand currently contains as it is being populated.

```
class Hand
{
    public const int HandSize = 13;
    private PlayingCard[] cards = new PlayingCard[HandSize];
    private int playingCardCount = 0;
    ...
}
```

The *ToString* method generates a string representation of the cards in the hand. It uses a *foreach* loop to iterate through the items in the *cards* array and calls the *ToString* method on each *PlayingCard* object it finds. These strings are concatenated together with a newline character in between (the \n character) for formatting purposes.

```
public override string ToString()
{
    string result = "";
    foreach (PlayingCard card in this.cards)
    {
        result += card.ToString() + "\n";
    }

    return result;
}
```

19. Locate the *AddCardToHand* method in the *Hand* class. The purpose of this method is to add the playing card specified as the parameter to the hand. Add the following statements shown in bold to this method:

```
public void AddCardToHand(PlayingCard cardDealt)
{
    if (this.playingCardCount >= HandSize)
    {
        throw new ArgumentException("Too many cards");
    }
    this.cards[this.playingCardCount] = cardDealt;
    this.playingCardCount++;
}
```

This code first checks to ensure that the hand is not already full. If the hand is full, it throws an *ArgumentException* exception (this should never occur, but it is good practice to be safe). Otherwise, the card is added to the *cards* array at the index specified by the *playingCardCount* variable, and this variable is then incremented.

20. In Solution Explorer, expand the MainWindow.xaml node and then open the MainWindow.xaml.cs file in the Code and Text Editor window.

This is the code for the Card Game window. Locate the *dealClick* method. This method runs when the user clicks the Deal button. Currently, it contains an empty *try* block and an exception handler that displays a message if an exception occurs.

21. Add the following statement shown in bold to the *try* block:

```
private void dealClick(object sender, RoutedEventArgs e)
{
    try
    {
        pack = new Pack();
    }
    catch (Exception ex)
    {
        ...
```

```
    }
}
```

This statement simply creates a new pack of cards. You saw earlier that this class contains a two-dimensional array holding the cards in the pack, and the constructor populates this array with the details of each card. You now need to create four hands of cards from this pack.

22. Add the following statements shown in bold to the *try* block:

```
try
{
    pack = new Pack();

    for (int handNum = 0; handNum < NumHands; handNum++)
    {
        hands[handNum] = new Hand();
    }
}
catch (Exception ex)
{
    ...
}
```

This *for* loop creates four hands from the pack of cards and stores them in an array called *hands*. Each hand is initially empty, so you need to deal the cards from the pack to each hand.

23. Add the following code shown in bold to the *for* loop:

```
try
{
    ...
    for (int handNum = 0; handNum < NumHands; handNum++)
    {
        hands[handNum] = new Hand();
        for (int numCards = 0; numCards < Hand.HandSize; numCards++)
        {
            PlayingCard cardDealt = pack.DealCardFromPack();
            hands[handNum].AddCardToHand(cardDealt);
        }
    }
}
catch (Exception ex)
{
    ...
}
```

The inner *for* loop populates each hand by using the *DealCardFromPack* method to retrieve a card at random from the pack and the *AddCardToHand* method to add this card to a hand.

24. Add the following code shown in bold after the outer *for* loop:

```
try
{
    ...
    for (int handNum = 0; handNum < NumHands; handNum++)
```

```
    {
        ...
    }

    north.Text = hands[0].ToString();
    south.Text = hands[1].ToString();
    east.Text = hands[2].ToString();
    west.Text = hands[3].ToString();
}
catch (Exception ex)
{
    ...
}
```

When all the cards have been dealt, this code displays each hand in the text boxes on the form. These text boxes are called *north, south, east*, and *west*. The code uses the *ToString* method of each hand to format the output.

If an exception occurs at any point, the *catch* handler displays a message box with the error message for the exception.

25. On the Debug menu, click Start Debugging. When the Card Game window appears, click Deal.

The cards in the pack should be dealt at random to each hand, and the cards in each hand should be displayed on the form, as shown in the following image:

Card Game

	North	South	East	West
	Seven of Diamonds	Two of Hearts	Seven of Spades	Ten of Hearts
	Queen of Diamonds	Five of Clubs	Four of Clubs	Two of Diamonds
	Ten of Spades	Ace of Spades	Six of Hearts	Three of Spades
	King of Diamonds	Three of Clubs	Nine of Spades	Ten of Clubs
	King of Spades	Four of Spades	Eight of Spades	Ace of Diamonds
	Five of Diamonds	Jack of Hearts	King of Clubs	Three of Diamonds
	Queen of Spades	Nine of Diamonds	Two of Spades	Four of Hearts
	Five of Spades	Two of Clubs	Eight of Hearts	Queen of Hearts
	Six of Diamonds	Ten of Diamonds	Queen of Clubs	Seven of Hearts
	Ace of Clubs	Seven of Clubs	Five of Hearts	Ace of Hearts
	Jack of Clubs	Nine of Clubs	Jack of Diamonds	Eight of Diamonds
	Six of Clubs	Four of Diamonds	Three of Hearts	King of Hearts
	Six of Spades	Jack of Spades	Eight of Clubs	Nine of Hearts

Deal

26. Click Deal again. Verify that a new set of hands is dealt and the cards in each hand change.

27. Return to Visual Studio and stop debugging.

Summary

In this chapter, you learned how to create and use arrays to manipulate sets of data. You saw how to declare and initialize arrays, access data held in arrays, pass arrays as parameters to methods, and return arrays from methods. You also learned how to create multidimensional arrays and how to use arrays of arrays.

- If you want to continue to the next chapter, keep Visual Studio 2013 running, and turn to Chapter 11.

- If you want to exit Visual Studio 2013 now, on the File menu, click Exit. If you see a Save dialog box, click Yes and save the project.

Quick reference

To	Do this
Declare an array variable	Write the name of the element type, followed by square brackets, followed by the name of the variable, followed by a semicolon. For example: `bool[] flags;`
Create an instance of an array	Write the keyword new, followed by the name of the element type, followed by the size of the array enclosed in square brackets. For example: `bool[] flags = new bool[10];`
Initialize the elements of an array to specific values	For an array, write the specific values in a comma-separated list enclosed in braces. For example: `bool[] flags = { true, false, true, false };`

Find the number of elements in an array	Use the Length property. For example:
	
	`int [] flags = ...;`
	`...`
	`int noOfElements = flags.Length;`

Access a single array element	Write the name of the array variable, followed by the integer index of the element enclosed in square brackets. Remember, array indexing starts at 0, not 1. For example: `bool initialElement = flags[0];`

Iterate through the elements of an array	Use a for statement or a foreach statement. For example:
	
	`bool[] flags = { true, false, true, false };`
	`for (int i = 0; i < flags.Length; i++)`
	`{`
	`    Console.WriteLine(flags[i]);`
	`}`
	
	`foreach (bool flag in flags)`
	`{`
	`    Console.WriteLine(flag);`
	`}`

To	Do this

Declare a multidimensional array variable	Write the name of the element type, followed by a set of square brackets with a comma separator indicating the number of dimensions, followed by the name of the variable, followed by a semicolon. For example, use the following to create a two-dimensional array called table and initialize it to hold 4 rows of 6 columns:
	
	`int[,] table;`
	`table = new int[4,6];`

Declare a jagged array variable	Declare the variable as an array of child arrays. You can initialize each child array to have a different length. For example, use the following to create a jagged array called items and initialize each child array:
	
	`int[][] items;`
	`items = new int[4][];`
	`items[0] = new int[3];`
	`items[1] = new int[10];`
	`items[2] = new int[40];`
	`items[3] = new int[25];`

Index

Symbols

& (ampersand)

    address-of operator, 202

    bitwise AND operator, 365

    && (logical AND) operator, 95, 97, 111

< > (angle brackets)

    <= and >= operators, pairing, 519

    > (greater than) operator, 95, 97, 111

    >= (greater than or equal to) operator, 95, 97, 111

    << (left-shift) operator, 364

    < (less than) operator, 95, 97, 111

    <= (less than or equal to) operator, 95, 97, 111

    < operator and > operator, pairing, 519

    >> right-shift operator, 364

* (asterisk)

    in pointers, 203

    *= (multiplication and assignment) operator, 114

    multiplication operator, 52, 58, 96

        associativity, 60

        overloading, 522

\ (backslash), escape character in C#, 110

{ } (braces or curly brackets), 59

    code block enclosed in, life span of variables defined in, 196

    enclosing code blocks, 98

    in lambda expressions, 425

    use in intializing array elements to specific values, 229

^ (caret), bitwise XOR operator, 365, 368

, (comma), separating multiple initializations and updates in for loops, 122

. (dot notation), 281, 318

= (equal sign)

    assignment operator, 60, 97, 113

        using properties to simulate, 512

    confusing = and == in if statement, 98

== (equal to) operator, 94, 97, 111

    implementing overloaded version, 523

    pairing with !+ operator, 518

    => operator, 425

    using with on clause of LINQ expression, 497

! (exclamation mark)

    != (inequality) operator, 94, 97, 111

        implementing overloaded version, 523

        pairing with == operator, 519

    logical NOT operator, 96, 191

- (minus sign)

    -- (decrement) operator, 61, 96, 114

        declaring your own version, 517

    negation operator, 96

    -= (subtraction and assignment) operator, 114, 133

        removing methods from delegates, 457

        unsubscribing from an event, 473

    subtraction operator, 52, 54, 57, 97

        overloading, 521

\n (newline character), 119

( ) (parentheses)

    enclosing parameters in lambda expressions, 425

    in method calls, 70

    using to override operator precedence, 59, 96

% (percent sign)

    %= (modulus and assignment) operator, 114

    modulus operator, 53, 55, 58, 96, 126

| (pipe symbol)

    bitwise OR operator, 364

    || (logical OR) operator, 95, 97, 111

+ (plus sign)

    += (addition and assignment) operator, 114, 127, 133

        evaluation, understanding, 516

        using with delegates, 456, 457

        using with events, 472

    addition operator, 52, 57, 97, 113, 513–516

N

About the author

JOHN SHARP is a principal technologist working for Content Master, a division of CM Group Ltd in the United Kingdom. The company specialises in advanced learning solutions for large international organisations, often utilising the latest and most innovative technologies to deliver effective learning outcomes. John gained an honors degree in Computing from Imperial College, London. He has been developing software and writing training courses, guides, and books for over 27 years. John has extensive experience in a range of technologies, from database systems and UNIX to C, C++, and C# applications for the .NET Framework.

He has also written about Java and JavaScript development, and designing enterprise solutions by using Windows Azure. Apart from seven editions of *Microsoft Visual C# Step By Step*, he has penned several other books, including *Microsoft Windows Communication Foundation Step By Step* and the *J# Core Reference*. In his role at Content Master he is a regular author for *Microsoft Patterns & Practices*, and has recently worked on guides such as *Building Hybrid Applications in the Cloud on Windows Azure*, and *Data Access for Highly Scalable Solutions Using SQL, NoSQL, and Polyglot Persistence.*