Microsoft

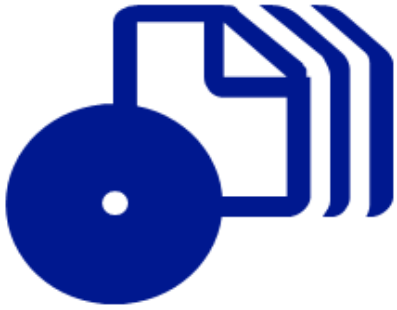
Microsoft® Visual C#® 2008

John Sharp

content master

Step by Step

How to access your CD files



The print edition of this book includes a CD. To access the CD files, go to http://aka.ms/624306/files, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft Press

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

Library of Congress Control Number: 2007939305

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 2 1 0 9 8 7

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Developmental and Project Editor: Lynn Finnel
Editorial Production: Waypoint Press
Technical Reviewer: Kurt Meyer; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Body Part No. X14-22686

Contents at a Glance

Table of Contents

Acknowledgments

An old Latin proverb says "Tempora mutantur, nos et mutantur in illis," which roughly translates into English as "Times change, and we change with them." This proverb has a quaint, sedate feel and was obviously penned before the Romans had heard of Microsoft, Windows, the .NET Framework, and C#; otherwise, they would have written something more like "Times change, and we run like mad trying to keep up!" When I look back over the last seven or eight years, I am absolutely flabbergasted to see how much the .NET Framework, and the C# language in particular, has evolved. I am also very thankful, because it keeps me in gainful employment, performing biannual updates on this book. I am not complaining because the .NET Framework is a superb platform for building applications and services, and I thank the visionaries in the various product groups at Microsoft who have dedicated several millennia of person-years of effort in its development. In my opinion, C# is the greatest vehicle for taking full advantage of the .NET Framework. I have thoroughly enjoyed watching its development and learning the new features that each new release provides. This book is my attempt to convey my enthusiasm for the language to other programmers who are just starting along the C# path of discovery.

As with all projects of this type, writing a book is a group effort. The team I have had the pleasure of working with at Microsoft Press is second to none. In particular, I would like to single out Lynn Finnel who has kept the faith in me over several editions of this book, Christina Palaia and Jennifer Harris for their thorough editing of my manuscripts, and Stephen Sagman who has worked like a Trojan keeping us all in order and on schedule. I must pay special thanks to Kurt Meyer for his sterling efforts in reviewing my work, correcting my mistakes, and suggesting modifications, and of course to Jon Jagger who coauthored the first edition of this book with me back in 2001.

My long-suffering family have been wonderful, as they always are. Diana is now familiar with terms such as "DLINQ" and "lambda expression" and throws them into conversation with effortless aplomb. (For example, "Will you ever stop talking about DLINQ and lambda expressions?") James is still convinced that I spend my life playing computer games rather than working. Francesca has developed a frowning nod that says, "I have no idea what you are talking about, but I will nod anyway in the hope that you might stop." And Ginger, my arch-competitor for the chair in my study, has tried her best to completely distract me and delay my efforts in the ways that only a cat can.

As ever, "Up the Gills!"

—John Sharp

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. With the advent of C# 2.0 in 2005, several important new features were added to the language, including generics, iterators, and anonymous methods. C# 3.0, available as part of Microsoft Visual Studio 2008, adds further features, such as extension methods, lambda expressions, and, most famously of all, the Language Integrated Query facility, or LINQ. The development environment provided by Visual Studio 2008 makes these powerful features easy to use, and the many new wizards and enhancements included in Visual Studio 2008 can greatly improve your productivity as a developer.

Who This Book Is For

The aim of this book is to teach you the fundamentals of programming with C# by using Visual Studio 2008 and the .NET Framework version 3.5. You will learn the features of the C# language, and then use them to build applications running on the Microsoft Windows operating system. By the time you complete this book, you will have a thorough understanding of C# and will have used it to build Windows Presentation Foundation (WPF) applications, access Microsoft SQL Server databases, develop ASP.NET Web applications, and build and consume a Windows Communication Foundation service.

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++, Sun Microsystems 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 next section, "Installing and Using the Practice Files." **2.** Work through the chapters in Parts I, II, and III sequentially. **3.** Complete Parts IV, V, and VI as your level of experience and interest dictates.
Familiar with procedural programming languages such as C, but new to C#	**1.** Install the practice files as described in the next section, "Installing and Using the Practice Files." Skim the first five chapters to get an overview of C# and Visual Studio 2008, and then concentrate on Chapters 6 through 21. **2.** Complete Parts IV, V, and VI 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 next section, "Installing and Using the Practice Files." **2.** Skim the first seven chapters to get an overview of C# and Visual Studio 2008, and then concentrate on Chapters 8 through 21. **3.** For information about building Windows-based applications and using a database, read Parts IV and V. **4.** For information about building Web applications and Web services, read Part VI.
Switching from Visual Basic 6	**1.** Install the practice files as described in the next section, "Installing and Using the Practice Files." **2.** Work through the chapters in Parts I, II, and III sequentially. **3.** For information about building Windows-based applications, read Part IV. **4.** For information about accessing a database, read Part V. **5.** For information about creating Web applications and Web services, read Part VI. **6.** Read the Quick Reference sections at the end of the chapters for information about specific C# and Visual Studio 2008 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.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow. Before you start, read the following list, which explains conventions you'll see throughout the book and points out helpful features that you might want to use.

Conventions

- Each exercise is a series of tasks. Each task is presented as a series of numbered steps (1, 2, and so on). A round bullet (•) indicates an exercise that has only one step.
- Notes labeled "tip" provide additional information or alternative methods for completing a step successfully.
- Notes labeled "important" alert you to information you need to check before continuing.
- Text that you type 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.

Other Features

- Sidebars throughout the book provide more in-depth information about the exercise. The sidebars might contain background information, design tips, or features related to the information being discussed.
- Each chapter ends with a Quick Reference section. The Quick Reference section contains quick reminders of how to perform the tasks you learned in the chapter.

System Requirements

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

- Windows Vista Home Premium Edition, Windows Vista Business Edition, or Windows Vista Ultimate Edition. The exercises will also run using Microsoft Windows XP Professional Edition with Service Pack 2

Important If you are using Windows XP, some of the dialog boxes and screen shots described in this book might look a little different from those that you see. This is because of differences in the user interface in the Windows Vista operating system and the way in which Windows Vista manages security.

- Microsoft Visual Studio 2008 Standard Edition, Visual Studio 2008 Enterprise Edition, or Microsoft Visual C# 2008 Express Edition and Microsoft Visual Web Developer 2008 Express Edition
- Microsoft SQL Server 2005 Express Edition, Service Pack 2
- 1.6-GHz Pentium III+ processor, or faster
- 1 GB of available, physical RAM
- Video (800 × 600 or higher resolution) monitor with at least 256 colors
- CD-ROM or DVD-ROM drive
- Microsoft mouse or compatible pointing device

You will also need to have Administrator access to your computer to configure SQL Server 2005 Express Edition and to perform the exercises.

Code Samples

The companion CD inside this book contains the code samples that you'll use as you perform the exercises. By using the code samples, you won't waste time creating files that aren't relevant to the exercise. The files and the step-by-step instructions in the lessons also let you learn by doing, which is an easy and effective way to acquire and remember new skills.

Installing the Code Samples

Follow these steps to install the code samples and required software on your computer so that you can use them with the exercises.

1. Remove the companion CD from the package inside this book and insert it into your CD-ROM drive.

Note An end-user license agreement should open automatically. If this agreement does not appear, open My Computer on the desktop or Start menu, double-click the icon for your CD-ROM drive, and then double-click StartCD.exe.

2. Review the end-user license agreement. If you accept the terms, select the accept option, and then click *Next*.

   A menu will appear with options related to the book.

3. Click *Install Code Samples*.

4. Follow the instructions that appear.

   The code samples are installed to the following location on your computer:

   Documents\Microsoft Press\Visual CSharp Step By Step

Using the Code Samples

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

Important The code samples have been tested by using an account that is a member of the local Administrators group. It is recommended that you perform the exercises by using an account that has Administrator rights.

For those of you who like to know all the details, here's a list of the code sample Visual Studio 2008 projects and solutions, grouped by the folders where you can find them.

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.
Chapter 2	
PrimitiveDataTypes	This project demonstrates how to declare variables by using each of the primitive types, how to assign values to these variables, and how to display their values in a window.
MathsOperators	This program introduces the arithmetic operators (+ – * / %).

Project	Description
Chapter 3	
Methods	In this project, you'll reexamine the code in the previous project and investigate how it uses methods to structure the code.
DailyRate	This project walks you through writing your own methods, running the methods, and stepping through the method calls by using the Visual Studio 2008 debugger.
Chapter 4	
Selection	This project shows 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 uses a *while* statement to read the contents of a source file one line at a time and display 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 reexamines the MathsOperators project from Chapter 2, "Working with Variables, Operators, and Expressions," and causes various unhandled exceptions to make the program fail. The *try* and *catch* keywords then make the application more robust so that it no longer fails.
Chapter 7	
Classes	This project covers the basics of defining your own classes, complete with public constructors, methods, and private fields. It also shows how to create class instances by using the *new* keyword and how to define static methods and fields.
Chapter 8	
Parameters	This program investigates the difference between value parameters and reference parameters. It demonstrates how to use the *ref* and *out* keywords.
Chapter 9	
StructsAndEnums	This project defines a *struct* type to represent a calendar date.

Project	Description
Chapter 10	
Cards	This project uses the *ArrayList* collection class to group together playing cards in a hand.
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	
Tokenizer	This project uses a hierarchy of interfaces and classes to simulate both reading a C# source file and classifying its contents into various kinds of tokens (identifiers, keywords, operators, and so on). As an example of use, it also derives classes from the key interfaces to display the tokens in a rich text box in color syntax.
Chapter 14	
UsingStatement	This project revisits a small piece of code from Chapter 5, "Using Compound Assignment and Iteration Statements," and reveals that it is not exception-safe. It shows you how to make the code exception-safe with a *using* statement.
Chapter 15	
WindowProperties	This project presents a simple Windows application that uses several properties to display the size of its main window. The display updates automatically as the user resizes the window.
AutomaticProperties	This project shows how to create automatic properties for a class and use them to initialize instances of the class.
Chapter 16	
Indexers	This project uses two indexers: one to look up a person's phone number when given a name, and the other to look up a person's name when given a phone number.
Chapter 17	
Delegates	This project displays the time in digital format by using delegate callbacks. The code is then simplified by using events.

Project	Description
Chapter 18	
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 19	
BinaryTree	This project shows you how to implement the generic IEnumerator<T> interface to create an enumerator for the generic BinaryTree class.
IteratorBinaryTree	This solution uses an iterator to generate an enumerator for the generic BinaryTree class.
Chapter 20	
QueryBinaryTree	This project shows how to use LINQ queries to retrieve data from a binary tree object.
Chapter 21	
Operators	This project builds three structs, called Hour, Minute, and Second, that contain user-defined operators. The code is then simplified by using a conversion operator.
Chapter 22	
BellRingers	This project is a Windows Presentation Foundation application demonstrating how to define styles and use basic WPF controls.
Chapter 23	
BellRingers	This project is an extension of the application created in Chapter 22, "Introducing Windows Presentation Foundation," but with drop-down and pop-up menus added to the user interface.
Chapter 24	
CustomerDetails	This project demonstrates how to implement business rules for validating user input in a WPF application using customer information as an example.
Chapter 25	
ReportOrders	This project shows how to access a database by using ADO.NET code. The application retrieves information from the Orders table in the Northwind database.
DLINQOrders	This project shows how to use DLINQ to access a database and retrieve information from the Orders table in the Northwind database.

Project	Description
Chapter 26	
Suppliers	This project demonstrates how to use data binding with a WPF application to display and format data retrieved from a database in controls on a WPF form. The application also enables the user to modify information in the Products table in the Northwind database.
Chapter 27	
Litware	This project creates a simple Microsoft ASP.NET Web site that enables the user to input information about employees working for a fictitious software development company.
Chapter 28	
Litware	This project is an extended version of the Litware project from the previous chapter and shows how to validate user input in an ASP.NET Web application.
Chapter 29	
Northwind	This project shows how to use Forms-based security for authenticating the user. The application also demonstrates how to use ADO.NET from an ASP.NET Web form, showing how to query and update a database in a scalable manner, and how to create applications that span multiple Web forms.
Chapter 30	
NorthwindServices	This project implements a Windows Communication Foundation Web service, providing remote access across the Internet to data in the *Products* table in the Northwind database.

Uninstalling the Code Samples

Follow these steps to remove the code samples from your computer.

1. In *Control Panel*, open *Add or Remove Programs*.
2. From the list of Currently Installed Programs, select Microsoft Visual C# 2008 Step by Step.
3. Click *Remove*.
4. Follow the instructions that appear to remove the code samples.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books and companion CDs at the following Web site:

http://www.microsoft.com/learning/support/books/

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion CD, or questions that are not answered by visiting the site above, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: *Microsoft Visual C# 2008 Step by Step* Series Editor
One Microsoft Way
Redmond, WA 98052-6399

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

Chapter 1

Welcome to C#

After completing this chapter, you will be able to:

- Use the Microsoft Visual Studio 2008 programming environment.
- Create a C# console application.
- Explain the purpose of namespaces.
- Create a simple graphical C# application.

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

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

Beginning Programming with the Visual Studio 2008 Environment

Visual Studio 2008 is a tool-rich programming environment containing all the functionality you need to create large or small C# projects. You can even create projects that seamlessly combine modules compiled using different programming languages. In the first exercise, you start the Visual Studio 2008 programming environment and learn how to create a console application.

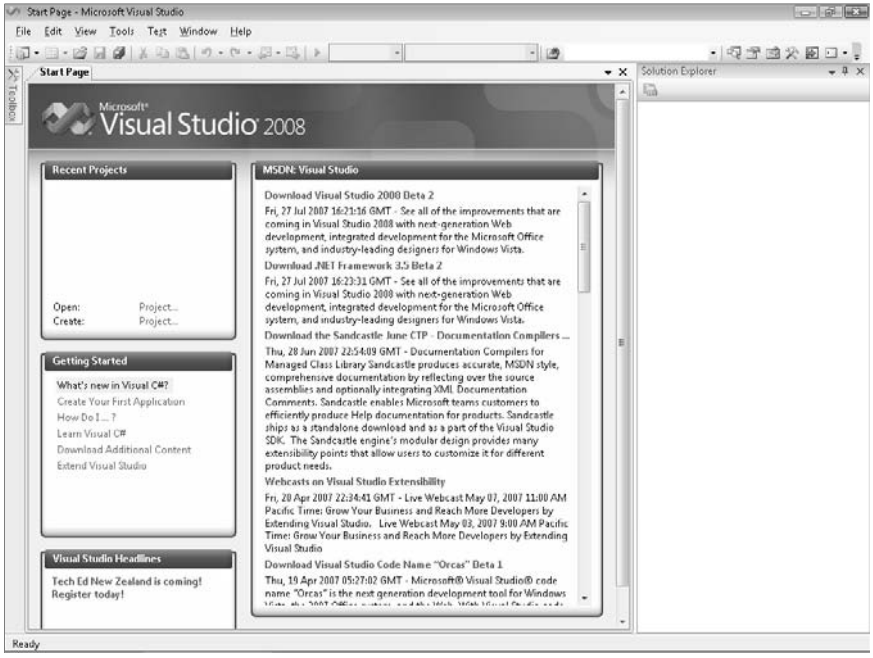
Note A console application is an application that runs in a command prompt window, rather than providing a graphical user interface.

Create a console application in Visual Studio 2008

- If you are using Visual Studio 2008 Standard Edition or Visual Studio 2008 Professional Edition, perform the following operations to start Visual Studio 2008:

  1. On the Microsoft Windows task bar, click the *Start* button, point to *All Programs*, and then point to the *Microsoft Visual Studio 2008* program group.
  2. In the Microsoft Visual Studio 2008 program group, click *Microsoft Visual Studio 2008*.

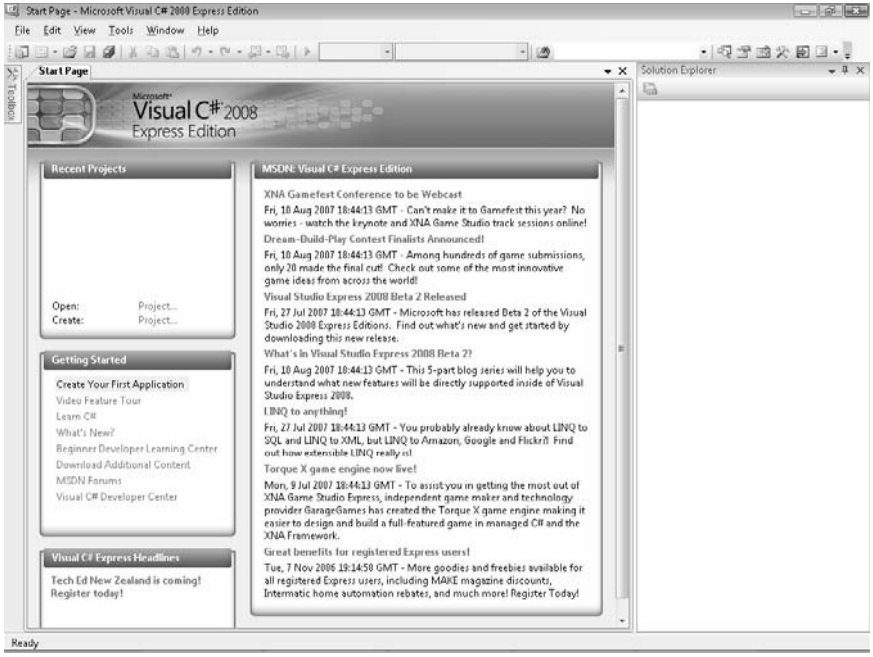
  Visual Studio 2008 starts, like this:



Note If this is the first time you have run Visual Studio 2008, you might see a dialog box prompting you to choose your default development environment settings. Visual Studio 2008 can tailor itself according to your preferred development language. The various dialog boxes and tools in the integrated development environment (IDE) will have their default selections set for the language you choose. Select *Visual C# Development Settings* from the list, and then click the *Start Visual Studio* button. After a short delay, the Visual Studio 2008 IDE appears.

- If you are using Visual C# 2008 Express Edition, on the Microsoft Windows task bar, click the *Start* button, point to *All Programs*, and then click *Microsoft Visual C# 2008 Express Edition*.

Visual C# 2008 Express Edition starts, like this:



Note To avoid repetition, throughout this book, I simply state, "Start Visual Studio" when you need to open Visual Studio 2008 Standard Edition, Visual Studio 2008 Professional Edition, or Visual C# 2008 Express Edition. Additionally, unless explicitly stated, all references to Visual Studio 2008 apply to Visual Studio 2008 Standard Edition, Visual Studio 2008 Professional Edition, and Visual C# 2008 Express Edition.

- If you are using Visual Studio 2008 Standard Edition or Visual Studio 2008 Professional Edition, perform the following tasks to create a new console application.

  1. 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.

  2. In the *Project types* pane, click *Visual C#*. In the *Templates* pane, click the *Console Application* icon.

3. In the *Location* field, if you are using the Windows Vista operating system, type **C:\Users*YourName*\Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 1**. If you are using Microsoft Windows XP or Windows Server 2003, type **C:\Documents and Settings*YourName*\My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1**.

   Replace the text *YourName* in these paths with your Windows user name.

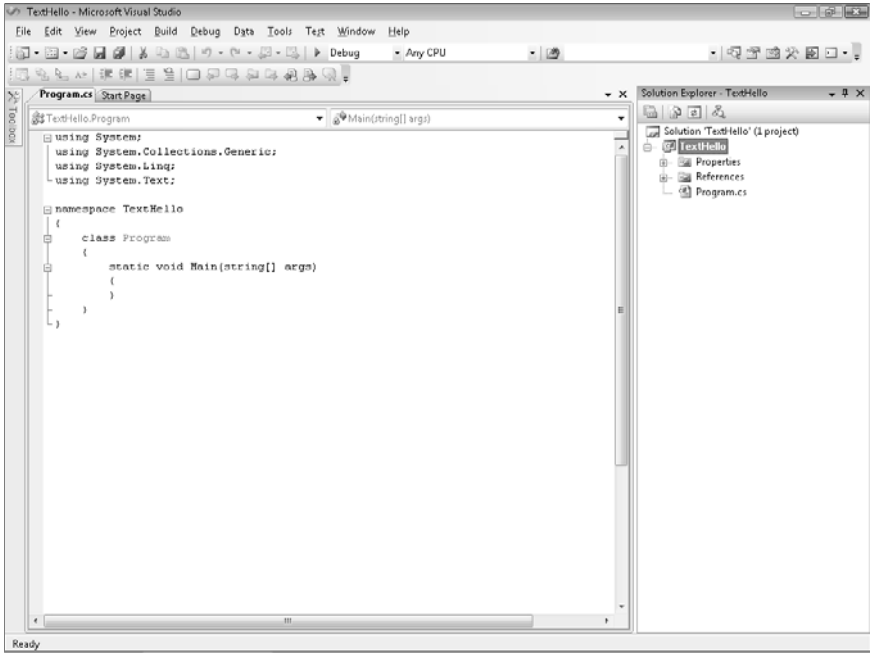
Note To save space throughout the rest of this book, I will simply refer to the path "C:\Users*YourName*\Documents" or "C:\Documents and Settings*YourName*\My Documents" as your Documents folder.

Tip If the folder you specify does not exist, Visual Studio 2008 creates it for you.

4. In the *Name* field, type **TextHello**.
5. Ensure that the *Create directory for solution* check box is selected, and then click *OK*.

- If you are using Visual C# 2008 Express Edition, the *New Project* dialog box won't allow you to specify the location of your project files; it defaults to the C:\Users*YourName*\AppData\Local\Temporary Projects folder. Change it by using the following procedure:
  1. On the *Tools* menu, click *Options*.
  2. In the *Options* dialog box, turn on the *Show All Settings* check box, and then click *Projects and Solutions* in the tree view in the left pane.
  3. In the right pane, in the *Visual Studio projects location* text box, specify the *Microsoft Press\Visual CSharp Step By Step\Chapter 1* folder under your Documents folder.
  4. Click *OK*.
- If you are using Visual C# 2008 Express Edition, perform the following tasks to create a new console application.
  1. On the *File* menu, click *New Project*.
  2. In the *New Project* dialog box, click the *Console Application* icon.
  3. In the *Name* field, type **TextHello**.
  4. 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 and provides button shortcuts to run the most frequently used commands. The *Code and Text Editor* window occupying the main part of the IDE displays the contents of source files. In a multi-file 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* 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* to bring that source file to the foreground in the *Code and Text Editor* window.

Before writing the code, examine the files listed in the *Solution Explorer*, which Visual Studio 2008 has created as part of your project:

- **Solution 'TextHello'** This is the top-level solution file, of which there is one per application. If you use Windows Explorer to look at your Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1\TextHello folder, you'll see that the actual name of this file is TextHello.sln. Each solution file contains references to one or more project files.

- **TextHello** This is the C# project file. Each project file references one or more files containing the source code and other items for the project. All the source code in a single project must be written in the same programming language. In Windows Explorer, this file is actually called TextHello.csproj, and it is stored in your \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1\TextHello\TextHello folder.
- **Properties** This is a folder in the TextHello project. If you expand it, 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. Learning how to use these attributes is outside the scope of this book.
- **References** This is a folder that contains references to compiled code that your application can use. When code is compiled, it is converted into an assembly and given a unique name. Developers use assemblies to package useful bits of code they have written so they can distribute it to other developers who might want to use the code in their applications. Many of the features that you will be using when writing applications using this book make use of assemblies provided by Microsoft with Visual Studio 2008.
- **Program.cs** This is a C# source file and is the one currently 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 2008 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*. All methods must be defined inside a class. You will learn more about classes in Chapter 7, "Creating and Managing Classes and Objects." The *Main* method is special—it designates the program's entry point. It must be a static method. (You will look at methods in detail in Chapter 3, "Writing Methods and Applying Scope," and I discuss static methods in Chapter 7.)

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

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

Write the code by using 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. On the new line, type the word **Console**, which is the name of a built-in class. As you type the letter *C* at the start of the word *Console*, an IntelliSense list appears. 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 *Con*, the IntelliSense list will automatically home 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 that contains the methods for displaying messages on the screen and getting input from the keyboard.

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

3. 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
}
```

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

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

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

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

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

   Main should now look like this:

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

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

   Main should now look like this:

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

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

IntelliSense Icons

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

Icon	Meaning
	method (discussed in Chapter 3)
	property (discussed in Chapter 15)
	class (discussed in Chapter 7)
	struct (discussed in Chapter 9)
	enum (discussed in Chapter 9)

Icon	Meaning
	interface (discussed in Chapter 13)
	delegate (discussed in Chapter 17)
	extension method (discussed in Chapter 12)

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

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

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

The compiler will skip 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 will skip 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.

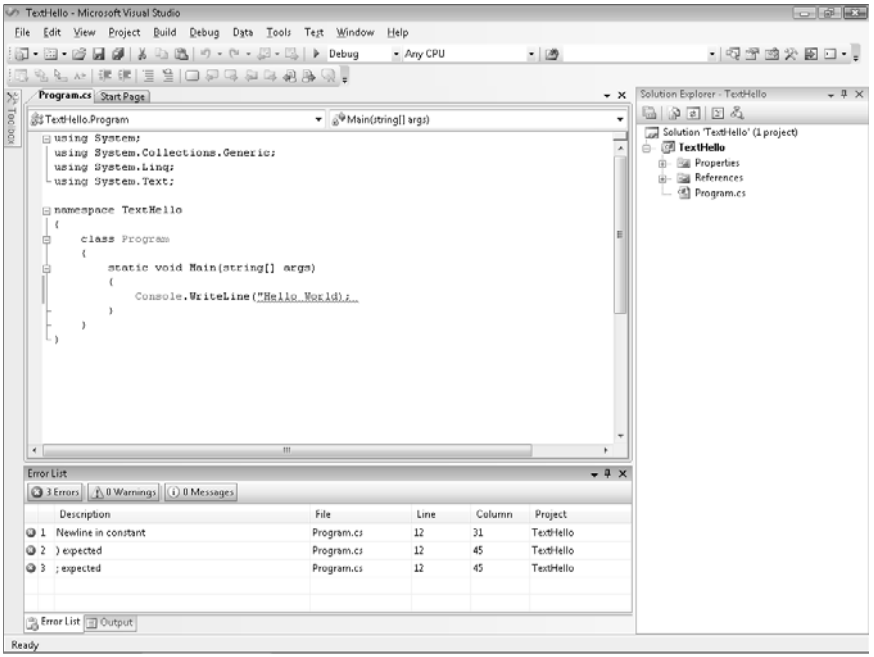
Tip If the *Output* window does not appear, on the *View* menu, click *Output* to display it.

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

```
------ Build started: Project: TextHello, Configuration: Debug Any CPU ----
C:\Windows\Microsoft.NET\Framework\v3.5\Csc.exe /config /nowarn:1701;1702 …
Compile complete -- 0 errors, 0 warnings
TextHello -> C:\Documents and Settings\John\My Documents\Microsoft Press\…
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ========
```

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

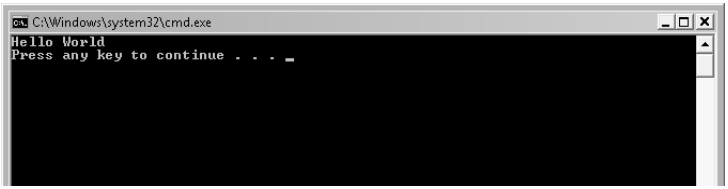
If you have followed the previous instructions carefully, there should be no errors or warnings, and the program should build successfully.

Tip There is no need to save the file explicitly before building because the *Build Solution* command automatically saves the file. If you are using Visual Studio 2008 Standard Edition or Visual Studio 2008 Professional Edition, the project is saved in the location specified when you created it. If you are using Visual C# 2008 Express Edition, the project is saved in a temporary location and is copied to the folder you specified in the *Options* dialog box only when you explicitly save the project by using the *Save All* command on the *File* menu or when you close Visual C# 2008 Express Edition.

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

2. On the *Debug* menu, click *Start Without Debugging*.

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



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

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

   The command window closes, and you return to the Visual Studio 2008 programming environment.

4. In *Solution Explorer*, click the TextHello project (not the solution), and then click the *Show All Files* toolbar button on the *Solution Explorer* toolbar—this is the second button from the left on the toolbar in the Solution Explorer window.

   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\TextHello\TextHello). Visual Studio creates these folders when you build your application, and they contain the executable version of the program together with some other files used to build and debug the application.

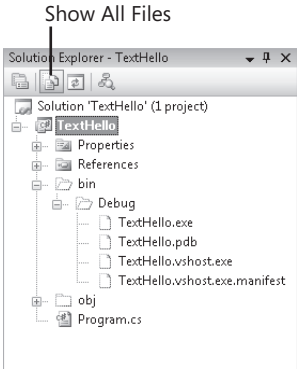
5. In *Solution Explorer*, click the plus sign (**+**) to the left of the *bin* entry.

   Another folder named *Debug* appears.

Note You may also see a folder called *Release*.

6. In *Solution Explorer*, click the plus sign (**+**) to the left of the *Debug* folder.

   Four more items named TextHello.exe, TextHello.pdb, TextHello.vshost.exe, and TextHello.vshost.exe.manifest appear, like this:



Note If you are using Visual C# 2008 Express Edition, you might not see all of these files.

The file TextHello.exe is the compiled program, and it is this file that runs when you click *Start Without Debugging* on the *Debug* menu. The other files contain information that is used by Visual Studio 2008 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 programs. Second, more code usually means more names, more methods, and more classes. As the number of names increases, so does the likelihood of the project build failing because two or more names clash (especially when a program also uses third-party libraries written by 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 solution is not a good one 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 named container for other identifiers, 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 *TextHello*, like this:

```
namespace TextHello
{
    class Greeting
    {
        ...
    }
}
```

You can then refer to the *Greeting* class as *TextHello.Greeting* in your programs. If another developer also creates a *Greeting* class in a different namespace, such as *NewNamespace*, and installs it on your computer, your programs will still work as expected because they are using the *TextHello.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 2008 environment follows this recommendation by using the name of your project as the top-level namespace. The .NET Framework software development kit (SDK) also adheres to this recommendation; every class in the .NET Framework lives inside a namespace. For example, the *Console* class lives inside the *System* namespace. This means that its full name is actually *System.Console*.

Of course, if you had to write the full name of a class every time you used it, the situation would be no better than prefixing qualifiers or even just naming the class with some globally unique name such *SystemConsole* and not bothering with a namespace. Fortunately, you can solve this problem with a *using* directive in your programs. If you return to the TextHello program in Visual Studio 2008 and look at the file Program.cs in the *Code and Text Editor* window, you will notice the following statements at the top of the file:

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

A *using* statement brings a namespace into scope. In subsequent code in the same file, you no longer have to explicitly qualify objects with the namespace to which they belong. The four namespaces shown contain classes that are used so often that Visual Studio 2008 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.

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 selected 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 *System.* the names of all the items in the *System* namespace are displayed by IntelliSense.

5. On the *Build* menu, click *Build Solution*.

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

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

Namespaces and Assemblies

A *using* statement 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 many thousands of classes. If they were all held in the same assembly, the assembly would be huge and difficult to maintain. (If Microsoft updated a single method in a single class, it would have to distribute the entire class library to all developers!)

For this reason, the .NET Framework class library is split into a number of assemblies, partitioned by the functional area to which the classes they contain relate. For example, there is a "core" assembly that contains all the common classes, such as *System.Console*, and there are further assemblies that contain classes for manipulating databases, accessing Web services, building graphical user interfaces, 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 for multiple namespaces, and a single namespace can span multiple assemblies. 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 TextHello project, click the plus sign (**+**) to the left of the *References* folder. You will see that a Console application automatically includes references to assemblies called *System*, *System.Core*, *System.Data*, and *System.Xml*. You can add references for additional assemblies to a project by right-clicking the *References* folder and clicking *Add Reference*—you will practice performing this task in later exercises.

Creating a Graphical Application

So far, you have used Visual Studio 2008 to create and run a basic Console application. The Visual Studio 2008 programming environment also contains everything you need to create graphical Windows-based applications. You can design the form-based user interface of a Windows-based application interactively. Visual Studio 2008 then generates the program statements to implement the user interface you've designed.

Visual Studio 2008 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 logic for a graphical application, and you use the *Design View* window to lay out your user interface. You can switch between the two views whenever you want.

In the following set of exercises, you'll learn how to create a graphical application by using Visual Studio 2008. This program will display a simple form containing a text box where you can enter your name and a button that displays a personalized greeting in a message box when you click the button.

Note Visual Studio 2008 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 it in preference to Windows Forms for all new development.

Create a graphical application in Visual Studio 2008

- If you are using Visual Studio 2008 Standard Edition or Visual Studio 2008 Professional Edition, perform the following operations to create a new graphical application:
  1. On the *File* menu, point to *New*, and then click *Project*.

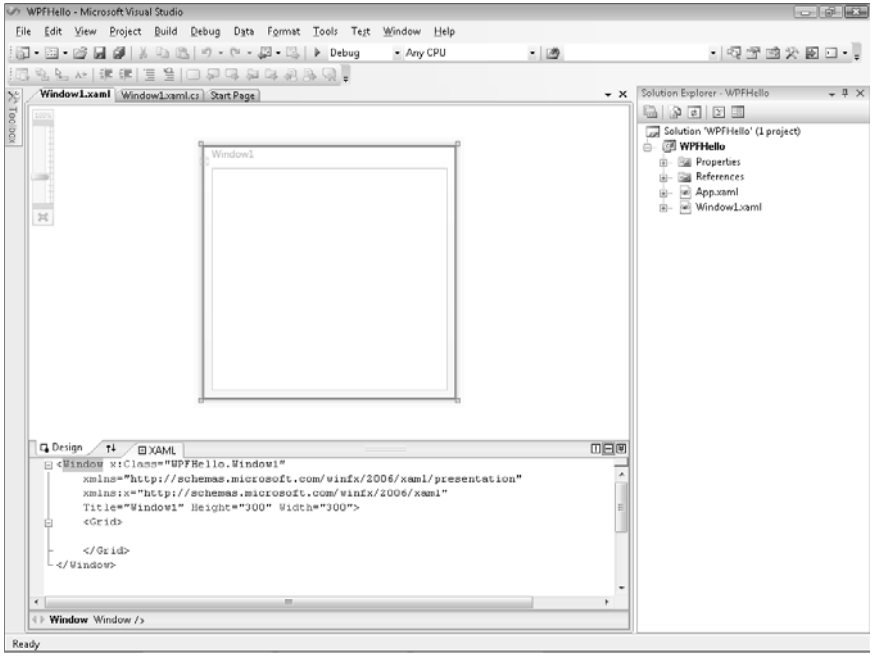
     The *New Project* dialog box opens.
  2. In the *Project Types* pane, click *Visual C#*.
  3. In the *Templates* pane, click the *WPF Application* icon.
  4. Ensure that the *Location* field refers to your *Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1* folder.
  5. In the *Name* field, type **WPFHello**.
  6. In the *Solution* field, ensure that *Create new solution* is selected.

     This action creates a new solution for holding the project. The alternative, *Add to Solution*, adds the project to the TextHello solution.
  7. Click *OK*.
- If you are using Visual C# 2008 Express Edition, perform the following tasks to create a new graphical application.
  1. On the *File* menu, click *New Project*.
  2. If the *New Project* message box appears, click *Save* to save your changes to the TextHello project. In the *Save Project* dialog box, verify that the *Location* field is set to *Microsoft Press\Visual CSharp Step By Step\Chapter 1* under your Documents folder, and then click *Save*.

3. In the *New Project* dialog box, click the *WPF Application* icon.
4. In the *Name* field, type **WPFHello**.
5. Click *OK*.

Visual Studio 2008 closes your current application and creates the new WPF application. It displays an empty WPF form in the *Design View* window, together with another window containing an XAML description of the form, as shown in the following graphic:

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



XAML stands for Extensible Application Markup Language and is an XML-like language used by WPF applications to define the layout of a form and its contents. If you have knowledge of XML, XAML should look familiar. You can actually define a WPF form completely by writing an XAML description if you don't like using the Design View window of Visual Studio or if you don't have access to Visual Studio; Microsoft provides an XAML editor called XMLPad that you can download free of charge from the MSDN Web site.

In the following exercise, you'll use the Design View window to add three controls to the Windows form and examine some of the C# code automatically generated by Visual Studio 2008 to implement these controls.

Create the user interface

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

   The Toolbox appears, partially obscuring the form, and displaying the various components and controls that you can place on a Windows form. The Common section displays a list of controls that are used by most WPF applications. The Controls section displays a more extensive list of controls.

2. In the Common section, click Label, and then click the visible part of the form.

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

Tip If you want the *Toolbox* to remain visible but not to hide any part of the form, click the *Auto Hide* button to the right in the *Toolbox* title bar (it looks like a pin). The *Toolbox* appears permanently on the left side of the Visual Studio 2008 window, and the *Design View* window shrinks to accommodate it. (You may 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.

3. The label 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 label control so that it is positioned toward the upper-left corner of the form. (The exact placement is not critical for this application.)

Note The XAML description of the form in the lower pane now includes the label control, together with properties such as its location on the form, governed by the *Margin* property. The *Margin* property consists of four numbers indicating the distance of each edge of the label from the edges of the form. If you move the control around the form, the value of the *Margin* property changes. If the form is resized, the controls anchored to the form's edges that move are resized to preserve their margin values. You can prevent this by setting the *Margin* values to zero. You learn more about the *Margin* and also the *Height* and *Width* properties of WPF controls in Chapter 22, "Introducing Windows Presentation Foundation."

4. On the *View* menu, click *Properties Window*.

   The *Properties* window appears on the lower-right side of the screen, under *Solution Explorer* (if it was not already displayed). The *Properties* window provides another way for you to modify the properties for items on a form, as well as other items in a project. It is context sensitive in that it displays the properties for the currently selected item. If you click the title bar of the form displayed in the *Design View* window, you can see that the *Properties* window displays the properties for the form itself. If you click the label control, the window displays the properties for the label instead. If you click anywhere else on the form, the *Properties* window displays the properties for a mysterious

item called a *grid*. A grid acts as a container for items on a WPF form, and you can use the grid, among other things, to indicate how items on the form should be aligned and grouped together.

5. Click the label control on the form. In the *Properties* window, locate the *Text* section.

   By using the properties in this section, you can specify the font and font size for the label but not the actual text that the label displays.

6. Change the *FontSize* property to **20**, and then click the title bar of the form.

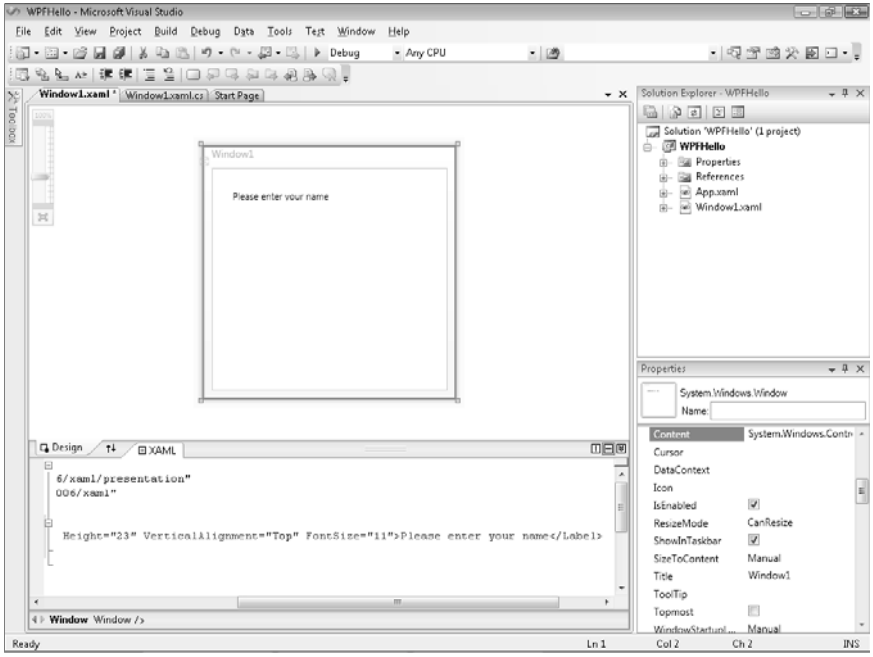
   The size of the text in the label changes, although the label is no longer big enough to display the text. Change the *FontSize* property back to **12**.

Note The text displayed in the label might not resize itself immediately in the *Design View* window. It will correct itself when you build and run the application, or if you close and open the form in the *Design View* window.

7. Scroll the XAML description of the form in the lower pane to the right, and examine the properties of the label control.

   The label control consists of a `<Label>` tag containing property values, followed by the text for the label itself ("Label"), followed by a closing `</Label>` tag.

8. Change the text Label (just before the closing tag) to **Please enter your name**, as shown in the following image.



Notice that the text displayed in the label on the form changes, although the label is still too small to display it correctly.

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

Note If you don't click the form in the *Design View* window, the *Toolbox* displays the message "There are no usable controls in this group."

10. In the *Toolbox*, click *TextBox*, and then click the form. A text box control is added to the form. Move the text box control so that it is directly underneath the label control.

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

11. While the text box control is selected, in the *Properties* window, change the value of the *Name* property displayed at the top of the window to **userName**.

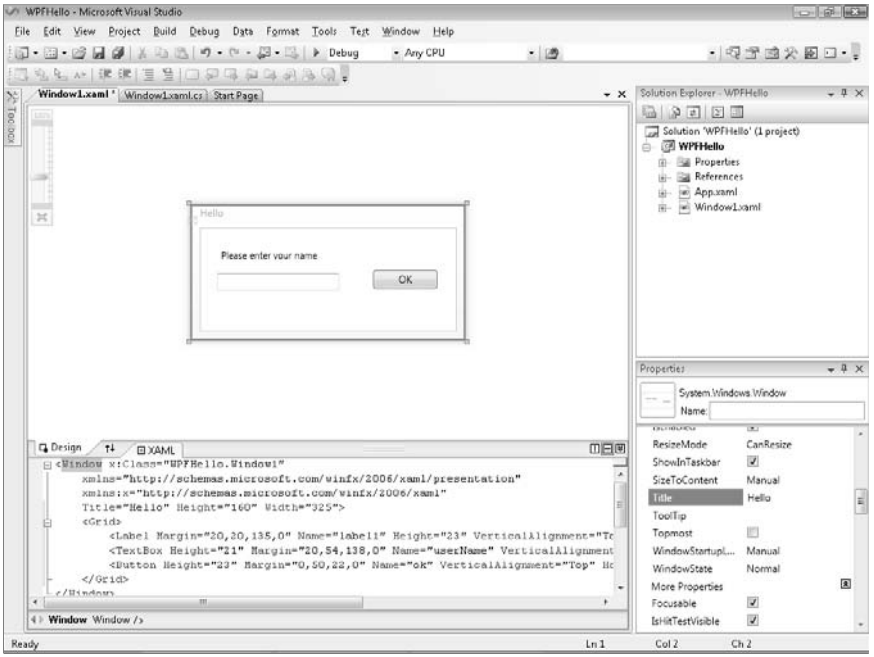
Note You will learn more about naming conventions for controls and variables in Chapter 2, "Working with Variables, Operators, and Expressions."

12. Display the *Toolbox* again, click *Button*, and then click the form. Drag the button control to the right of the text box control on the form so that the bottom of the button is aligned horizontally with the bottom of the text box.
13. Using the *Properties* window, change the *Name* property of the button control to **ok**.
14. In the XAML description of the form, scroll the text to the right to display the caption displayed by the button, and change it from Button to **OK**. Verify that the caption of the button control on the form changes.
15. Click the title bar of the Window1.xaml form in the *Design View* window. In the *Properties* window, change the *Title* property to **Hello**.
16. In the *Design View* window, notice that a resize handle (a small square) appears on the lower right-hand corner of the form when it is selected. Move the mouse pointer over the resize handle. When the pointer changes to a diagonal double-headed arrow, click and drag the pointer to resize the form. Stop dragging and release the mouse button when the spacing around the controls is roughly equal.

Important Click the title bar of the form and not the outline of the grid inside the 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.

Note If you make the form narrower, the *OK* button remains a fixed distance from the right-hand edge of the form, determined by its *Margin* property. If you make the form too narrow, the *OK* button will overwrite the text box control. The right-hand margin of the label is also fixed, and the text for the label will start to disappear when the label shrinks as the form becomes narrower.

The form should now look similar to this:



17. On the *Build* menu, click *Build Solution*, and verify that the project builds successfully.
18. On the *Debug* menu, click *Start Without Debugging*.

    The application should run and display your form. You can type your name in the text box and click *OK*, but nothing happens yet. You need to add some code to process the *Click* event for the *OK* button, which is what you will do next.
19. Click the *Close* button (the *X* in the upper-right corner of the form) to close the form and return to Visual Studio.

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 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 form. Before adding your own code to the application, it helps to have an understanding of what Visual Studio has generated for you.

In *Solution Explorer*, click the plus sign (**+**) beside the file Window1.xaml. The file Window1.xaml.cs appears. Double-click the file Window1.xaml.cs. The code for the form is displayed in the *Code and Text Editor* window. It looks like this:

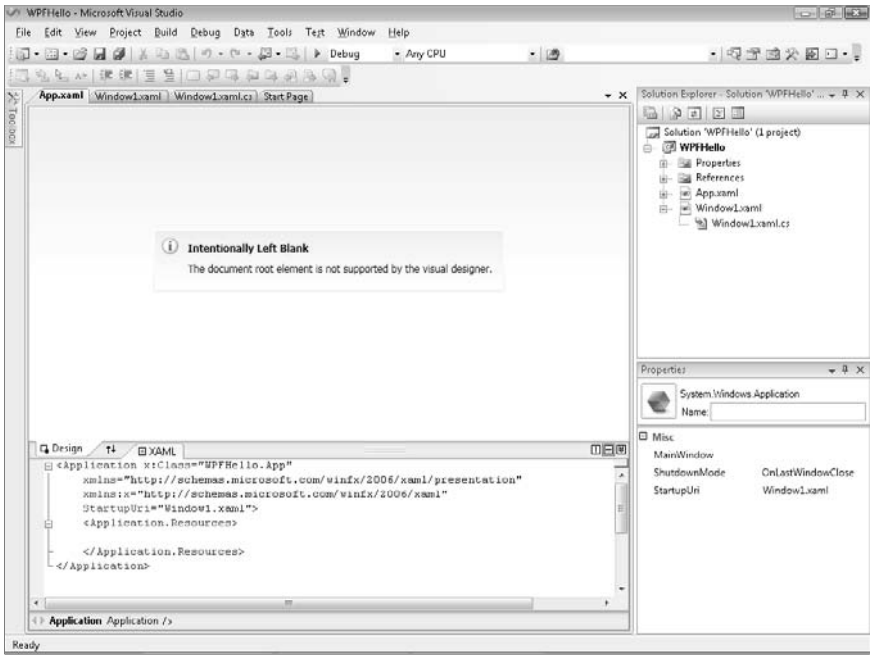
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WPFHello
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>

    public partial class Window1 : Window
    {

        public Window1()
        {
            InitializeComponent();
        }

    }
}
```

Apart from a good number of *using* statements bringing into scope some namespaces that most WPF applications use, the file contains the definition of a class called *Window1* but not much else. There is a little bit of code for the *Window1* class known as a constructor that calls a method called *InitializeComponent*, but that is all. (A *constructor* is a special method with the same name as the class. It is executed when an instance of the class is created and can contain code to initialize the instance. You will learn about constructors in Chapter 7.) In fact, the application contains a lot more code, but most 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.

The purpose of the code that you *can* see in this class is so that you can add your own methods to handle the logic for your application, such as what happens when the user clicks the *OK* button.

Tip You can also display the C# code file for a WPF form by right-clicking anywhere in the *Design View* window and then clicking *View Code*.

At this point you might well be wondering where the *Main* method is and how the form gets displayed when the application runs; remember that *Main* defines the point at which the program starts. In *Solution Explorer*, you should notice another source file called App.xaml. If you double-click this file, the *Design View* window displays the message "Intentionally Left Blank," but the file has an XAML description. One property in the XAML code is called *StartupUri*, and it refers to the Window1.xaml file as shown here:



If you click the plus sign (**+**) adjacent to App.xaml in *Solution Explorer*, you will see that there is also an Application.xaml.cs file. If you double-click this file, you will find it contains the following code:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Windows;
```

```
namespace WPFHello
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>

    public partial class App : Application
    {

    }
}
```

Once again, there are a number of *using* statements, but not a lot else, not even a *Main* method. In fact, *Main* is there, but it is also hidden. The code for *Main* is generated based on the settings in the App.xaml file; in particular, *Main* will create and display the form specified by the *StartupUri* property. If you want to display a different form, you edit the App.xaml file.

The time has come to write some code for yourself!

Write the code for the OK button

1. Click the *Window1.xaml* tab above the *Code and Text Editor* window to display Window1 in the *Design View* window.
2. Double-click the *OK* button on the form.

   The Window1.xaml.cs file appears in the *Code and Text Editor* window, but a new method has been added called *ok_Click*. Visual Studio automatically generates code to call this method whenever the user clicks the *OK* button. This is an example of an event, and you will learn much more about how events work as you progress through this book.
3. Add the code shown in bold type to the *ok_Click* method:

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

   This is the code that will run when the user clicks the *OK* button. Do not worry too much about the syntax of this code just yet (just make sure you copy it exactly as shown) because you will learn all about methods in Chapter 3. The interesting part is the *MessageBox.Show* statement. This statement displays a message box containing the text "Hello" with whatever name the user typed into the username text box on the appended form.
4. Click the *Window1.xaml* tab above the *Code and Text Editor* window to display Window1 in the *Design View* window again.

5. 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 contains an element called *Click* that refers to the *ok_Click* method:

```
<Button Height="23" … Click="ok_Click">OK</Button>
```

6. On the *Debug* menu, click *Start Without Debugging*.

7. When the form appears, type your name in the text box, and then click *OK*. A message box appears, welcoming you by name.

8. Click *OK* in the message box.

   The message box closes.

9. Close the form.

   - If you want to continue to the next chapter

     Keep Visual Studio 2008 running, and turn to Chapter 2.

   - If you want to exit Visual Studio 2008 now

     On the *File* menu, click *Exit*. If you see a *Save* dialog box, click *Yes* (if you are using Visual Studio 2008) or *Save* (if you are using Visual C# 2008 Express Edition) and save the project.

Chapter 1 Quick Reference

To	Do this	Key combination
Create a new console application using Visual Studio 2008 Standard or Professional Edition	On the *File* menu, point to *New*, and then click *Project* to open the *New Project* dialog box. For the project type, select *Visual C#*. For the template, select *Console Application*. Select a directory for the project files in the *Location* box. Choose a name for the project. Click *OK*.	
Create a new console application using Visual C# 2008 Express Edition	On the *Tools* menu, click *Options*. In the *Options* dialog box, click *Projects and Solutions*. In the *Visual Studio projects location* box, specify a directory for the project files. On the *File* menu, click *New Project* to open the *New Project* dialog box. For the template, select *Console Application*. Choose a name for the project. Click *OK*.	
Create a new graphical application using Visual Studio 2008 Standard or Professional Edition	On the *File* menu, point to *New*, and then click *Project* to open the *New Project* dialog box. For the project type, select Visual C#. For the template, select *WPF Application*. Select a directory for the project files in the *Location* box. Choose a name for the project. Click *OK*.	
Create a new graphical application using Visual C# 2008 Express Edition	On the *Tools* menu, click *Options*. In the *Options* dialog box, click *Projects and Solutions*. In the *Visual Studio projects location* box, specify a directory for the project files. On the *File* menu, click *New Project* to open the *New Project* dialog box. For the template, select *WPF Application*. Choose a name for the project. Click *OK*.	
Build the application	On the *Build* menu, click *Build Solution*.	F6
Run the application	On the *Debug* menu, click *Start Without Debugging*.	Ctrl+F5

Chapter 25

Querying Information in a Database

After completing this chapter, you will be able to:

- Fetch and display data from a Microsoft SQL Server database by using Microsoft ADO.NET.
- Define entity classes for holding data retrieved from a database.
- Use DLINQ to query a database and populate instances of entity classes.
- Create a custom *DataContext* class for accessing a database in a typesafe manner.

In Part IV of this book, "Working with Windows Applications," you learned how to use Microsoft Visual C# to build user interfaces and present and validate information. In Part V, you will learn about managing data by using the data access functionality available in Microsoft Visual Studio 2008 and the Microsoft .NET Framework. The chapters in this part of the book describe ADO.NET, a library of objects specifically designed to make it easy to write applications that use databases. In this chapter, you will also learn how to query data by using DLINQ—extensions to LINQ based on ADO.NET that are designed for retrieving data from a database. In Chapter 26, "Displaying and Editing Data by Using Data Binding," you will learn more about using ADO.NET and DLINQ for updating data.

Important To perform the exercises in this chapter, you must have installed Microsoft SQL Server 2005 Express Edition, Service Pack 2. This software is available on the retail DVD with Microsoft Visual Studio 2008 and Visual C# 2008 Express Edition and is installed by default.

Important It is recommended that you use an account that has Administrator privileges to perform the exercises in this chapter and the remainder of this book.

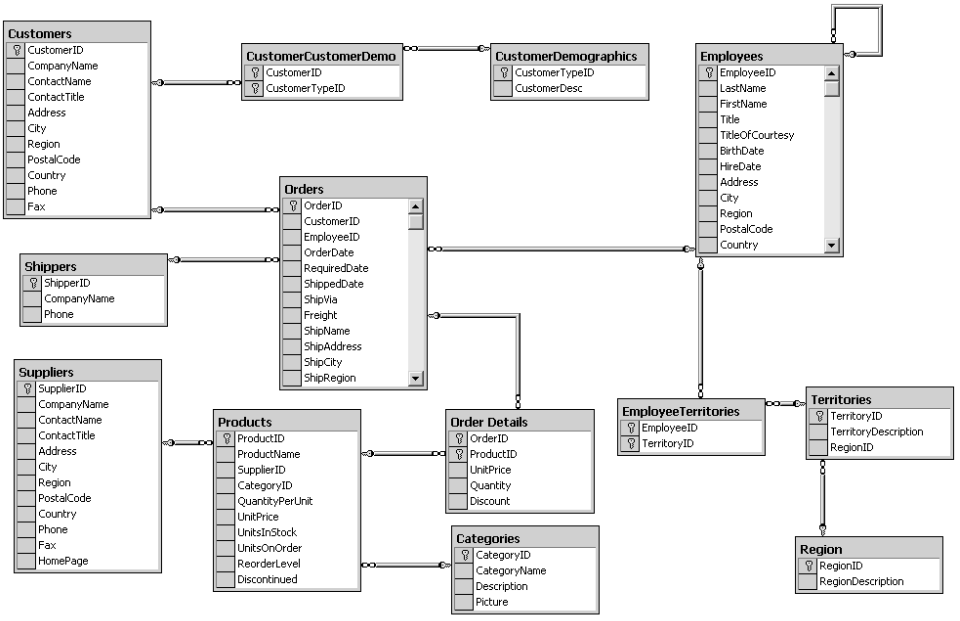
Querying a Database by Using ADO.NET

The ADO.NET class library contains a comprehensive framework for building applications that need to retrieve and update data held in a relational database. The model defined by ADO.NET is based on the notion of data providers. Each database management system (such as SQL Server, Oracle, IBM DB2, and so on) has its own data provider that implements an abstraction of the mechanisms for connecting to a database, issuing queries, and updating data. By using these abstractions, you can write portable code that is independent of the

underlying database management system. In this chapter, you will connect to a database managed by SQL Server 2005 Express Edition, but the techniques that you will learn are equally applicable when using a different database management system.

The Northwind Database

Northwind Traders is a fictitious company that sells edible goods with exotic names. The Northwind database contains several tables with information about the goods that Northwind Traders sells, the customers they sell to, orders placed by customers, suppliers from whom Northwind Traders obtains goods to resell, shippers that they use to send goods to customers, and employees who work for Northwind Traders. Figure 25-1 shows all the tables in the Northwind database and how they are related to one another. The tables that you will be using in this chapter are *Orders* and *Products*.



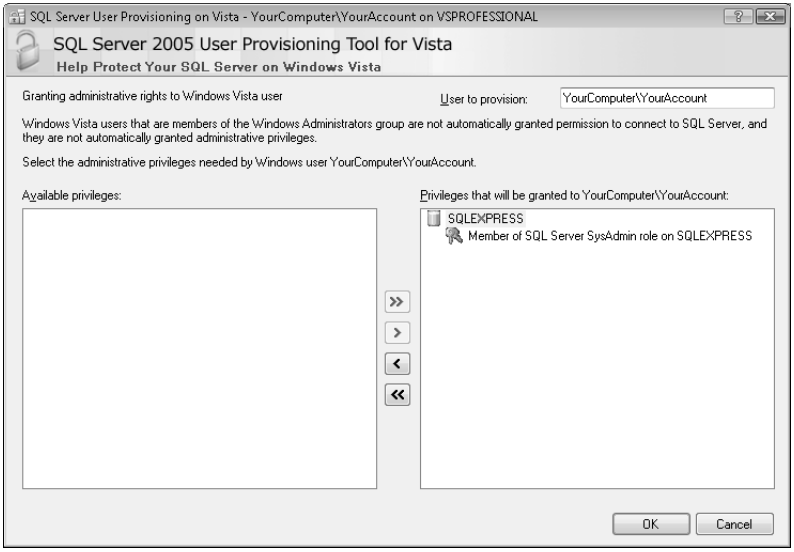
Creating the Database

Before proceeding further, you need to create the Northwind database.

Granting Permissions for Creating a SQL Server 2005 Database

You must have administrative rights for SQL Server 2005 Express before you can create a database. By default, if you are using the Windows Vista operating system, the computer *Administrator* account and members of the *Administrators* group do not have these rights. You can easily grant these permissions by using the SQL Server 2005 User Provisioning Tool for Vista, as follows:

1. Log on to your computer as an account that has administrator access.
2. Run the sqlprov.exe program, located in the folder C:\Program Files\Microsoft SQL Server\90\Shared.
3. In the *User Account Control* dialog box, click *Continue*. A console window briefly appears, and then the *SQL Server User Provisioning on Vista* window is displayed.
4. In the *User to provision* text box, type the name of the account you are using to perform the exercises. (Replace *YourComputer\YourAccount* with the name of your computer and your account.)
5. In the *Available privileges* box, click *Member of SQL Server SysAdmin role on SQLEXPRESS*, and then click the **>>** button.



6. Click *OK*.

   The permission will be granted to the specified user, and the SQL Server 2005 User Provisioning Tool for Vista will close automatically.

Create the Northwind database

1. On the Windows *Start* menu, click *All Programs*, click *Accessories*, and then click *Command Prompt* to open a command prompt window. If you are using Windows Vista, in the command prompt window type the following command to go to the \Microsoft Press\Visual CSharp Step by Step\Chapter 25 folder under your Documents folder. Replace *Name* with your user name.

   ```
   cd "\Users\Name\Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 25"
   ```

   If you are using Windows XP, type the following command to go to the \Microsoft Press\Visual CSharp Step by Step\Chapter 25 folder under your My Documents folder, replacing *Name* with your user name.

   ```
   cd "\Documents and Settings\Name\My Documents\Microsoft Press\Visual CSharp Step by
   Step\Chapter 25"
   ```

2. In the command prompt window, type the following command:

   ```
   sqlcmd -S YourComputer\SQLExpress -E -iinstnwnd.sql
   ```

   Replace *YourComputer* with the name of your computer.

   This command uses the sqlcmd utility to connect to your local instance of SQL Server 2005 Express and run the instnwnd.sql script. This script contains the SQL commands that create the Northwind Traders database and the tables in the database and fills them with some sample data.

Tip Ensure that SQL Server 2005 Express is running before you attempt to create the Northwind database. (It is set to start automatically by default. You will simply receive an error message if it is not started when you execute the *sqlcmd* command.) You can check the status of SQL Server 2005 Express, and start it running if necessary, by using the SQL Configuration Manager available in the Configuration Tools folder of the Microsoft SQL Server 2005 program group.

3. When the script finishes running, close the command prompt window.

Note You can run the command you executed in step 2 at any time if you need to reset the Northwind Traders database. The instnwnd.sql script automatically drops the database if it exists and then rebuilds it. See Chapter 26 for additional information.

Using ADO.NET to Query Order Information

In the next set of exercises, you will write code to access the Northwind database and display information in a simple console application. The aim of the exercise is to help you learn more about ADO.NET and understand the object model it implements. In later exercises, you will use DLINQ to query the database. In Chapter 26, you will see how to use the wizards included with Visual Studio 2008 to generate code that can retrieve and update data and display data graphically in a Windows Presentation Foundation (WPF) application.

The application you are going to create first will produce a simple report displaying information about customers' orders. The program will prompt the user for a customer ID and then display the orders for that customer.

Connect to the database

1. Start Visual Studio 2008 if it is not already running.
2. Create a new project called ReportOrders by using the Console Application template. Save it in the \Microsoft Press\Visual CSharp Step By Step\Chapter 25 folder under your Documents folder, and then click *OK*.

Note Remember, if you are using Visual C# 2008 Express Edition, you can specify the location for saving your project by setting the *Visual Studio projects location* in the *Projects and Solutions* section of the *Options* dialog box on the *Tools* menu.

3. In *Solution Explorer*, change the name of the file Program.cs to Report.cs. In the *Microsoft Visual Studio* message, click *Yes* to change all references of the *Program* class to *Report*.
4. In the *Code and Text Editor* window, add the following *using* statement to the list at the top of the file:

```
using System.Data.SqlClient;
```

   The *System.Data.SqlClient* namespace contains the SQL Server data provider classes for ADO.NET. These classes are specialized versions of the ADO.NET classes, optimized for working with SQL Server.

5. In the *Main* method of the *Report* class, add the following statement shown in bold type, which declares a *SqlConnection* object:

```
static void Main(string[] args)
{
    SqlConnection dataConnection = new SqlConnection();
}
```

SqlConnection is a subclass of an ADO.NET class called *Connection*. It is designed to handle connections to SQL Server databases.

6. After the variable declaration, add a *try/catch* block to the *Main* method. All the code that you will write for gaining access to the database goes inside the *try* part of this block. In the *catch* block, add a simple handler that catches *SqlException* exceptions. The new code is shown in bold type here:

```
static void Main(string[] args)
{
    ...
    try
    {
        // You will add your code here in a moment
    }
    catch(SqlException e)
    {
        Console.WriteLine("Error accessing the database: {0}", e.Message);
    }
}
```

A *SqlException* is thrown if an error occurs when accessing a SQL Server database.

7. Replace the comment in the *try* block with the code shown in bold type here:

```
try
{
    dataConnection.ConnectionString =
        "Integrated Security=true;Initial Catalog=Northwind;" +
        "Data Source=YourComputer\\SQLExpress";
    dataConnection.Open();
}
```

Important In the *ConnectionString* property, replace *YourComputer* with the name of your computer. Make sure that you type the string on a single line.

This code attempts to create a connection to the Northwind database. The contents of the *ConnectionString* property of the *SqlConnection* object contain elements that specify that the connection will use Windows Authentication to connect to the Northwind database on your local instance of SQL Server 2005 Express Edition. This is the preferred method of access because you do not have to prompt the user for any form of user name or password, and you are not tempted to hard-code user names and passwords into your application. Notice that a semicolon separates all the elements in the *ConnectionString*.

You can also encode many other elements in the connection string. See the documentation supplied with Visual Studio 2008 for details.

Using SQL Server Authentication

Windows Authentication is useful for authenticating users who are all members of a Windows domain. However, there might be occasions when the user accessing the database does not have a Windows account, for example, if you are building an application designed to be accessed by remote users over the Internet. In these cases, you can use the *User ID* and *Password* parameters instead, like this:

```
string userName = ...;
string password = ...;
// Prompt the user for their name and password, and fill these variables

string connString = String.Format(
    "User ID={0};Password={1};Initial Catalog=Northwind;"  +
    "Data Source=YourComputer\\SQLExpress", username, password);

myConnection.ConnectionString = connString;
```

At this point, I should offer a sentence of advice: never hard-code user names and passwords into your applications. Anyone who obtains a copy of the source code (or who reverse-engineers the compiled code) can see this information, and this renders the whole point of security meaningless.

The next step is to prompt the user for a customer ID and then query the database to find all of the orders for that customer.

Query the *Orders* table

1. Add the statements shown here in bold type to the *try* block after the *dataConnection.Open();* statement:

```
try
{
    ...
    Console.Write("Please enter a customer ID (5 characters): ");
    string customerId = Console.ReadLine();
}
```

   These statements prompt the user for a customer ID and read the user's response in the string variable *customerId*.

2. Type the following statements shown in bold type after the code you just entered:

```
try
{
    ...
    SqlCommand dataCommand = new SqlCommand();
    dataCommand.Connection = dataConnection;
```

```
    dataCommand.CommandText =
        "SELECT OrderID, OrderDate, ShippedDate, ShipName, ShipAddress, " +
        "ShipCity, ShipCountry " +
        "FROM Orders WHERE CustomerID='" + customerId + "'";
    Console.WriteLine("About to execute: {0}\n\n", dataCommand.CommandText);
}
```

The first statement creates a *SqlCommand* object. Like *SqlConnection*, this is a specialized version of an ADO.NET class, *Command*, that has been designed for performing queries against a SQL Server database. An ADO.NET *Command* object is used to execute a command against a data source. In the case of a relational database, the text of the command is a SQL statement.

The second line of code sets the *Connection* property of the *SqlCommand* object to the database connection you opened in the preceding exercise. The next two statements populate the *CommandText* property with a SQL SELECT statement that retrieves information from the *Orders* table for all orders that have a *CustomerID* that matches the value in the *customerId* variable. The *Console.WriteLine* statement just repeats the command about to be executed to the screen.

Important If you are an experienced database developer, you will probably be about to e-mail me telling me that using string concatenation to build SQL queries is bad practice. This approach renders your application vulnerable to SQL injection attacks. However, the purpose of this code is to quickly show you how to execute queries against a SQL Server database by using ADO.NET, so I have deliberately kept it simple. Do not write code such as this in your production applications.

For a description of what a SQL injection attack is, how dangerous it can be, and how you should write code to avoid such attacks, see the SQL Injection topic in SQL Server Books Online, available at *http://msdn2.microsoft.com/en-us/library/ms161953.aspx*.

3. Add the following statement shown in bold type after the code you just entered:

```
try
{
    ...
    SqlDataReader dataReader = dataCommand.ExecuteReader();
}
```

The *ExecuteReader* method of a *SqlCommand* object constructs a *SqlDataReader* object that you can use to fetch the rows identified by the SQL statement. The *SqlDataReader* class provides the fastest mechanism available (as fast as your network allows) for retrieving data from a SQL Server.

The next task is to iterate through all the orders (if there are any) and display them.

Fetch data and display orders

1. Add the *while* loop shown here in bold type after the statement that creates the *SqlDataReader* object:

```
try
{
    ...
    while (dataReader.Read())
    {
        // Code to display the current row
    }
}
```

The *Read* method of the *SqlDataReader* class fetches the next row from the database. It returns *true* if another row was retrieved successfully; otherwise, it returns *false*, usually because there are no more rows. The *while* loop you have just entered keeps reading rows from the *dataReader* variable and finishes when there are no more rows.

2. Add the statements shown here in bold type to the body of the *while* loop you created in the preceding step:

```
while (dataReader.Read())
{
    int orderId = dataReader.GetInt32(0);
    DateTime orderDate = dataReader.GetDateTime(1);
    DateTime shipDate = dataReader.GetDateTime(2);
    string shipName = dataReader.GetString(3);
    string shipAddress = dataReader.GetString(4);
    string shipCity = dataReader.GetString(5);
    string shipCountry = dataReader.GetString(6);
    Console.WriteLine(
        "Order: {0}\nPlaced: {1}\nShipped: {2}\n" +
        "To Address: {3}\n{4}\n{5}\n{6}\n\n", orderId, orderDate,
        shipDate, shipName, shipAddress, shipCity, shipCountry);
}
```

This block of code shows how you read the data from the database by using a *SqlDataReader* object. A *SqlDataReader* object contains the most recent row retrieved from the database. You can use the *GetXXX* methods to extract the information from each column in the row—there is a *GetXXX* method for each common type of data. For example, to read an *int* value, you use the *GetInt32* method; to read a string, you use the *GetString* method; and you can probably guess how to read a *DateTime* value. The *GetXXX* methods take a parameter indicating which column to read: 0 is the first column, 1 is the second column, and so on. The preceding code reads the various columns from the current *Orders* row, stores the values in a set of variables, and then prints out the values of these variables.

Firehose Cursors

One of the major drawbacks in a multiuser database application is locked data. Unfortunately, it is common to see applications retrieve rows from a database and keep those rows locked to prevent another user from changing the data while the application is using them. In some extreme circumstances, an application can even prevent other users from reading data that it has locked. If the application retrieves a large number of rows, it locks a large proportion of the table. If there are many users running the same application at the same time, they can end up waiting for one another to release locks and it all leads to a slow-running and frustrating mess.

The *SqlDataReader* class has been designed to remove this drawback. It fetches rows one at a time and does not retain any locks on a row after it has been retrieved. It is wonderful for improving concurrency in your applications. The *SqlDataReader* class is sometimes referred to as a "firehose cursor." (The term *cursor* is an acronym that stands for "current set of rows.")

When you have finished using a database, it's good practice to close your connection and release any resources you have been using.

Disconnect from the database, and test the application

1. Add the statement shown here in bold type after the *while* loop in the *try* block:

```
try
{
    ...
    while(dataReader.Read())
    {
        ...
    }
    dataReader.Close();
}
```

This statement closes the *SqlDataReader* object. You should always close a *SqlDataReader* object when you have finished with it because you will not able to use the current *SqlConnection* object to run any more commands until you do. It is also considered good practice to do it even if all you are going to do next is close the *SqlConnection*.

Note If you activate multiple active result sets (MARS) with SQL Server 2005, you can open more than one *SqlDataReader* object against the same *SqlConnection* object and process multiple sets of data. MARS is disabled by default. To learn more about MARS and how you can activate and use it, consult SQL Server 2005 Books Online.

2. After the *catch* block, add the following *finally* block:

```
catch(SqlException e)
{
    ...
}
finally
{
    dataConnection.Close();
}
```

Database connections are scarce resources. You need to ensure that they are closed when you have finished with them. Putting this statement in a *finally* block guarantees that the *SqlConnection* will be closed, even if an exception occurs; remember that the code in the *finally* block will be executed after the *catch* handler has finished.

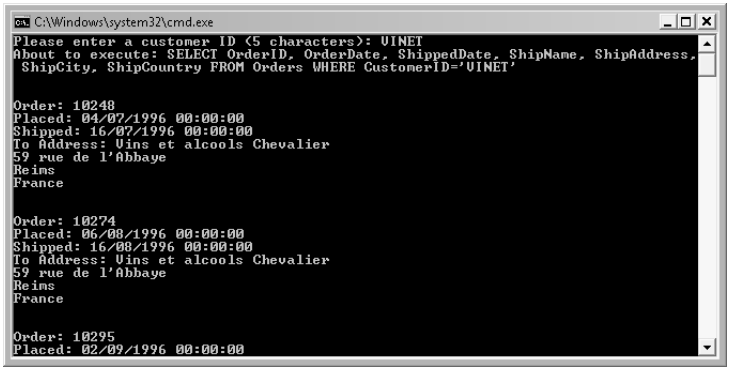
Tip An alternative approach to using a *finally* block is to wrap the code that creates the *SqlDataConnection* object in a *using* statement, as shown in the following code. At the end of the block defined by the *using* statement, the *SqlConnection* object is closed automatically, even if an exception occurs:

```
using (SqlConnection dataConnection = new SqlConnection())
{
    try
    {
        dataConnection.ConnectionString = "...";
        ...
    }
    catch (SqlException e)
    {
        Console.WriteLine("Error accessing the database: {0}", e.Message);
    }
}
```

3. On the *Debug* menu, click *Start Without Debugging* to build and run the application.

4. At the customer ID prompt, type the customer ID **VINET**, and press Enter.

The SQL SELECT statement appears, followed by the orders for this customer, as shown in the following image:



You can scroll back through the console window to view all the data. Press the Enter key to close the console window when you have finished.

5. Run the application again, and then type **BONAP** when prompted for the customer ID.

   Some rows appear, but then an error occurs. If you are using Windows Vista, a message box appears with the message "ReportOrders has stopped working." Click *Close program* (or *Close the program* if you are using Visual C# Express). If you are using Windows XP, a message box appears with the message "ReportOrders has encountered a problem and needs to close. We are sorry for the inconvenience." Click *Don't Send*.

   An error message containing the text "Data is Null. This method or property cannot be called on Null values" appears in the console window.

   The problem is that relational databases allow some columns to contain null values. A null value is a bit like a null variable in C#: It doesn't have a value, but if you try to read it, you get an error. In the *Orders* table, the *ShippedDate* column can contain a null value if the order has not yet been shipped. You should also note that this is a *SqlNullValueException* and consequently is not caught by the *SqlException* handler.

6. Press Enter to close the console window and return to Visual Studio 2008.

Closing Connections

In many older applications, you might notice a tendency to open a connection when the application starts and not close the connection until the application terminates. The rationale behind this strategy was that opening and closing database connections were expensive and time-consuming operations. This strategy had an impact on the scalability of applications because each user running the application had a connection to the database open while the application was running, even if the user went to lunch for a few hours. Most databases limit the number of concurrent connections that they allow. (Sometimes this is because of licensing, but usually it's because each connection consumes resources on the database server that are not infinite.) Eventually, the database would hit a limit on the number of users that could operate concurrently.

Most .NET Framework data providers (including the SQL Server provider) implement *connection pooling*. Database connections are created and held in a pool. When an application requires a connection, the data access provider extracts the next available connection from the pool. When the application closes the connection, it is returned to the pool and made available for the next application that wants a connection. This means that opening and closing database connections are no longer expensive operations. Closing a connection does not disconnect from the database; it just returns the connection to the pool. Opening a connection is simply a matter of obtaining an already-open connection from the pool. Therefore, you should not hold on to connections longer than you need to—open a connection when you need it, and close it as soon as you have finished with it.

You should note that the *ExecuteReader* method of the *SqlCommand* class, which creates a *SqlDataReader*, is overloaded. You can specify a *System.Data.CommandBehavior* parameter that automatically closes the connection used by the *SqlDataReader* when the *SqlDataReader* is closed, like this:

```
SqlDataReader dataReader =
    dataCommand.ExecuteReader(System.Data.CommandBehavior.CloseConnection);
```

When you read the data from the *SqlDataReader* object, you should check that the data you are reading is not null. You'll see how to do this next.

Handle null database values

1. In the *Main* method, change the code in the body of the *while* loop to contain an *if* ... *else* block, as shown here in bold type:

```
while (dataReader.Read())
{
    int orderId = dataReader.GetInt32(0);
    if (dataReader.IsDBNull(2))
    {
        Console.WriteLine("Order {0} not yet shipped\n\n", orderId);
    }
    else
    {
        DateTime orderDate = dataReader.GetDateTime(1);
        DateTime shipDate = dataReader.GetDateTime(2);
        string shipName = dataReader.GetString(3);
        string shipAddress = dataReader.GetString(4);
        string shipCity = dataReader.GetString(5);
        string shipCountry = dataReader.GetString(6);
        Console.WriteLine(
            "Order {0}\nPlaced {1}\nShipped{2}\n" +
            "To Address {3}\n{4}\n{5}\n{6}\n\n", orderId, orderDate,
            shipDate, shipName, shipAddress, shipCity, shipCountry);
    }
}
```

   The *if* statement uses the *IsDBNull* method to determine whether the *ShippedDate* column (column 2 in the table) is null. If it is null, no attempt is made to fetch it (or any of the other columns, which should also be null if there is no *ShippedDate* value); otherwise, the columns are read and printed as before.

2. Build and run the application again.

3. Type **BONAP** for the customer ID when prompted.

   This time you do not get any errors, but you receive a list of orders that have not yet been shipped.

4. When the application finishes, press Enter and return to Visual Studio 2008.

Querying a Database by Using DLINQ

In Chapter 20, "Querying In-Memory Data by Using Query Expressions," you saw how to use LINQ to examine the contents of enumerable collections held in memory. LINQ provides query expressions, which use SQL-like syntax for performing queries and generating a result set that you can then step through. It should come as no surprise that you can use an extended form of LINQ, called DLINQ, for querying and manipulating the contents of a database. DLINQ is built on top of ADO.NET. DLINQ provides a high level of abstraction, removing the need for you to worry about the details of constructing an ADO.NET *Command* object, iterating through a result set returned by a *DataReader* object, or fetching data column by column by using the various *GetXXX* methods.

Defining an Entity Class

You saw in Chapter 20 that using LINQ requires the objects that you are querying be enumerable; they must be collections that implement the *IEnumerable* interface. DLINQ can create its own enumerable collections of objects based on classes you define and that map directly to tables in a database. These classes are called *entity classes*. When you connect to a database and perform a query, DLINQ can retrieve the data identified by your query and create an instance of an entity class for each row fetched.

The best way to explain DLINQ is to see an example. The *Products* table in the Northwind database contains columns that contain information about the different aspects of the various products that Northwind Traders sells. The part of the instnwnd.sql script that you ran in the first exercise in this chapter contains a *CREATE TABLE* statement that looks similar to this (some of the columns, constraints, and other details have been omitted):

```
CREATE TABLE "Products" (
    "ProductID" "int" NOT NULL ,
    "ProductName" nvarchar (40) NOT NULL ,
    "SupplierID" "int" NULL ,
    "UnitPrice" "money" NULL,
    CONSTRAINT "PK_Products" PRIMARY KEY CLUSTERED ("ProductID"),
    CONSTRAINT "FK_Products_Suppliers" FOREIGN KEY ("SupplierID")
        REFERENCES "dbo"."Suppliers" ("SupplierID")
)
```

You can define an entity class that corresponds to the *Products* table like this:

```
[Table(Name = "Products")]
public class Product
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int ProductID { get; set; }

    [Column(CanBeNull = false)]
    public string ProductName { get; set; }
```

```
    [Column]
    public int? SupplierID { get; set; }

    [Column(DbType = "money")]
    public decimal? UnitPrice { get; set; }
}
```

The *Product* class contains a property for each of the columns in which you are interested in the *Products* table. You don't have to specify every column from the underlying table, but any columns that you omit will not be retrieved when you execute a query based on this entity class. The important points to note are the *Table* and *Column* attributes.

The *Table* attribute identifies this class as an entity class. The *Name* parameter specifies the name of the corresponding table in the database. If you omit the *Name* parameter, DLINQ assumes that the entity class name is the same as the name of the corresponding table in the database.

The *Column* attribute describes how a column in the *Products* table maps to a property in the *Product* class. The *Column* attribute can take a number of parameters. The ones shown in this example and described in the following list are the most common:

- The *IsPrimaryKey* parameter specifies that the property makes up part of the primary key. (If the table has a composite primary key spanning multiple columns, you should specify the *IsPrimaryKey* parameter for each corresponding property in the entity class.)
- The *DbType* parameter specifies the type of the underlying column in the database. In many cases, DLINQ can detect and convert data in a column in the database to the type of the corresponding property in the entity class, but in some situations you need to specify the data type mapping yourself. For example, the *UnitPrice* column in the *Products* table uses the SQL Server *money* type. The entity class specifies the corresponding property as a *decimal* value.

Note The default mapping of *money* data in SQL Server is to the *decimal* type in an entity class, so the *DbType* parameter shown here is actually redundant. However, I wanted to show you the syntax.

- The *CanBeNull* parameter indicates whether the column in the database can contain a null value. The default value for the *CanBeNull* parameter is *true*. Notice that the two properties in the *Product* table that correspond to columns that permit null values in the database (*SupplierID* and *UnitPrice*) are defined as nullable types in the entity class.

Note You can also use DLINQ to create new databases and tables based on the definitions of your entity classes by using the *CreateDatabase* method of the *DataContext* object. In the current version of DLINQ, the part of the library that creates tables uses the definition of the *DbType* parameter to specify whether a column should allow null values. If you are using DLINQ to create a new database, you should specify the nullability of each column in each table in the *DbType* parameter, like this:

```
[Column(DbType = "NVarChar(40) NOT NULL", CanBeNull = false)]
public string ProductName { get; set; }
...
[Column(DbType = "Int NULL", CanBeNull = true)]
public int? SupplierID { get; set; }
```

Like the *Table* attribute, the *Column* attribute provides a *Name* parameter that you can use to specify the name of the underlying column in the database. If you omit this parameter, DLINQ assumes that the name of the column is the same as the name of the property in the entity class.

Creating and Running a DLINQ Query

Having defined an entity class, you can use it to fetch and display data from the *Products* table. The following code shows the basic steps for doing this:

```
DataContext db = new DataContext("Integrated Security=true;" +
    "Initial Catalog=Northwind;Data Source=YourComputer\\SQLExpress");

Table<Product> products = db.GetTable<Product>();
var productsQuery = from p in products
                    select p;

foreach (var product in productsQuery)
{
    Console.WriteLine("ID: {0}, Name: {1}, Supplier: {2}, Price: {3:C}",
                      product.ProductID, product.ProductName,
                      product.SupplierID, product.UnitPrice);
}
```

Note Remember that the keywords *from*, *in*, and *select* in this context are C# identifiers. You must type them in lowercase.

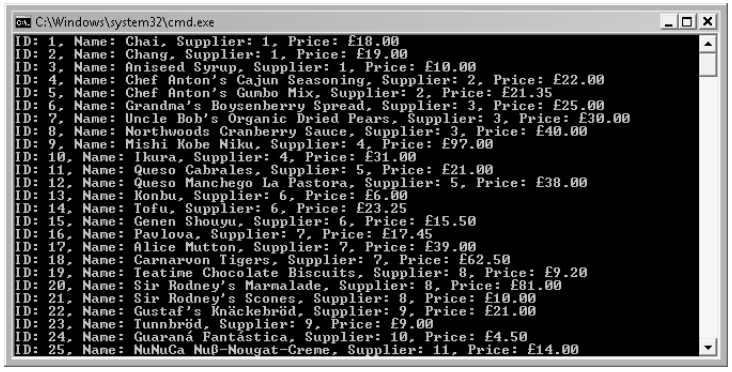
The *DataContext* class is responsible for managing the relationship between your entity classes and the tables in the database. You use it to establish a connection to the database and create collections of the entity classes. The *DataContext* constructor expects a connection string as a parameter, specifying the database that you want to use. This connection string is exactly the same as the connection string that you would use when connecting

through an ADO.NET *Connection* object. (The *DataContext* class actually creates an ADO.NET connection behind the scenes.)

The generic *GetTable<TEntity>* method of the *DataContext* class expects an entity class as its *TEntity* type parameter. This method constructs an enumerable collection based on this type and returns the collection as a *Table<TEntity>* type. You can perform DLINQ queries over this collection. The query shown in this example simply retrieves every object from the *Products* table.

Note If you need to recap your knowledge of LINQ query expressions, turn back to Chapter 20.

The *foreach* statement iterates through the results of this query and displays the details of each product. The following image shows the results of running this code. (The prices shown are per case, not per individual item.)



The *DataContext* object controls the database connection automatically; it opens the connection immediately prior to fetching the first row of data in the *foreach* statement and then closes the connection after the last row has been retrieved.

The DLINQ query shown in the preceding example retrieves every column for every row in the *Products* table. In this case, you can actually iterate through the *products* collection directly, like this:

```
Table<Product> products = db.GetTable<Product>();

foreach (Product product in products)
{
    ...
}
```

When the *foreach* statement runs, the *DataContext* object constructs a SQL SELECT statement that simply retrieves all the data from the *Products* table. If you want to retrieve a single row in the *Products* table, you can call the *Single* method of the *Products* entity class.

Single is an extension method that itself takes a method that identifies the row you want to find and returns this row as an instance of the entity class (as opposed to a collection of rows in a *Table* collection). You can specify the method parameter as a lambda expression. If the lambda expression does not identify exactly one row, the *Single* method returns an *InvalidOperationException*. The following code example queries the Northwind database for the product with the *ProductID* value of 27. The value returned is an instance of the *Product* class, and the *Console.WriteLine* statement prints the name of the product. As before, the database connection is opened and closed automatically by the *DataContext* object.

```
Product singleProduct = products.Single(p => p.ProductID == 27);
Console.WriteLine("Name: {0}", singleProduct.ProductName);
```

Deferred and Immediate Fetching

An important point to emphasize is that by default, DLINQ retrieves the data from the database only when you request it and not when you define a DLINQ query or create a *Table* collection. This is known as deferred fetching. In the example shown earlier that displays all of the products from the *Products* table, the *productsQuery* collection is populated only when the *foreach* loop runs. This mode of operation matches that of LINQ when querying in-memory objects; you will always see the most up-to-date version of the data, even if the data changes after you have run the statement that creates the *productsQuery* enumerable collection.

When the *foreach* loop starts, DLINQ creates and runs a SQL SELECT statement derived from the DLINQ query to create an ADO.NET *DataReader* object. Each iteration of the *foreach* loop performs the necessary *GetXXX* methods to fetch the data for that row. After the final row has been fetched and processed by the *foreach* loop, DLINQ closes the database connection.

Deferred fetching ensures that only the data an application actually uses is retrieved from the database. However, if you are accessing a database running on a remote instance of SQL Server, fetching data row by row does not make the best use of network bandwidth. In this scenario, you can fetch and cache all the data in a single network request by forcing immediate evaluation of the DLINQ query. You can do this by calling the *ToList* or *ToArray* extension methods, which fetch the data into a list or array when you define the DLINQ query, like this:

```
var productsQuery = from p in products.ToList()
                    select p;
```

In this code example, *productsQuery* is now an enumerable list, populated with information from the *Products* table. When you iterate over the data, DLINQ retrieves it from this list rather than sending fetch requests to the database.

Joining Tables and Creating Relationships

DLINQ supports the *join* query operator for combining and retrieving related data held in multiple tables. For example, the *Products* table in the Northwind database holds the ID of the supplier for each product. If you want to know the name of each supplier, you have to query the *Suppliers* table. The *Suppliers* table contains the *CompanyName* column, which specifies the name of the supplier company, and the *ContactName* column, which contains the name of the person in the supplier company that handles orders from Northwind Traders. You can define an entity class containing the relevant supplier information like this (the *SupplierName* column in the database is mandatory, but the *ContactName* allows null values):

```
[Table(Name = "Suppliers")]
public class Supplier
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int SupplierID { get; set; }

    [Column(CanBeNull = false)]
    public string CompanyName { get; set; }

    [Column]
    public string ContactName { get; set; }
}
```

You can then instantiate *Table*<*Product*> and *Table*<*Supplier*> collections and define a DLINQ query to join these tables together, like this:

```
DataContext db = new DataContext(...);
Table<Product> products = db.GetTable<Product>();
Table<Supplier> suppliers = db.GetTable<Supplier>();
var productsAndSuppliers = from p in products
                           join s in suppliers
                           on p.SupplierID equals s.SupplierID
                           select new { p.ProductName, s.CompanyName, s.ContactName };
```

When you iterate through the *productsAndSuppliers* collection, DLINQ will execute a SQL SELECT statement that joins the *Products* and *Suppliers* tables in the database over the *SupplierID* column in both tables and fetches the data.

However, with DLINQ you can specify the relationships between tables as part of the definition of the entity classes. DLINQ can then fetch the supplier information for each product automatically without requiring that you code a potentially complex and error-prone *join* statement. Returning to the products and suppliers example, these tables have a many-to-one relationship in the Northwind database; each product is supplied by a single supplier, but a single supplier can supply several products. Phrasing this relationship slightly differently, a row in the *Product* table can reference a single row in the *Suppliers* table through the *SupplierID* columns in both tables, but a row in the *Suppliers* table can reference a whole set

of rows in the *Products* table. DLINQ provides the *EntityRef<TEntity>* and *EntitySet<TEntity>* generic types to model this type of relationship. Taking the *Product* entity class first, you can define the "one" side of the relationship with the *Supplier* entity class by using the *EntityRef<Supplier>* type, as shown here in bold type:

```
[Table(Name = "Products")]
public class Product
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int ProductID { get; set; }
    ...
    [Column]
    public int? SupplierID { get; set; }
    ...
    private EntityRef<Supplier> supplier;
    [Association(Storage = "supplier", ThisKey = "SupplierID", OtherKey = "SupplierID")]
    public Supplier Supplier
    {
        get { return this.supplier.Entity; }
        set { this.supplier.Entity = value; }
    }
}
```

The private *supplier* field is a reference to an instance of the *Supplier* entity class. The public *Supplier* property provides access to this reference. The *Association* attribute specifies how DLINQ locates and populates the data for this property. The *Storage* parameter identifies the *private* field used to store the reference to the *Supplier* object. The *ThisKey* parameter indicates which property in the *Product* entity class DLINQ should use to locate the *Supplier* to reference for this product, and the *OtherKey* parameter specifies which property in the *Supplier* table DLINQ should match against the value for the *ThisKey* parameter. In this example, The *Product* and *Supplier* tables are joined across the *SupplierID* property in both entities.

Note The *Storage* parameter is actually optional. If you specify it, DLINQ accesses the corresponding data member directly when populating it rather than going through the *set* accessor. The *set* accessor is required for applications that manually fill or change the entity object referenced by the *EntityRef<TEntity>* property. Although the *Storage* parameter is actually redundant in this example, it is recommended practice to include it.

The *get* accessor in the *Supplier* property returns a reference to the *Supplier* entity by using the *Entity* property of the *EntityRef<Supplier>* type. The *set* accessor populates this property with a reference to a *Supplier* entity.

You can define the "many" side of the relationship in the *Supplier* class with the *EntitySet<Product>* type, like this:

```
[Table(Name = "Suppliers")]
public class Supplier
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int SupplierID { get; set; }
    ...
    private EntitySet<Product> products = null;
    [Association(Storage = "products", OtherKey = "SupplierID", ThisKey = "SupplierID")]
    public EntitySet<Product> Products
    {
        get { return this.products; }
        set { this.products.Assign(value); }
    }
}
```

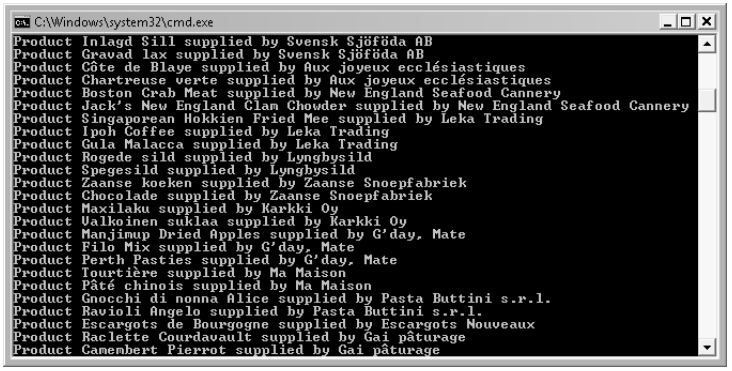
Tip It is conventional to use a singular noun for the name of an entity class and its properties. The exception to this rule is that *EntitySet<TEntity>* properties typically take the plural form because they represent a collection rather than a single entity.

This time, notice that the *Storage* parameter of the *Association* attribute specifies the private *EntitySet<Product>* field. An *EntitySet<TEntity>* object holds a collection of references to entities. The *get* accessor of the public *Products* property returns this collection. The *set* accessor uses the *Assign* method of the *EntitySet<Product>* class to populate this collection.

So, by using the *EntityRef<TEntity>* and *EntitySet<TEntity>* types you can define properties that can model a one-to-many relationship, but how do you actually fill these properties with data? The answer is that DLINQ fills them for you when it fetches the data. The following code creates an instance of the *Table<Product>* class and issues a DLINQ query to fetch the details of all products. This code is similar to the first DLINQ example you saw earlier. The difference is in the *foreach* loop that displays the data.

```
DataContext db = new DataContext(...);
Table<Product> products = db.GetTable<Product>();

var productsAndSuppliers = from p in products
                           select p;

foreach (var product in productsAndSuppliers)
{
    Console.WriteLine("Product {0} supplied by {1}",
        product.ProductName, product.Supplier.CompanyName);
}
```

The *Console.WriteLine* statement reads the value in the *ProductName* property of the product entity as before, but it also accesses the *Supplier* entity and displays the *CompanyName* property from this entity. If you run this code, the output looks like this:



As the code fetches each *Product* entity, DLINQ executes a second, deferred, query to retrieve the details of the supplier for that product so that it can populate the *Supplier* property, based on the relationship specified by the *Association* attribute of this property in the *Product* entity class.

When you have defined the *Product* and *Supplier* entities as having a one-to-many relationship, similar logic applies if you execute a DLINQ query over the *Table<Supplier>* collection, like this:

```
DataContext db = new DataContext(...);
Table<Supplier> suppliers = db.GetTable<Supplier>();
var suppliersAndProducts = from s in suppliers
                           select s;

foreach (var supplier in suppliersAndProducts)
{
    Console.WriteLine("Supplier name: {0}", supplier.CompanyName);
    Console.WriteLine("Products supplied");
    foreach (var product in supplier.Products)
    {
        Console.WriteLine("\t{0}", product.ProductName);
    }
    Console.WriteLine();
}
```

In this case, when the *foreach* loop fetches a supplier, it runs a second query (again deferred) to retrieve all the products for that supplier and populate the *Products* property. This time, however, the property is a collection (an *EntitySet<Product>*), so you can code a nested

foreach statement to iterate through the set, displaying the name of each product. The output of this code looks like this:

Deferred and Immediate Fetching Revisited

Earlier in this chapter, I mentioned that DLINQ defers fetching data until the data is actually requested but that you could apply the *ToList* or *ToArray* extension method to retrieve data immediately. This technique does not apply to data referenced as *EntitySet<TEntity>* or *EntityRef<TEntity>* properties; even if you use *ToList* or *ToArray*, the data will still be fetched only when accessed. If you want to force DLINQ to query and fetch referenced data immediately, you can set the *LoadOptions* property of the *DataContext* object as follows:

```
DataContext db = new DataContext(...);
Table<Supplier> suppliers = db.GetTable<Supplier>();
DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.LoadWith<Supplier>(s => s.Products);
db.LoadOptions = loadOptions;
var suppliersAndProducts = from s in suppliers
                           select s;
```

The *DataLoadOptions* class provides the generic *LoadWith* method. By using this method, you can specify whether an *EntitySet<TEntity>* property in an instance should be loaded when the instance is populated. The parameter to the *LoadWith* method is another method, which you can supply as a lambda expression. The example shown here causes the *Products* property of each *Supplier* entity to be populated as soon as the data for each *Product* entity is fetched rather than being deferred. If you specify the *LoadOptions* property of the *DataContext* object together with the *ToList* or *ToArray* extension method of a *Table* collection, DLINQ will load the entire collection as well as the data for the referenced properties for the entities in that collection into memory as soon as the DLINQ query is evaluated.

Tip If you have several *EntitySet<TEntity>* properties, you can call the *LoadWith* method of the same *LoadOptions* object several times, each time specifying the *EntitySet<TEntity>* to load.

Defining a Custom *DataContext* Class

The *DataContext* class provides functionality for managing databases and database connections, creating entity classes, and executing commands to retrieve and update data in a database. Although you can use the raw *DataContext* class provided with the .NET Framework, it is better practice to use inheritance and define your own specialized version that declares the various *Table<TEntity>* collections as public members. For example, here is a specialized *DataContext* class that exposes the *Products* and *Suppliers Table* collections as public members:

```
public class Northwind : DataContext
{
    public Table<Product> Products;
    public Table<Supplier> Suppliers;

    public Northwind(string connectionInfo) : base(connectionInfo)
    {
    }
}
```

Notice that the *Northwind* class also provides a constructor that takes a connection string as a parameter. You can create a new instance of the *Northwind* class and then define and run DLINQ queries over the *Table* collection classes it exposes like this:

```
Northwind nwindDB = new Northwind(...);

var suppliersQuery = from s in nwindDB.Suppliers
                     select s;

foreach (var supplier in suppliersQuery)
{
    ...
}
```

This practice makes your code easier to maintain, especially if you are retrieving data from multiple databases. Using an ordinary *DataContext* object, you can instantiate any entity class by using the *GetTable* method, regardless of the database to which the *DataContext* object connects. You find out that you have used the wrong *DataContext* object and have connected to the wrong database only at run time, when you try to retrieve data. With a custom *DataContext* class, you reference the *Table* collections through the *DataContext* object. (The *base DataContext* constructor uses a mechanism called *reflection* to examine its members, and it automatically instantiates any members that are *Table* collections—the details of how

reflection works are outside the scope of this book.) It is obvious to which database you need to connect to retrieve data for a specific table; if IntelliSense does not display your table when you define the DLINQ query, you have picked the wrong *DataContext* class, and your code will not compile.

Using DLINQ to Query Order Information

In the following exercise, you will write a version of the console application that you developed in the preceding exercise that prompts the user for a customer ID and displays the details of any orders placed by that customer. You will use DLINQ to retrieve the data. You will then be able to compare DLINQ with the equivalent code written by using ADO.NET.

Define the *Order* entity class

1. Using Visual Studio 2008, create a new project called DLINQOrders by using the Console Application template. Save it in the \Microsoft Press\Visual CSharp Step By Step\Chapter 25 folder under your Documents folder, and then click *OK*.
2. In *Solution Explorer*, change the name of the file Program.cs to DLINQReport.cs. In the *Microsoft Visual Studio* message, click *Yes* to change all references of the *Program* class to *DLINQReport*.
3. On the *Project* menu, click *Add Reference*. In the *Add Reference* dialog box, click the *.NET* tab, select the *System.Data.Linq* assembly, and then click *OK*.

   This assembly holds the DLINQ types and attributes.
4. In the *Code and Text Editor* window, add the following *using* statements to the list at the top of the file:

   ```
   using System.Data.Linq;
   using System.Data.Linq.Mapping;
   using System.Data.SqlClient;
   ```
5. Add the *Order* entity class to the DLINQReport.cs file after the *DLINQReport* class, as follows:

   ```
   [Table(Name = "Orders")]
   public class Order
   {
   }
   ```

   The table is called *Orders* in the Northwind database. Remember that it is common practice to use the singular noun for the name of an entity class because an entity object represents one row from the database.

6. Add the property shown here in bold type to the *Order* class:

```
[Table(Name = "Orders")]
public class Order
{
    [Column(IsPrimaryKey = true, CanBeNull = false)]
    public int OrderID { get; set; }
}
```

The *OrderID* column is the primary key for this table in the Northwind database.

7. Add the following properties shown in bold type to the *Order* class:

```
[Table(Name = "Orders")]
public class Order
{
    ...
    [Column]
    public string CustomerID { get; set; }

    [Column]
    public DateTime? OrderDate { get; set; }

    [Column]
    public DateTime? ShippedDate { get; set; }

    [Column]
    public string ShipName { get; set; }

    [Column]
    public string ShipAddress { get; set; }

    [Column]
    public string ShipCity { get; set; }

    [Column]
    public string ShipCountry { get; set; }
}
```

These properties hold the customer ID, order date, and shipping information for an order. In the database, all of these columns allow null values, so it is important to use the nullable version of the *DateTime* type for the *OrderDate* and *ShippedDate* properties (*string* is a reference type that automatically allows null values). Notice that DLINQ automatically maps the SQL Server *NVarChar* type to the .NET Framework *string* type and the SQL Server *DateTime* type to the .NET Framework *DateTime* type.

8. Add the following *Northwind* class to the DLINQReport.cs file after the *Order* entity class:

```
public class Northwind : DataContext
{
    public Table<Order> Orders;
```

```
    public Northwind(string connectionInfo) : base (connectionInfo)
    {
    }
}
```

The *Northwind* class is a *DataContext* class that exposes a *Table* property based on the *Order* entity class. In the next exercise, you will use this specialized version of the *DataContext* class to access the *Orders* table in the database.

Retrieve order information by using a DLINQ query

1. In the *Main* method of the *DLINQReport* class, add the statement shown here in bold type, which creates a *Northwind* object. Be sure to replace *YourComputer* with the name of your computer:

```
static void Main(string[] args)
{
    Northwind northwindDB = new Northwind("Integrated Security=true;" +
        "Initial Catalog=Northwind;Data Source=YourComputer\\SQLExpress");
}
```

The connection string specified here is exactly the same as in the earlier exercise. The *northwindDB* object uses this string to connect to the Northwind database.

2. After the variable declaration, add a *try/catch* block to the *Main* method:

```
static void Main(string[] args)
{
    ...
    try
    {
        // You will add your code here in a moment
    }
    catch(SqlException e)
    {
        Console.WriteLine("Error accessing the database: {0}", e.Message);
    }
}
```

As when using ordinary ADO.NET code, DLINQ raises a *SqlException* if an error occurs when accessing a SQL Server database.

3. Replace the comment in the *try* block with the following code shown in bold type:

```
try
{
    Console.Write("Please enter a customer ID (5 characters): ");
    string customerId = Console.ReadLine();
}
```

These statements prompt the user for a customer ID and save the user's response in the string variable *customerId*.

4. Type the statement shown here in bold type after the code you just entered:

```
try
{
    ...
    var ordersQuery = from o in northwindDB.Orders
                      where String.Equals(o.CustomerID, customerId)
                      select o;
}
```

This statement defines the DLINQ query that will retrieve the orders for the specified customer.

5. Add the *foreach* statement and *if...else* block shown here in bold type after the code you added in the preceding step:

```
try
{
    ...
    foreach (var order in ordersQuery)
    {
        if (order.ShippedDate == null)
        {
            Console.WriteLine("Order {0} not yet shipped\n\n", order.OrderID);
         }
         else
         {
             // Display the order details
         }
    }
}
```

The *foreach* statement iterates through the orders for the customer. If the value in the *ShippedDate* column in the database is *null*, the corresponding property in the *Order* entity object is also *null*, and then the *if* statement outputs a suitable message.

6. Replace the comment in the *else* part of the *if* statement you added in the preceding step with the code shown here in bold type:

```
if (order.ShippedDate == null)
{
    ...
}
else
{
    Console.WriteLine("Order: {0}\nPlaced: {1}\nShipped: {2}\n" +
                      "To Address: {3}\n{4}\n{5}\n{6}\n\n", order.OrderID,
                       order.OrderDate, order.ShippedDate, order.ShipName,
                       order.ShipAddress, order.ShipCity,
                       order.ShipCountry);
}
```

7. On the *Debug* menu, click *Start Without Debugging* to build and run the application.
8. In the console window displaying the message "Please enter a customer ID (5 characters):", type **VINET**.

   The application should display a list of orders for this customer. When the application has finished, press Enter to return to Visual Studio 2008.
9. Run the application again. This time type **BONAP** when prompted for a customer ID.

   The final order for this customer has not yet shipped and contains a null value for the *ShippedDate* column. Verify that the application detects and handles this null value. When the application has finished, press Enter to return to Visual Studio 2008.

You have now seen the basic elements that DLINQ provides for querying information from a database. DLINQ has many more features that you can employ in your applications, including the ability to modify data and update a database. You will look briefly at some of these aspects of DLINQ in the next chapter.

- If you want to continue to the next chapter

  Keep Visual Studio 2008 running, and turn to Chapter 26.
- If you want to exit Visual Studio 2008 now

  On the *File* menu, click *Exit*. If you see a *Save* dialog box, click *Yes* (if you are using Visual Studio 2008) or *Save* (if you are using Visual C# 2008 Express Edition) and save the project.

Chapter 25 Quick Reference

To	Do this
Connect to a SQL Server database by using ADO.NET	Create a *SqlConnection* object, set its *ConnectionString* property with details specifying the database to use, and call the *Open* method.
Create and execute a database query by using ADO.NET	Create a *SqlCommand* object. Set its *Connection* property to a valid *SqlConnection* object. Set its *CommandText* property to a valid SQL SELECT statement. Call the *ExecuteReader* method to run the query and create a *SqlDataReader* object.
Fetch data by using an ADO.NET *SqlDataReader* object	Ensure that the data is not null by using the *IsDBNull* method. If the data is not null, use the appropriate *GetXXX* method (such as *GetString* or *GetInt32*) to retrieve the data.

Define an entity class	Define a class with public properties for each column. Prefix the class definition with the *Table* attribute, specifying the name of the table in the underlying database. Prefix each property with the *Column* attribute, and specify parameters indicating the name, type, and nullability of the corresponding column in the database.
Create and execute a query by using DLINQ	Create a *DataContext* variable, and specify a connection string for the database. Create a *Table* collection variable based on the entity class corresponding to the table you want to query. Define a DLINQ query that identifies the data to be retrieved from the database and returns an enumerable collection of entities. Iterate through the enumerable collection to retrieve the data for each row and process the results.