

THE EXPERT'S VOICE™

Carsten Thomsen

*Foreword by Manohar Kamath*

*Microsoft MVP and .NETBooks Editor*

([www.dotnetbooks.com](http://www.dotnetbooks.com))

# Database Programming with C#

Understand the options available in using relational and hierarchical databases with  
classic ADO and the new ADO.NET

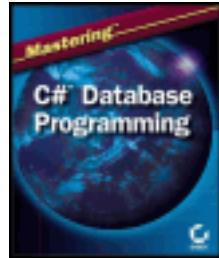


Discover how to connect to SQL Server using any of the three .NET data providers: SQL Server  
.NET Data Provider, OLE DB .NET Data Provider, or ODBC .NET Data Provider



Connect to MySQL, Oracle 9i, Microsoft Access databases, Exchange Server 2000,  
Active Directory (LDAP), and Message Queuing from the example code

**a!**  
apress™



## **Mastering C# Database Programming**

by Jason Price

ISBN:0782141838

Sybex © 2003 (665 pages)

This guide will help you understand database programming as it provides focused coverage of ADO and how it relates to C#, as well as other complex topics including transactions, Windows forms, Web Forms, XML, and Web Services.

## **Table of Contents**

[Mastering C# Database Programming](#)

[Introduction](#)

**Part 1 - Introduction to ADO.NET and Databases**

[Chapter 1](#) - Introduction to Database Programming with ADO.NET

[Chapter 2](#) - Introduction to Databases

[Chapter 3](#) - Introduction to Structured Query Language (SQL)

[Chapter 4](#) - Introduction to Transact-SQL Programming

[Chapter 5](#) - Overview of the ADO.NET Classes

[Chapter 6](#) - Introducing Windows Applications and ADO.NET

**Part 2 - Fundamental Database Programming with ADO.NET**

[Chapter 7](#) - Connecting to a Database

[Chapter 8](#) - Executing Database Commands

[Chapter 9](#) - Using *DataReader* Objects to Read Results

[Chapter 10](#) - Using *Dataset* Objects to Store Data

[Chapter 11](#) - Using *DataSet* Objects to Modify Data

[Chapter 12](#) - Navigating and Modifying Related Data

[Chapter 13](#) - Using *DataView* Objects

**Part 3 - Advanced Database Programming with ADO.NET**

[Chapter 14](#) - Advanced Transaction Control

[Chapter 15](#) - Introducing Web Applications—ASP.NET

[Chapter 16](#) - Using SQL Server's XML Support

[Chapter 17](#) - Web Services

[Index](#)

[List of Figures](#)

[List of Tables](#)

[List of Listings](#)

[List of Sidebars](#)

## Back Cover

C# and ADO.NET facilitate the development of a new generation of database applications, including remote applications that run on the Web. *Mastering C# Database Programming* is the resource you need to thrive in this new world. Assuming no prior experience with database programming, this book teaches you every aspect of the craft, from GUI design to server development to middle-tier implementation. If you're familiar with earlier versions of ADO, you'll master the many new features of ADO.NET all the more quickly. You'll also learn the importance of XML within the new .NET paradigm.

## Coverage Includes

- Accessing a database using C# and ADO.NET
- Using SQL to access a database
- Using Visual Studio .NET to build applications
- Creating and modifying database tables
- Understanding ADO.NET classes
- Designing, building, and deploying Web applications that access a database
- Designing, building, and deploying effective Web services
- Using SQL Server's built-in XML capabilities
- Working with a database in a disconnected manner
- Using advanced transaction controls
- Using Transact-SQL to create stored procedures and functions in a SQL Server database

## About the Author

Jason Price is an independent consultant and writer, and is both a Microsoft Certified Professional and an Oracle Certified Professional. Jason has more than 10 years of experience in the software industry, and he has extensive experience with C#, .NET, and Java. He is the author of *Mastering Visual C# .NET*, *Oracle9i JDBC Programming*, and *Java Programming with Oracle SQLJ*.

# Mastering C# Database Programming

Jason Price



San Francisco London

**Associate Publisher:** Joel Fugazzotto

**Acquisitions Editor:** Denise Santoro-Lincoln

**Developmental Editor:** Tom Cirtin

**Production Editor:** Erica Yee

**Technical Editor:** Acey Bunch

**Copyeditor:** Laura Ryan

**Compositor:** Jill Niles

**Graphic Illustrator:** Jeff Wilson, Happenstance Type-O-Rama

**Proofreaders:** Emily Hsuan, Laurie O'Connell, Nancy Riddough, Monique van den Berg

**Indexer:** Ted Laux

**Book Designer:** Maureen Forys, Happenstance Type-O-Rama

**Cover Designer:** Design Site

**Cover Illustrator:** Tania Kac, Design Site

Copyright © 2003 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501.

World rights reserved. The author created reusable code in this publication expressly for reuse by readers. Sybex grants readers limited permission to reuse the code found in this publication or its accompanying CD-ROM so long as the author is attributed in any application containing the reusable code and the code itself is never distributed, posted on line by electronic transmission, sold, or commercially exploited as a stand-alone product. Aside from this specific exception concerning reusable code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 2002116881

ISBN: 0-7821-4183-8

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the United States and/or other countries.

Mastering and the Mastering logo are trademarks of SYBEX Inc.

Screen reproductions produced with FullShot 99. FullShot 99 © 1991-1999 Inbit Incorporated. All rights reserved. FullShot is a trademark of Inbit Incorporated.

Internet screen shot(s) using Microsoft Internet Explorer 6 reprinted by permission from Microsoft Corporation.

**TRADEMARKS:** SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Photographs and illustrations used in this book have been downloaded from publicly accessible file archives and are used in this book for news reportage purposes only to demonstrate the variety of graphics resources available via electronic access. Text and images available over the Internet may be subject to copyright and other rights owned by third parties. Online availability of text and images does not imply that they may be reused without the permission of rights holders, although the Copyright Act does permit certain unauthorized reuse as fair use under 17 U.S.C. Section 107.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

### **Software License Agreement: Terms and Conditions**

The media and/or any online materials accompanying this book that are available now or in the future contain programs and/or text files (the "Software") to be used in connection with the book. SYBEX hereby grants to you a license to use the Software, subject to the terms that follow. Your purchase, acceptance, or use of the Software will constitute your acceptance of such terms.

The Software compilation is the property of SYBEX unless otherwise indicated and is protected by copyright to SYBEX or other copyright owner(s) as indicated in the media files (the "Owner(s)"). You are hereby granted a single-user license to use the Software for your personal, noncommercial use only. You may not reproduce, sell, distribute, publish, circulate, or commercially exploit the Software, or any portion thereof, without the written consent of SYBEX and the specific copyright owner(s) of any component software included on this media.

In the event that the Software or components include specific license requirements or end-user agreements, statements of condition, disclaimers, limitations or warranties ("End-User License"), those End-User Licenses supersede the terms and conditions herein as to that particular Software component. Your purchase, acceptance, or use of the Software will constitute your acceptance of such End-User Licenses.

By purchase, use, or acceptance of the Software, you further agree to comply with all export laws and regulations of the United States as such laws and regulations may exist from time to time.

### **REUSABLE CODE IN THIS BOOK**

The author(s) created reusable code in this publication expressly for reuse by readers. Sybex grants readers limited permission to reuse the code found in this publication, its accompanying CD-ROM or available for download from our website so long as the author(s) are attributed in any application containing the reusable code and the code itself is never distributed, posted online by electronic transmission, sold, or commercially exploited as a stand-alone product.

### **SOFTWARE SUPPORT**

Components of the supplemental Software and any offers associated with them may be supported by the specific Owner(s) of that material, but they are not supported by SYBEX. Information regarding any available support may be obtained from the Owner(s) using the information provided in the appropriate read.me files or listed elsewhere on the media.

Should the manufacturer(s) or other Owner(s) cease to offer support or decline to honor any offer, SYBEX bears no responsibility. This notice concerning support for the Software is provided for your information only. SYBEX is not the agent or principal of the Owner(s), and SYBEX is in no way responsible for providing any support for the Software, nor is it liable or responsible for any support provided, or not provided, by the Owner(s).

### **WARRANTY**

SYBEX warrants the enclosed media to be free of physical defects for a period of ninety (90) days after purchase. The Software is not available from SYBEX in any other form or media than that enclosed herein or posted to [www.sybex.com](http://www.sybex.com). If you discover a defect in the media during this warranty period, you may obtain a replacement of identical format at no charge by sending the defective media, postage

prepaid, with proof of purchase to:

SYBEX Inc.  
Product Support Department  
1151 Marina Village Parkway  
Alameda, CA 94501  
Web: <http://www.sybex.com>

After the 90-day period, you can obtain replacement media of identical format by sending us the defective disk, proof of purchase, and a check or money order for \$10, payable to SYBEX.

#### DISCLAIMER

SYBEX makes no warranty or representation, either expressed or implied, with respect to the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will SYBEX, its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, SYBEX further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by SYBEX reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

#### SHAREWARE DISTRIBUTION

This Software may contain various programs that are distributed as shareware. Copyright laws apply to both shareware and ordinary commercial software, and the copyright Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

#### COPY PROTECTION

The Software in whole or in part may or may not be copy-protected or encrypted. However, in all cases, reselling or redistributing these files without authorization is expressly forbidden except as specifically provided for by the Owner(s) therein.

*This book is dedicated to my late mother Patricia Anne Price and to my family. You're still in my heart, even though you are far away.*

#### Acknowledgments

Many thanks to all the great, hard-working people at Sybex-including Tom Cirtin, Denise Santoro Lincoln, Laura Ryan, and Erica Yee.

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## Introduction

Welcome to *Mastering C# .NET Database Programming!* As you might already know, .NET is poised to become *the* hot platform for the next wave of technology deployment. .NET's strength is that it is built from the ground up to be used in a distributed environment-in other words, an environment that consists of computers and devices connected via a network.

**Note** The focus of this book is how you write C# programs that interact with a database. C# uses ADO.NET to interact with a database; ADO.NET is the successor to ADO. In this book, you'll learn the details of interacting with a SQL Server database. SQL Server is Microsoft's premier database software.

Microsoft has pledged its commitment and resources to making .NET a pervasive component of life in our technological society-ignore .NET at your own peril. The bottom line is you need to learn .NET if you want to remain competitive in today's-and tomorrow's-marketplace.

In a nutshell, .NET is a completely new framework for writing many types of applications. The applications you can write using .NET include Windows applications and Web-based applications. You can use .NET to develop systems composed of interconnected services that communicate with each other over the Internet.

In addition, you can use .NET to create applications that run on devices such as handheld computers and cellular phones. Although other languages allow you to develop such applications, .NET was designed with the interconnected network in mind.

The .NET Framework consists of three primary components:

- **Development Languages and Tools** The development languages that enable you to write .NET programs include C#, Visual Basic .NET (VB .NET), and Managed C++. Microsoft also has a Rapid Application Development (RAD) tool called Visual Studio .NET (VS .NET) that allows you to develop programs in an integrated development environment (IDE). You'll use C# and VS .NET in this book.
- **Common Language Runtime (CLR)** CLR manages your running code and provides services such as memory management, thread management (which allows you to perform multiple tasks in parallel), and remoting (which allows objects in one application to communicate with objects in another application). The CLR also enforces strict safety and accuracy of your executable code to ensure that no tampering occurs.
- **Framework Base Class Library** The Framework Base Class Library is an extensive collection of code written by Microsoft that you can use in your own programs. For example, among many other functions, the Framework Base Class Library contains code that allows you to develop Windows applications, access directories and files on disk, interact with databases, and send and receive data across a network.

## Who Should Read This Book?

This book was written for programmers who already know C#. It contains everything you need to know to master database programming with C#. No prior experience of databases is assumed, but if you already have some knowledge of database software such as SQL Server or Oracle, you'll be off to a running start.

**Note** If you don't know C#, I recommend the book *Mastering Visual C# .NET* from Sybex (2002).

## How to Use This Book

This book is divided into three parts. In [Part 1](#), "Introduction to ADO.NET and Databases," you'll learn everything you need to know about databases. You'll also be introduced to ADO.NET, which enables your C# programs to interact with a database. In [Part 2](#), "Fundamental Database Programming with ADO.NET," you'll learn the C# programming with ADO.NET from the ground up. In [Part 3](#), "Advanced Database Programming with ADO.NET," you'll go beyond the basics to learn programming techniques needed by professional database developers.

The following sections describe the chapters in detail.

### Part 1: "Introduction to ADO.NET and Databases"

In [Chapter 1](#), "Introduction to Database Programming with ADO.NET," you'll see how to use ADO.NET in a C# program to interact with a database. You also learn about Microsoft's RAD tool, Visual Studio .NET. Finally, you'll see how to use the extensive documentation from Microsoft that comes with .NET and SQL Server.

In [Chapter 2](#), "Introduction to Databases," you'll learn the details of what databases are and how they are used to store information. You'll see the use of a SQL Server database named Northwind. This database contains the information for the fictitious Northwind Company, which sells food products. This database is one of the example databases that is typically installed with SQL Server.

In [Chapter 3](#), "Introduction to the Structured Query Language," you'll learn how to use the Structured Query Language (SQL) to access a database. You'll see how you use SQL to interact with the Northwind database, and how to retrieve and modify information stored in that database.

In [Chapter 4](#), "Introduction to Transact-SQL Programming," you'll be introduced to programming with Microsoft's Transact-SQL. Transact-SQL enables you to write programs that contain SQL statements, along with standard programming constructs such as variables, conditional logic, loops, procedures, and functions.

In [Chapter 5](#), "Overview of the ADO.NET Classes," you'll get an overview of the ADO.NET classes. You'll also see a C# program that connects to a database, stores the rows locally, disconnects from the database, and then reads the contents of those local rows while disconnected from the database. This ability to store a local copy of rows retrieved from the database is one of the main strengths of ADO.NET.

In [Chapter 6](#), "Introducing Windows Applications and ADO.NET," you'll be introduced to Windows applications. A Windows application takes advantage of displaying and using the mouse and keyboard for input. Windows provides graphical items such as menus, text boxes, and radio buttons so you can build a visual interface that will be easy to use. You'll see how to build Windows applications that interact with the Northwind database.

### Part 2: "Fundamental Database Programming with ADO.NET"

In [Chapter 7](#), "Connecting to a Database," you'll learn the details on connecting to a database. There are three Connection classes: `SqlConnection`, `OleDbConnection`, and `OdbcConnection`. You use an object of the `SqlConnection` class to connect to a SQL Server database. You use an object of the `OleDbConnection` class to connect to any database that supports OLE DB (Object Linking and Embedding for Databases), such as Oracle or Access. You use an object of the `OdbcConnection` class to connect to any database that supports ODBC (Open Database Connectivity). Ultimately, all communication with a database is done through a `Connection` object.

In [Chapter 8](#), "Executing Database Commands," you'll learn the details on executing database commands. You use a `Command` object to execute a SQL SELECT, INSERT, UPDATE, or DELETE statement. You can also use a `Command` object to call a stored procedure, or retrieve all the rows and columns from a specific table.

In [Chapter 9](#), "Using DataReader Objects to Read Results," you'll see how to use a `DataReader` object to read results returned from the database. You use a `DataReader` object to read rows retrieved from the database using a `Command` object.

In [Chapter 10](#), "Using DataSet Objects to Store Data," you'll learn how to use a `DataSet` object to store results returned from the database. `DataSet` objects allow you to store a copy of the tables and rows from the database, and you can work with that local copy.

while disconnected from the database.

In [Chapter 11](#), "Using `DataSet` Objects to Modify Data," you'll examine how to modify the rows in a `DataSet` and then push those changes to the database via a `DataAdapter`.

In [Chapter 12](#), "Navigating and Modifying Related Data," you'll delve into the details of how you navigate related data in tables, make changes in that data in memory, and finally push those changes to the database.

In [Chapter 13](#), "Using `DataView` Objects," you'll see how to use `DataView` objects to filter and sort rows. The advantage of a `DataView` is that you can bind it to a visual component in a Windows or ASP.NET application.

## Part 3: "Advanced Database Programming with ADO.NET"

In [Chapter 14](#), "Advanced Transaction Control," you'll delve into advanced transaction control using SQL Server and ADO.NET.

In [Chapter 15](#), "Introducing Web Applications: ASP.NET," you'll learn the basics of ASP.NET, and you'll see how to use Visual Studio .NET to create ASP.NET applications.

In [Chapter 16](#), "Using SQL Server's XML Support," you'll learn about SQL Server's extensive support for XML. You'll also see how to store XML in a C# program using  `XmlDocument` and  `XmlDataDocument` objects.

In [Chapter 17](#), "Web Services," you'll learn how to build a simple web service, which is a software component that may be used across the Web. For example, you could build a eb service that allows one company to send another company an order across the Web using XML.

## Downloading the Example Programs

Throughout this book, you'll see many example programs that illustrate the concepts described in the text. These are marked with a listing number and title, such as the one shown here:

### **LISTING 1.1: FIRSTEXAMPLE.CS**

The filenames will correspond to the listing name: `FirstExample.cs` is the filename for [Listing 1.1](#). You can download a Zip file containing the programs from the Sybex Web site at [www.sybex.com](http://www.sybex.com). You can use a program such as WinZip to extract the contents of the Zip file.

When you unzip this file, one directory for each chapter will be created. Each directory will contain the following sub-directories as required:

- **programs** Contains the C# programs.
- **sql** Contains SQL scripts.
- **VS. .NET projects** Contains the Visual Studio .NET projects.
- **xml** Contains the XML files.

**Note** Not all chapters reference programs, sql scripts, etc., and therefore may not contain all the previous sub-directories.

Team LiB

◀ PREVIOUS ▶ NEXT ▶

## Part 1: Introduction to ADO.NET and Databases

### Chapter List

[Chapter 1:](#) Introduction to Database Programming with ADO.NET

[Chapter 2:](#) Introduction to Databases

[Chapter 3:](#) Introduction to Structured Query Language (SQL)

[Chapter 4:](#) Introduction to Transact-SQL Programming

[Chapter 5:](#) Overview of the ADO.NET Classes

[Chapter 6:](#) Introducing Windows Applications and ADO.NET

Team LiB

◀ PREVIOUS ▶ NEXT ▶

# Chapter 1: Introduction to Database Programming with ADO.NET

## Overview

A **Database** is an organized collection of information that is divided into **tables**. Each table is further divided into **rows** and **columns**; these columns store the actual information. You access a database using **Structured Query Language** (SQL), which is a standard language supported by most database software including SQL Server, Access, and Oracle.

In this chapter, you'll see a C# program that connects to a SQL Server database, retrieves and displays the contents stored in the columns of a row from a table, and then disconnects from the database. You'll also see programs that connect to Access and Oracle databases.

You'll also learn about Microsoft's rapid application development (RAD) tool, Visual Studio .NET (VS .NET). VS .NET enables you to develop, run, and debug programs in an integrated development environment. This environment uses all the great features of Windows, such as the mouse and intuitive menus, and increases your productivity as a programmer.

In the final sections of this chapter, you'll see how to use the extensive Microsoft documentation that comes with the .NET Software Development Kit (SDK) and VS .NET. You'll find this documentation invaluable as you become an expert with ADO.NET and C#. You'll also learn how to use the SQL Server documentation.

Featured in this chapter:

- Obtaining the required software
- Developing your first ADO.NET program
- Connecting to Access and Oracle databases
- [Introducing Visual Studio .NET](#)
- [Using the .NET documentation](#)
- Using the SQL Server documentation

## Obtaining the Required Software

Before you can develop C# programs, you'll need to install either the .NET Software Development Kit (SDK) or VS .NET. You can download the .NET SDK at <http://msdn.microsoft.com/downloads> (search for the Microsoft .NET Framework Software Development Kit). You can purchase a trial or full copy of VS .NET from Microsoft at <http://msdn.microsoft.com/vstudio>.

To install the .NET SDK, run the executable file you downloaded and follow the instructions on the screen to install it on your computer. To install VS .NET, run the `setup.exe` file on the disk and follow the instructions on the screen.

You'll also need a copy of the SQL Server database software. At time of writing, you can download a trial version of SQL Server from Microsoft at <http://www.microsoft.com/sql>. You can also purchase a trial or full copy of SQL Server from Microsoft's Web site.

This book uses the Developer Edition of the SQL Server 2000 software and uses a database named Northwind. This database contains the information for the fictitious Northwind Company, which sells food products to customers. Northwind is one of the example databases that you can install with SQL Server. Customer information in the Northwind database is stored in a table named `Customers`; you'll see the use of this table in the example program later in this chapter.

If you don't want to download or purchase a trial version of SQL Server, the .NET SDK (and VS .NET) comes with a stand-alone desktop database server known as the Microsoft SQL Server 2000 Desktop Engine (MSDE 2000). MSDE 2000 has a version of the Northwind database that you can use instead of the SQL Server Northwind database—although you won't get all of the graphical administration tools that come with SQL Server. If you're using the .NET SDK and want to install MSDE 2000, select Start > Microsoft .NET Framework SDK > Samples and QuickStart Tutorials. If you're using VS .NET and want to install MSDE 2000, run the `setup.exe` program that you use to install VS .NET and select MSDE 2000 as a new feature to install.

**Note** You can learn more about MSDE 2000 at <http://www.microsoft.com/sql/techinfo/development/2000/msde2000.asp>.

## Developing Your First ADO.NET Program

In this section you'll plunge into ADO.NET programming and see a C# program that performs the following tasks:

1. Connects to the SQL Server Northwind database
2. Retrieves a row from the *Customers* table
3. Displays the columns from the row
4. Closes the database connection

You'll be introduced to many concepts in this section that are fully explored in later chapters. Don't be too concerned about all the details of the concepts at this stage; you'll learn those details in the later chapters.

[Listing 1.1](#) shows the example program, which is contained in the file `FirstExample.cs`.

### Listing 1.1: FIRSTEXAMPLE.CS

```
/*
FirstExample.cs illustrates how to:
1. Connect to the SQL Server Northwind database.
2. Retrieve a row from the Customers table using
   a SQL SELECT statement.
3. Display the columns from the row.
4. Close the database connection.
*/
using System;
using System.Data.SqlClient;

class FirstExample
{
    public static void Main()
    {
        try
        {
            // step 1: create a SqlConnection object to connect to the
            // SQL Server Northwind database
            SqlConnection mySqlConnection =
                new SqlConnection(
                    "server=localhost;database=Northwind;uid=sa;pwd=sa"
                );

            // step 2: create a SqlCommand object
            SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

            // step 3: set the CommandText property of the SqlCommand object to
            // a SQL SELECT statement that retrieves a row from the Customers table
            mySqlCommand.CommandText =
                "SELECT CustomerID, CompanyName, ContactName, Address "+
                "FROM Customers "+
                "WHERE CustomerID = 'ALFKI'";

            // step 4: open the database connection using the
            // Open() method of the SqlConnection object
            mySqlConnection.Open();
            // step 5: create a SqlDataReader object and call the ExecuteReader()
            // method of the SqlCommand object to run the SELECT statement
            SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();
        }
    }
}
```

```

// step 6: read the row from the SqlDataReader object using
// the Read() method
mySqlDataReader.Read();

// step 7: display the column values
Console.WriteLine("mySqlDataReader[\"CustomerID\"] = " +
    mySqlDataReader["CustomerID"]);
Console.WriteLine("mySqlDataReader[\"CompanyName\"] = " +
    mySqlDataReader["CompanyName"]);
Console.WriteLine("mySqlDataReader[\"ContactName\"] = " +
    mySqlDataReader["ContactName"]);
Console.WriteLine("mySqlDataReader[\"Address\"] = " +
    mySqlDataReader["Address"]);

// step 8: close the SqlDataReader object using the Close() method
mySqlDataReader.Close();

// step 9: close the SqlConnection object using the Close() method
mySqlConnection.Close();
}

catch (SqlException e)
{
    Console.WriteLine("A SqlException was thrown");
    Console.WriteLine("Number = " + e.Number);
    Console.WriteLine("Message = " + e.Message);
    Console.WriteLine("StackTrace:\n" + e.StackTrace);
}
}
}
}

```

---

**Note** You can download all the source files for the programs featured in this book from the Sybex Web site at [www.sybex.com](http://www.sybex.com). You'll find instructions on downloading these files in the introduction of this book. Once you've downloaded the files, you can follow along with the examples without having to type in the program listings.

Let's go through the lines in `FirstExample.cs`. The first set of lines is a comment that indicates what the program does:

```

/*
FirstExample.cs illustrates how to:
1. Connect to the SQL Server Northwind database.
2. Retrieve a row from the Customers table using
   a SQL SELECT statement.
3. Display the columns from the row.
4. Close the database connection.
*/

```

The next two lines indicate the namespaces being referenced in the program with the `using` statement:

```

using System;
using System.Data.SqlClient;

```

The `System` namespace is the root namespace and is referenced so that we can simply use `Console.WriteLine()` calls in the program, rather than the fully qualified `System.Console.WriteLine()` call. The `System.Data.SqlClient` namespace contains the ADO.NET classes for use with SQL Server, including the `SqlConnection`, `SqlCommand`, and `SqlDataReader` classes that are used later in the program. You'll be introduced to these classes shortly, and you'll learn the full details of the ADO.NET classes as you progress through this book.

You handle exceptions that might be thrown in your code by placing the code within a `try/catch` block. You'll notice that the nine steps

are placed within a `try/catch` block in the `Main()` method, with the `catch` block handling a `SqlException` object that might be thrown by the code in the `try` block. You'll learn more about this later in the section "[Handling Exceptions](#)" after I've discussed the nine steps in the following sections.

## Step 1: Create a `SqlConnection` Object to Connect to the Database

You use an object of the `SqlConnection` class to connect to a SQL Server database. Step 1 in the `Main()` method creates a `SqlConnection` object named `mySqlConnection` to connect to the SQL Server Northwind database:

```
SqlConnection mySqlConnection =
    new SqlConnection(
        "server=localhost;database=Northwind;uid=sa;pwd=sa"
    );
```

The string passed to the `SqlConnection` constructor is known as the *connection string* and contains the following elements:

- **server** Specifies the name of the computer on which SQL Server is running-`localhost` in this example; `localhost` is a common name that refers to the computer on which your program runs. If your database is running on a computer other than the one your program is running on, then you'll need to replace `localhost` with the name of that computer.
- **database** Specifies the name of the database-`Northwind` in this example.
- **uid** Specifies the name of the database user account-`sa` in this example; `sa` is a common database user account used by the database administrator (DBA). You can use any database user account as long as it has access to the Northwind database.
- **pwd** Specifies the password for the user. The password for the `sa` user in my database is also `sa`. You'll need to change `pwd` to the password for your `sa` account, or whichever account you specified in `uid`.

You'll need to change the settings of some or all of the previous elements in your connection string. You might need to speak with your DBA to get the various elements that make up your connection string. Once you have the correct values, you should make the changes to the connection string in your copy of `FirstExample.cs`.

**Note** A database administrator (DBA) is responsible for performing tasks such as installing the database software, backing up the databases, and so on.

## Step 2: Create a `SqlCommand` Object

Step 2 creates a `SqlCommand` object named `mySqlCommand` that is used later to send a `SELECT` statement to the database for execution.

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
```

## Step 3: Set the `CommandText` Property of the `SqlCommand` Object

You use SQL to work with the information stored in a database. SQL is an industry standard language supported by SQL Server, Access, and Oracle. You use the SQL `SELECT` statement for retrieving information from a database. You'll learn the basics of SQL in [Chapter 3](#), "Introduction to the Structured Query Language."

Step 3 sets the `CommandText` property of `mySqlCommand` created in the previous step to a `SELECT` statement. This statement will retrieve the `CustomerID`, `CompanyName`, `ContactName`, and `Address` columns from the row in the `Customers` table whose `CustomerID` is `ALFKI`:

```
mySqlCommand.CommandText =
    "SELECT CustomerID, CompanyName, ContactName, Address " +
    "FROM Customers " +
    "WHERE CustomerID = 'ALFKI'";
```

## Step 4: Open the `SqlConnection` Object

Step 4 opens the database connection using the `Open()` method of the `SqlConnection` object created in [step 1](#):

```
mySqlConnection.Open();
```

Once the connection to the database is open, you can send commands to the database for execution.

## Step 5: Run the `SELECT` Statement

You run the `SELECT` statement previously set in `mySqlCommand` by calling the `ExecuteReader()` method. This method returns a `SqlDataReader` object that you then use to read the row data returned by the `SELECT` statement.

Step 5 creates a `SqlDataReader` object and calls the `ExecuteReader()` method of `mySqlCommand` object to run the `SELECT` statement:

```
SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();
```

## Step 6: Read the Row Using the `SqlDataReader` Object

Step 6 reads the row in `mySqlDataReader` using the `Read()` method:

```
mySqlDataReader.Read();
```

## Step 7: Display the Column Values from the `SqlDataReader` Object

You can read the value for a column from `mySqlDataReader` by passing the name of the column in square brackets. For example, `mySqlDataReader["CustomerID"]` returns the value of the `CustomerID` column.

Step 7 displays the column values for the `CustomerID`, `CompanyName`, `ContactName`, and `Address` column values:

```
Console.WriteLine("mySqlDataReader[\"CustomerID\"] = " +
    mySqlDataReader["CustomerID"]);
Console.WriteLine("mySqlDataReader[\"CompanyName\"] = " +
    mySqlDataReader["CompanyName"]);
Console.WriteLine("mySqlDataReader[\"ContactName\"] = " +
    mySqlDataReader["ContactName"]);
Console.WriteLine("mySqlDataReader[\"Address\"] = " +
    mySqlDataReader["Address"]);
```

## Step 8: Close the `SqlDataReader` Object

When you're finished reading rows from a `SqlDataReader` object, close it using the `Close()` method. Step 8 calls the `Close()` method for `mySqlDataReader`:

```
mySqlDataReader.Close();
```

## Step 9: Close the `SqlConnection` Object

When you're finished accessing the database, close your `SqlConnection` object using the `Close()` method. Step 9 calls the `Close()` method for `mySqlConnection`:

```
mySqlConnection.Close();
```

## Handling Exceptions

You handle exceptions that might be thrown in your code by placing the code within a `try/catch` block. You'll notice that the nine steps are placed within a `try/catch` block, with the `catch` block handling a `SqlException` object that might be thrown by the code in the

`try` block. The `SqlException` class is specifically for use with code that accesses a SQL Server database.

The following example shows how to structure a `try/catch` block:

```
try
{
    /* code that might throw a SqlException */
}
catch (SqlException e)
{
    Console.WriteLine("A SqlException was thrown");
    Console.WriteLine("Number = " + e.Number);
    Console.WriteLine("Message = " + e.Message);
    Console.WriteLine("StackTrace:\n" + e.StackTrace);
}
```

The properties displayed in the `catch` block are as follows:

- **Number** The error number
- **Message** A string containing a description of the error
- **StackTrace** A string containing the name of the class and the method from which the exception was thrown

The two most common examples of when a `SqlException` object is thrown are as follows:

- Your `SqlConnection` object is unable to connect to the database. If this happens, you should check the connection string that specifies how to connect to your database.
- Your `SELECT` statement contains a mistake in the spelling of a table or column.

The following example output shows what happens when the `SqlConnection` object in `FirstExample.cs` is unable to connect to the database because the database is currently down:

```
A SqlException was thrown
Number = -2
Message = Timeout expired. Possible reasons: the timeout period elapsed prior
          to completion of the operation, the server is not responding,
          or the maximum pool size was exceeded.
Please see the documentation for further details.
StackTrace:
  at System.Data.SqlClient.SqlConnection.Open()
  at FirstExample.Main()
```

You can use the output from your `catch` block to determine the problem. If the database is down, contact your DBA.

**Note** For brevity, the only program to use a `try/catch` block in this book is `FirstExample.cs`. You should use `try/catch` blocks in your own programs to catch exceptions. For more details on handling exceptions, I recommend the book *Mastering Visual C# .NET* from Sybex (2002).

In the [next section](#) you'll see how to compile `FirstExample.cs` and run it.

## Compiling and Running `FirstExample.cs`

You can compile the `FirstExample.cs` program using either the command-line tool that comes with the .NET SDK or VS .NET. In this section, you'll see how to use the command-line version of the compiler for `FirstExample.cs` program. Later in this chapter, in the section "[Introducing Visual Studio .NET](#)," you'll see how to use VS .NET to compile and run a program.

You run the command-line version of the compiler by entering `csc` in the Command Prompt tool, followed by the name of your program

source file. For example, to compile `FirstExample.cs`, you would enter the following command in the Command Prompt tool:

```
csc FirstExample.cs
```

If you want to follow along with the examples, start the Command Prompt tool by selecting Start > Programs > Accessories > Command Prompt.

**Note** If you're using Windows XP rather than Windows 2000, start the Command Prompt tool by selecting Start > All Programs > Accessories > Command Prompt.

Next, you need to change directories to where you copied the `FirstExample.cs` file. To do this, you first enter the partition on your hard disk where you saved the file. For example, let's say you saved the file in the `ADO.NET\book\ch01\programs` directory of the C partition of your hard disk. To access the C partition, you enter the following line into the Command Prompt tool and then you press the Enter key:

```
C:
```

Next, to move to the `ADO.NET\book\ch01\programs` directory, you enter `cd` followed by `ADO.NET\book\ch01\programs`:

```
cd ADO.NET\book\ch01\programs
```

To compile `FirstExample.cs` using `csc`, you enter the following command:

```
csc FirstExample.cs
```

Notice that the name of the program source file follows `csc`; in this case, it's `FirstExample.cs`.

If you get an error when running `csc`, you'll need to add the directory where you installed the SDK to your Path environment variable. The Path environment variable specifies a list of directories that contain executable programs. Whenever you run a program from the command prompt, the directories in the Path variable are searched for the program you want to run. Your current directory is also searched. To set your Path environment variable, do the following:

1. Select Start > Settings > Control Panel. Then double-click System and select the Advanced tab.
2. Click the Environment Variables button and double-click `Path` from the system variables area at the bottom.
3. Add the directory where you installed the SDK to your `Path` environment variable.
4. Click OK to save your change, and then click OK again on the next dialog.
5. Restart Command Prompt so that your change is picked up. You should then be able to run `csc` successfully.

The compiler takes the `FirstExample.cs` file and compiles it into an executable file named `FirstExample.exe`. The `.exe` file contains instructions that a computer can run, and the `.exe` file extension indicates the file is an executable file.

You run an executable file using the Command Prompt tool by entering the name of that executable file. For example, to run the `FirstExample.exe` file, you enter the following line in the Command Prompt tool and then you press the Enter key:

```
FirstExample
```

When you run the program, you should see the following text displayed in your Command Prompt window:

```
mySqlDataReader[ "CustomerID" ] = ALFKI
mySqlDataReader[ "CompanyName" ] = Al�eds Futterkiste
mySqlDataReader[ "ContactName" ] = Maria Anders
mySqlDataReader[ "Address" ] = Obere Str. 57
```

If you encounter an exception-such as your program can't connect to the database-you should check the connection string set in step 1 of `FirstExample.cs`, and speak with your DBA if necessary.

## Connecting to Access and Oracle Databases

In this section you'll see examples of connecting to both an Access and an Oracle database. To interact with either of these databases in your program, you use classes from the `System.Data.OleDb` namespace. This namespace contains classes for use with databases that support object linking and embedding for databases (OLE DB) such as Access or Oracle. You'll learn more about the `System.Data.OleDb` namespace in [Chapter 5](#), "Overview of the ADO.NET Classes."

### Connecting to an Access Database

You connect to an Access database using an `OleDbConnection` object-rather than a `SqlConnection` object-with a connection string of the following format:

```
provider=Microsoft.Jet.OLEDB.4.0;data source=databaseFile
```

where `databaseFile` is the directory and filename of your Access database. Notice that you specify the `provider` in the connection string, which is set to `Microsoft.Jet.OLEDB.4.0`.

The following example creates a string named `connectionString` with the appropriate format to connect to the Access Northwind database stored in the `Northwind.mdb` file:

```
string connectionString =
    "provider=Microsoft.Jet.OLEDB.4.0;" +
    "data source=F:\\Program Files\\Microsoft
    Office\\Office\\Samples\\Northwind.mdb";
```

**Note** Notice the use of two backslash characters in the `data source` part of the connection string. The first backslash is used to specify that the second backslash is to be treated literally; therefore \\ is treated as \ in the connection string. You'll need to locate the `Northwind.mdb` file on your hard disk and set your connection string appropriately.

Assuming the `System.Data.OleDb` namespace has been imported, the following example creates an `OleDbConnection` object, passing `connectionString` (set in the previous line of code) to the constructor:

```
OleDbConnection myOleDbConnection =
    new OleDbConnection(connectionString);
```

[Listing 1.2](#) illustrates how to connect to the Northwind Access database using an `OleDbConnection` object and retrieve a row from the `Customers` table. Notice that you use an `OleDbCommand` and `OleDbDataReader` object to run a SQL statement and read the returned results from an Access database.

### Listing 1.2: OLEDBCONNECTIONACCESS.CS

---

```
/*
 OleDbConnectionAccess.cs illustrates how to use an
 OleDbConnection object to connect to an Access database
 */

using System;
using System.Data;
using System.Data.OleDb;

class OleDbConnectionAccess
{
    public static void Main()
    {
        // formulate a string containing the details of the
        // database connection
        string connectionString =
            "provider=Microsoft.Jet.OLEDB.4.0;" +
```

```

    "data source=F:\\Program Files\\Microsoft
Office\\Office\\Samples\\Northwind.mdb";

    // create an OleDbConnection object to connect to the
    // database, passing the connection string to the constructor
    OleDbConnection myOleDbConnection =
        new OleDbConnection(connectionString);

    // create an OleDbCommand object
    OleDbCommand myOleDbCommand = myOleDbConnection.CreateCommand();

    // set the CommandText property of the OleDbCommand object to
    // a SQL SELECT statement that retrieves a row from the Customers table
    myOleDbCommand.CommandText =
        "SELECT CustomerID, CompanyName, ContactName, Address " +
        "FROM Customers " +
        "WHERE CustomerID = 'ALFKI'";
    // open the database connection using the
    // Open() method of the OleDbConnection object
    myOleDbConnection.Open();

    // create an OleDbDataReader object and call the ExecuteReader()
    // method of the OleDbCommand object to run the SELECT statement
    OleDbDataReader myOleDbDataReader = myOleDbCommand.ExecuteReader();

    // read the row from the OleDbDataReader object using
    // the Read() method
    myOleDbDataReader.Read();

    // display the column values
    Console.WriteLine("myOleDbDataReader[\"CustomerID\"] = " +
        myOleDbDataReader["CustomerID"]);
    Console.WriteLine("myOleDbDataReader[\"CompanyName\"] = " +
        myOleDbDataReader["CompanyName"]);
    Console.WriteLine("myOleDbDataReader[\"ContactName\"] = " +
        myOleDbDataReader["ContactName"]);
    Console.WriteLine("myOleDbDataReader[\"Address\"] = " +
        myOleDbDataReader["Address"]);

    // close the OleDbDataReader object using the Close() method
    myOleDbDataReader.Close();

    // close the OleDbConnection object using the Close() method
    myOleDbConnection.Close();
}
}

```

---

The output from this program is as follows:

```

myOleDbDataReader["CustomerID"] = ALFKI
myOleDbDataReader["CompanyName"] = Alfreds Futterkiste
myOleDbDataReader["ContactName"] = Maria Anders
myOleDbDataReader["Address"] = Obere Str. 57

```

## Connecting to an Oracle Database

You connect to an Oracle database using an OleDbConnection object with a connection string of the following format:

```
provider=MSDAORA;data_source=OracleNetServiceName;user
```

`id=username;password=password`

where

- **OracleNetServiceName** Specifies the Oracle Net service name for the database. Oracle Net is a software component that allows you to connect to a database over a network. You'll need to speak with your DBA to get the Oracle Net service name.
- **username** Specifies the name of the database user you want to connect to the database as.
- **password** Specifies the password for the database user.

The following example creates a connection string named `connectionString` with the correct format to connect to an Oracle database:

```
string connectionString =
    "provider=MSDAORA;data source=ORCL;user id=SCOTT;password=TIGER";
```

**Note** The user ID of `SCOTT` with a password of `TIGER` is the default for accessing one of the example databases that comes with Oracle. This database contains a table called `emp` that contains sample employee data.

Assuming the `System.Data.OleDb` namespace has been imported, the following example creates an `OleDbConnection` object, passing `connectionString` to the constructor:

```
OleDbConnection myOleDbConnection =
    new OleDbConnection(connectionString);
```

[Listing 1.3](#) illustrates how to connect to an Oracle database using an `OleDbConnection` object and retrieve a row from the `emp` table.

Notice that you use an `OleDbCommand` and `OleDbDataReader` object to run a SQL statement and read the returned results from an Oracle database.

### **Listing 1.3: OLEDBCONNECTIONORACLE.CS**

---

```
/*
 OleDbConnectionOracle.cs illustrates how to use an
 OleDbConnection object to connect to an Oracle database
 */

using System;
using System.Data;
using System.Data.OleDb;

class OleDbConnectionOracle
{
    public static void Main()
    {
        // formulate a string containing the details of the
        // database connection
        string connectionString =
            "provider=MSDAORA;data source=ORCL;user id=SCOTT;password=TIGER";
        // create an OleDbConnection object to connect to the
        // database, passing the connection string to the constructor
        OleDbConnection myOleDbConnection =
            new OleDbConnection(connectionString);

        // create an OleDbCommand object
        OleDbCommand myOleDbCommand = myOleDbConnection.CreateCommand();

        // set the CommandText property of the OleDbCommand object to
        // a SQL SELECT statement that retrieves a row from the emp table
        myOleDbCommand.CommandText =
            "SELECT empno, ename, sal "+
```

```
"FROM emp "+  
"WHERE empno = 7369";  
  
// open the database connection using the  
// Open() method of the SqlConnection object  
myOleDbConnection.Open();  
  
// create an OleDbDataReader object and call the ExecuteReader()  
// method of the OleDbCommand object to run the SELECT statement  
OleDbDataReader myOleDbDataReader = myOleDbCommand.ExecuteReader();  
  
// read the row from the OleDbDataReader object using  
// the Read() method  
myOleDbDataReader.Read();  
  
// display the column values  
Console.WriteLine("myOleDbDataReader[\"empno\"] = "+  
    myOleDbDataReader["empno"]);  
Console.WriteLine("myOleDbDataReader[\"ename\"] = "+  
    myOleDbDataReader["ename"]);  
Console.WriteLine("myOleDbDataReader[\"sal\"] = "+  
    myOleDbDataReader["sal"]);  
  
// close the OleDbDataReader object using the Close() method  
myOleDbDataReader.Close();  
  
// close the OleDbConnection object using the Close() method  
myOleDbConnection.Close();  
}  
}
```

---

The output from this program is as follows:

```
myOleDbDataReader["empno"] = 7369  
myOleDbDataReader["ename"] = SMITH  
myOleDbDataReader["sal"] = 800
```

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## Introducing Visual Studio .NET

In the previous sections, you saw programs that connect to various databases, retrieve a row from a table, and display the column values for that row on your computer screen. This type of program is known as a *console application* because it displays output directly on the screen on which the program is running.

You can use Visual Studio .NET (VS .NET) to create console applications, as well as the following types of applications:

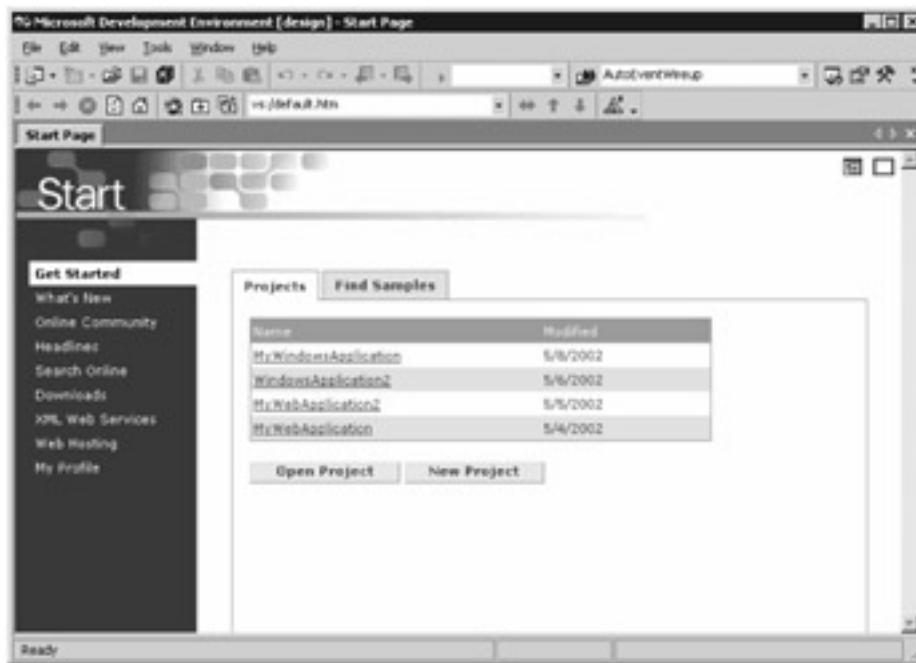
- **Windows Applications** These take advantage of the visual controls offered by the Windows operating system, such as menus, buttons, and editable text boxes. Windows Explorer, which you use to navigate the file system of your computer, is one example. You'll learn about Windows programming in [Chapter 6](#), "Introducing Windows Applications and ADO.NET."
- **ASP.NET Applications** These run over the Internet. You access an ASP.NET application using a Web browser, such as Internet Explorer. Examples of ASP.NET applications would be online banking, stock trading, or auction systems. You'll learn about ASP.NET programming in [Chapter 15](#), "Introducing Web Applications: ASP.NET."
- **ASP.NET Web Services** These also run over the Internet. Also known as XML Web services, the difference is that you can use them to offer a service that could be used in a distributed system of interconnected services. For example, Microsoft's Passport Web service offers identification and authentication of Web users you could then use in your own Web application. You'll learn about Web services in [Chapter 17](#), "Web Services."

This is not an exhaustive list of the types of applications you can develop with VS .NET, but it does give you flavor for the broad range of VS .NET's capabilities.

In the rest of this section, you'll see how to develop and run a console application using VS .NET. If you've installed VS .NET on your computer, you'll be able to follow along with the example. If you don't have VS .NET, don't worry; you'll still be able to see what's going on from the figures provided.

## Starting Visual Studio .NET and Creating a Project

All of your work in VS .NET is organized into *projects*. Projects contain the source and executable files for your program, among other items. If you have VS .NET installed, start it by selecting Start > Programs > Microsoft Visual Studio .NET > Microsoft Visual Studio .NET. Once VS .NET has started, you'll see the Start page (see [Figure 1.1](#)).



**Figure 1.1:** The Start page

From the Start page, you can see any existing projects you've created. You can open and create projects using the Open Project and New Project buttons, respectively. You'll create a new project shortly.

## Using the VS .NET Links

As you can see from [Figure 1.1](#), VS .NET contains a number of links on the left of the Start page. Some of these links provide access to useful information on the Internet about .NET; the links are as follows:

- **Get Started** Open the Start page.
- **What's New** View any updates for VS .NET or Windows. You can also view upcoming training events and conferences.
- **Online Community** Get in touch with other members of the .NET community. Includes links to Web sites and newsgroups.
- **Headlines** View the latest news on .NET.
- **Search Online** Search the MSDN Online Library for technical material such as published articles on .NET.
- **Downloads** Download trial applications and example programs from the Web sites featured here.
- **XML Web Services** Find registered XML Web services that you can then use in your own programs. XML Web services are also known as ASP.NET Web services. You'll learn more about Web services in [Chapter 17](#).
- **Web Hosting** A Web hosting company can take your program and run it for you. It takes care of the computers on which your program runs. Use the Web Hosting link to view companies that provide these services.
- **My Profile** Set items such as your required keyboard scheme and window layout.

Click these links and explore the information provided. As you'll see, there's a lot of information about .NET on the Internet.

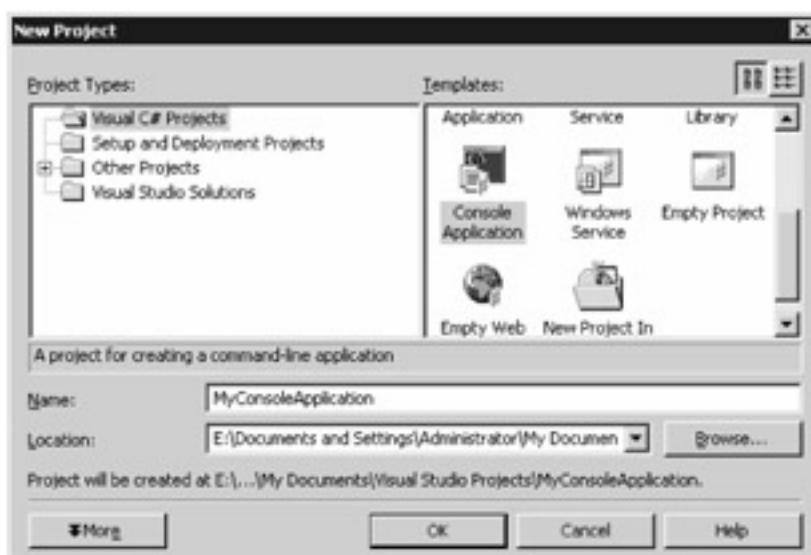
## Creating a New Project

When you're finished examining the information in the previous links, create a new project by clicking the New Project button on the Get Started page.

**Note** You can also create a new project by selecting File > New > Project, or by pressing Ctrl+Shift+N on your keyboard.

When you create a new project, VS .NET displays the New Project dialog box, which you use to select the type of project you want to create. You also enter the name and location of your new project; the location is the directory where you want to store the files for your project.

Because you're going to be creating a C# console application, select Visual C# Projects from the Project Types section on the left of the New Project dialog box, and select Console Application from the Templates section on the right. Enter **MyConsoleApplication** in the Name field, and keep the default directory in the Location field. [Figure 1.2](#) shows the completed New Project dialog box with these settings.

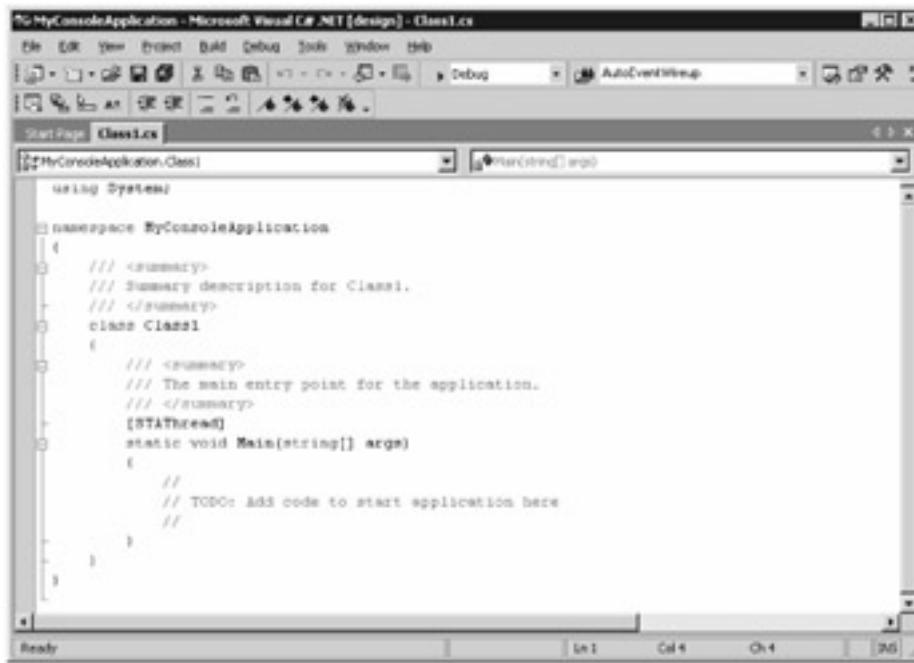


**Figure 1.2:** The New Project dialog box with the appropriate settings for a C# console application

Click the OK button to create the new project.

## Working in the VS .NET Environment

Once you've created a new project, the main development screen is displayed (see [Figure 1.3](#)). This screen is the environment in which you'll develop your project. As you can see, VS .NET has already created some starting code for you. This code is a skeleton for your program; you'll see how to modify it shortly. In this section, I'll give you a brief description of the different parts of the VS .NET environment.



**Figure 1.3:** The VS .NET environment

**Note** Depending on your settings for VS .NET, your screen might look slightly different from that shown in [Figure 1.3](#).

The VS .NET menu contains the following items:

- **File** Open, close, and save project files.
- **Edit** Cut, copy, and paste text from the Clipboard. The Clipboard is a temporary storage area.
- **View** Hide and show different windows such as the Solution Explorer (which lets you see the files that make up your project), Class View (which lets you see the classes and objects in your project), Server Explorer (which lets you explore items such as databases), and the Properties window (which lets you set the properties of objects, such as the size of a button). You can also use the View menu to select the toolbars you want to display.
- **Project** Add class files to your project and add Windows forms and controls.
- **Build** Compile the source files in your project.
- **Debug** Start your program with or without debugging. Debugging lets you step through your program line by line, looking for errors.
- **Tools** Connect to a database and customize your settings for VS .NET. For example, set the colors used for different parts of your program lines or set the initial page displayed by VS .NET when you start it.
- **Window** Switch between files you've opened and hide windows.
- **Help** Open the documentation on .NET. You'll learn how to use this documentation later in this chapter in the section "[Using the .NET Documentation](#)."

The VS .NET toolbar contains a series of buttons that act as shortcuts to some of the menu options. For example, you can save a file or all files, cut and paste text from the Clipboard, and start a program using the debugger. You'll learn how to use some of these features later in

this chapter.

The code shown in the window (below the toolbar) with the title `Class1.cs` is code that is automatically generated by VS .NET, and in the [next section](#) you'll modify this code.

## Modifying the VS .NET-Generated Code

Once VS .NET has created your project, it will display some starting code for the console application with a class name of `Class1.cs`. You can use this code as the beginning for your own program. [Figure 1.3](#), shown earlier, shows the starting code created by VS .NET.

The `Main()` method created by VS .NET is as follows:

```
static void Main(string[] args)
{
    //
    // TODO: Add code to start application here
    //
}
```

As you can see, this code contains comments that indicate where you add your own code. Replace the `Main()` method with the following code taken from the `Main()` method in `FirstExample.cs`, shown earlier in [Listing 1.1](#):

```
public static void Main()
{
    try
    {
        // step 1: create a SqlConnection object to connect to the
        // SQL Server Northwind database
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        // step 2: create a SqlCommand object
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        // step 3: set the CommandText property of the SqlCommand object to
        // a SQL SELECT statement that retrieves a row from the Customers table
        mySqlCommand.CommandText =
            "SELECT CustomerID, CompanyName, ContactName, Address "+
            "FROM Customers "+
            "WHERE CustomerID = 'ALFKI'";

        // step 4: open the database connection using the
        // Open() method of the SqlConnection object
        mySqlConnection.Open();

        // step 5: create a SqlDataReader object and call the ExecuteReader()
        // method of the SqlCommand object to run the SELECT statement
        SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();

        // step 6: read the row from the SqlDataReader object using
        // the Read() method
        mySqlDataReader.Read();

        // step 7: display the column values
        Console.WriteLine("mySqlDataReader[\"CustomerID\"] = "+
            mySqlDataReader["CustomerID"]);
        Console.WriteLine("mySqlDataReader[\"CompanyName\"] = "+
            mySqlDataReader["CompanyName"]);
    }
}
```

```
Console.WriteLine("mySqlDataReader[\"ContactName\"] = "+  
    mySqlDataReader["ContactName"]);  
Console.WriteLine("mySqlDataReader[\"Address\"] = "+  
    mySqlDataReader["Address"]);  
  
// step 8: close the SqlDataReader object using the Close() method  
mySqlDataReader.Close();  
  
// step 9: close the SqlConnection object using the Close() method  
mySqlConnection.Close();  
}  
catch (SqlException e)  
{  
    Console.WriteLine("A SqlException was thrown");  
    Console.WriteLine("Number = " + e.Number);  
    Console.WriteLine("Message = " + e.Message);  
    Console.WriteLine("StackTrace:\n" + e.StackTrace);  
}  
}
```

**Note** You'll also need to add the following line near the start of your class: `using System.Data.SqlClient;`

Once you've added the previous code, your next steps are to compile and run your program.

## Compiling and Running the Program Using VS .NET

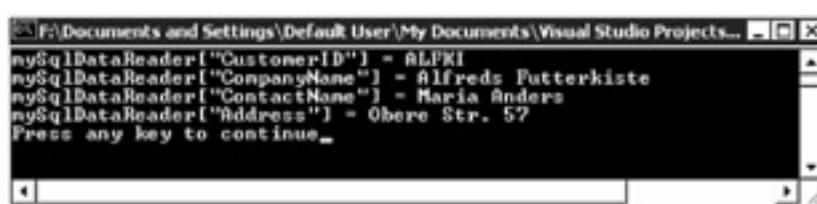
As always, you must first compile your program before you can run it. Because programs in VS .NET are organized into projects, you must compile the project; this is also known as *building* the project. To build your project, select Build > Build Solution. This compiles the `Class1.cs` source file into an executable file.

**Tip** You can also press Ctrl+Shift+B on your keyboard to build your project.

Finally, you can now run your program. Select Debug > Start Without Debugging. When you select Start Without Debugging, the program will pause at the end, allowing you to view the output.

**Tip** You can also press Ctrl+F5 on your keyboard to run your program.

When you run your program, VS .NET will run the program in a new Command Prompt window, as shown in [Figure 1.4](#). Your program is run in this window because it is a console application.



**Figure 1.4:** The running program

To end the program, press any key. This will also close the Command Prompt window.

You've barely scratched the surface of VS .NET in this section. You'll explore some of the other features of VS .NET later in this book. In the [next section](#), you'll learn how to use the extensive documentation that comes with .NET.

## Using the .NET Documentation

Both the .NET SDK and VS .NET come with extensive documentation, including the full reference to all the classes in .NET. As you become proficient with C#, you'll find this reference documentation invaluable.

In the following sections, you'll see how to access and search the .NET documentation, and view some of the contents of the documentation. Depending on whether you're using the .NET SDK or VS .NET, you access the documentation in a slightly different way. You'll see how to use both ways to access the documentation in this section.

**Note** The documentation that comes with the .NET SDK is a subset of the documentation that comes with VS .NET.

## Accessing the Documentation Using the .NET SDK

If you're using the .NET SDK, you access the documentation by selecting Start > Programs > Microsoft .NET Framework SDK > Documentation. [Figure 1.5](#) shows the .NET Framework SDK document home page; this is the starting page for the documentation.

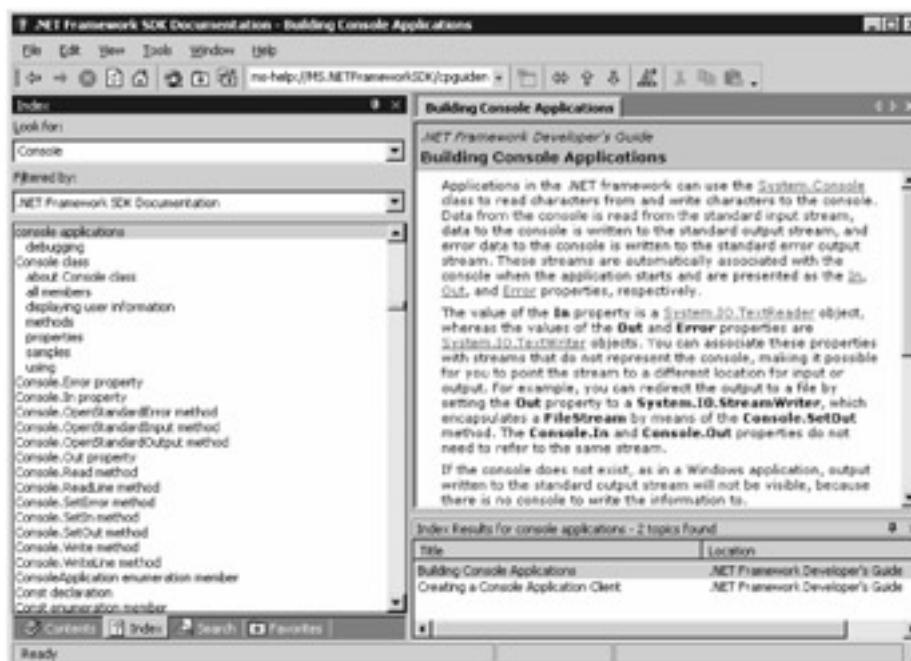


**Figure 1.5:** The documentation home page

On the left of the page, you can see the various sections that make up the contents of the documentation. You can view the index of the documentation by selecting the Index tab at the bottom of the page.

**Tip** You can also view the Index window by selecting Help > Index, or by pressing Ctrl+Alt+F2 on your keyboard.

You can search the index by entering a word in the Look For field of the Index tab. [Figure 1.6](#) shows the results of searching for **Console**. [Figure 1.6](#) also shows the text for the details on building console applications on the top right of the screen. I opened this overview by double-clicking the Building Console Applications link in the Index Results on the bottom right of the screen.

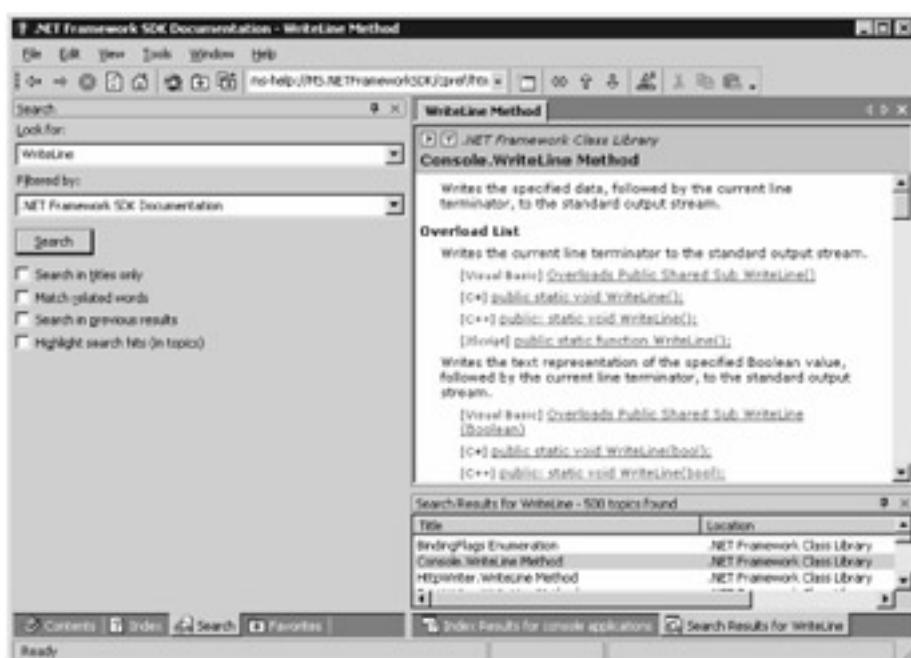


**Figure 1.6:** Searching the index for the word console

You can also search all pages in the documentation using the Search tab. You display the Search tab by selecting it from the bottom of the screen.

**Tip** You can also view the Search window by selecting Help > Search, or by pressing Ctrl+Alt+F3 on your keyboard.

You enter the words you want to search for in the Look For field of the Search window. [Figure 1.7](#) shows the search page and the search results returned by a search for *WriteLine*. When you run the search, the names of the pages that contain your required words are displayed in the Search Results window that appears at the bottom of the screen (you can see this window in [Figure 1.7](#)).



**Figure 1.7:** Searching all of the documentation for the word WriteLine

**Tip** You can also view the Search Results window by selecting Help > Search results, or by pressing Shift+Alt+F3 on your keyboard.

You view the contents of a particular page shown in the Search Results window by double-clicking the appropriate line. For example, in [Figure 1.7](#), we double-clicked the second line in the Search Results window. This line contained the page with the title "Console.WriteLine Method," and as you can see, this page is displayed in the window above the Search Results in [Figure 1.7](#).

In the [next section](#), you'll see how to access the documentation using VS .NET.

## Accessing the Documentation Using VS .NET

If you're using VS .NET, you access the documentation using the Help menu. To access the contents of the documentation, you select Help ➤ Contents. [Figure 1.8](#) shows the contents displayed in VS .NET. Notice that the documentation is displayed directly in VS .NET, rather than in a separate window, as is done when viewing documentation with the .NET SDK.



**Figure 1.8:** The documentation contents viewed in VS .NET

**Note** The same keyboard shortcuts shown in the [previous section](#) also apply to VS .NET.

The Help menu also provides access to similar Index and Search windows as you saw in the [previous section](#).

**Team LiB**

◀ PREVIOUS ▶ NEXT ▶

## Using the SQL Server Documentation

SQL Server also comes with extensive electronic documentation. To access this documentation, you select Start > Programs > Microsoft SQL Server > Books Online. [Figure 1.9](#) shows the SQL Server documentation home page.



**Figure 1.9:** SQL Server documentation home page

You can browse the online books using the Contents tab, and you can search for specific information using the Index and Search tabs. [Figure 1.10](#) shows some of the information for the SELECT statement, which is located in the Transact-SQL reference book.



**Figure 1.10:** SELECT examples documentation

**Note** Transact-SQL is Microsoft's implementation of SQL and contains programming extensions. You'll learn about Transact-SQL programming in [Chapter 4](#).

You can see the information shown in [Figure 1.10](#) yourself by opening Contents > Transact-SQL Reference > SELECT > SELECT Examples.



## Summary

A **database** is an organized collection of information that is divided into **tables**. Each table is further divided into **rows** and **columns**; these columns store the actual information. You access a database using the **Structured Query Language** (SQL), which is a standard language supported by most database software including SQL Server, Access, and Oracle.

You saw a C# program that connected to a SQL Server database, retrieved and displayed the contents stored in the columns of a row from a table, and then disconnected from the database. You also saw programs that connected to an Access and an Oracle database.

Microsoft's Rapid Application Development (RAD) tool is Visual Studio .NET (VS .NET). VS .NET enables you to develop and run programs in an integrated development environment. This environment uses all the great features of Windows, such as the mouse and intuitive menus, and increases your productivity as a programmer.

In the final sections of this chapter, you saw how to use the extensive documentation from Microsoft that comes with the .NET Software Development Kit (SDK) and VS .NET. You also saw how to use the SQL Server documentation.

In the [next chapter](#), you'll learn more about databases.

## Chapter 2: Introduction to Databases

### Overview

In this chapter, you'll learn the basics of databases: how databases are constructed, how to create and relate tables, and how to build queries to retrieve information. This chapter also shows you how to use a SQL Server database named Northwind. This database contains the information for the fictitious Northwind Company, which sells food products. This database is one of the example databases that is typically installed with SQL Server. You can obtain a trial version of SQL Server from Microsoft's website at [www.microsoft.com](http://www.microsoft.com).

**Note** At time of writing, you can download the trial version of SQL Server from Microsoft's website. If your Internet connection is too slow, you can also order a CD-ROM containing the trial version.

I used the Developer edition of SQL Server when preparing this book. When running a production system, you should typically use the Enterprise edition of SQL Server. You can view the differences between the various types of SQL Server at Microsoft's website.

Featured in this chapter:

- Introducing databases
- Using SQL Server
- Exploring the Northwind database
- Building queries using Enterprise Manager
- [Creating a table](#)

## Introducing Databases

A database is an organized collection of information. A *relational database* is a collection of related information that has been organized into structures known as *tables*. Each table contains *rows* that are further organized into *columns*. You should already be familiar with information being represented in the form of a table with columns. For example, [Table 2.1](#) shows the details of some products sold by the Northwind Company. [Table 2.1](#) lists the product ID, name, quantity per unit, and unit price for the first 10 products; this information comes from the `Products` table of the Northwind database.

**Table 2.1: SOME ROWS FROM THE PRODUCTS TABLE**

PRODUCT ID	NAME	QUANTITY PER UNIT	Unit Price
1	Chai	10 boxes x 20 bags	\$18
2	Chang	24-12oz bottles	\$19
3	Aniseed Syrup	12-550ml bottles	\$10
4	Chef Anton's Cajun Seasoning	48-6oz jars	\$22
5	Chef Anton's Gumbo Mix	36 boxes	\$21.35
6	Grandma's Boysenberry Spread	12-8oz jars	\$25
7	Uncle Bob's Organic Dried Pears	12-1lb pkgs.	\$30
8	Northwoods Cranberry Sauce	12-12oz jars	\$40
9	Mishi Kobe Niku	18-500g pkgs.	\$97
10	Ikura	12-200ml jars	\$31

You can store the information in a database on paper in a filing cabinet or in electronic format stored in the memory and file system of a computer. The system used to manage the information in the database is the *database management system*. In the case of an electronic database, the database management system is the software that manages the information in the computer's memory and files. One example of such software is SQL Server (this is the relational database management system, or *RDBMS*, used in this book). Other examples of RDBMS software include Oracle and DB2.

**Note** You must be careful to differentiate between a database and a database management system. A database is an organized collection of information, and a database management system is the software that stores and provides the tools to manipulate the stored information. This distinction is blurred these days, so the term database is often used to refer to the software.

Another term you need to be familiar with is a database *schema*, which is a representation of the structure of data, and includes the definition of the tables and columns that make up the database.

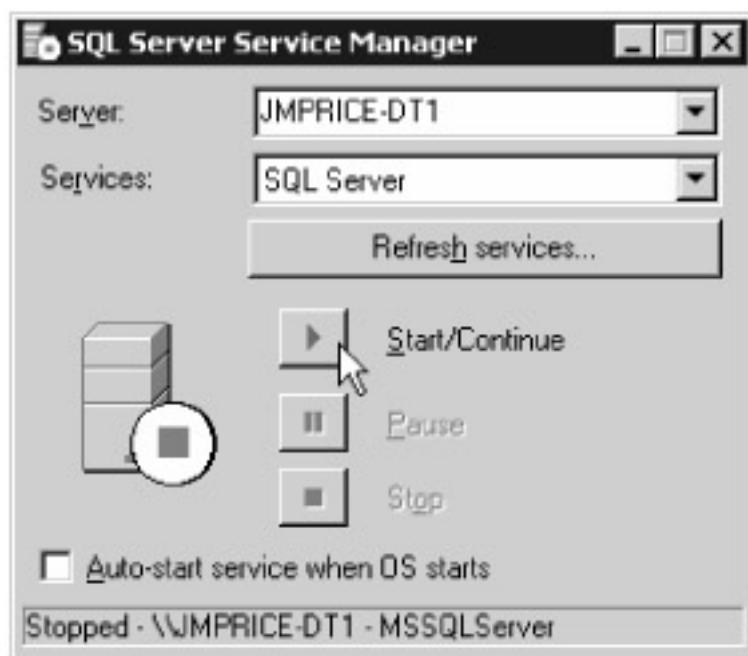
In the [next section](#), you'll explore SQL Server.

## Using SQL Server

In this section, you'll explore some of the tools you use to manage SQL Server. Specifically, you'll learn how to start and stop SQL Server using the Service Manager and use the Enterprise Manager to administer SQL Server.

### Starting and Stopping SQL Server

To start and stop SQL Server, you use the Service Manager tool. To open the Service Manager, you select Start > Programs > Microsoft SQL Server > Service Manager. The Service Manager is shown in [Figure 2.1](#).



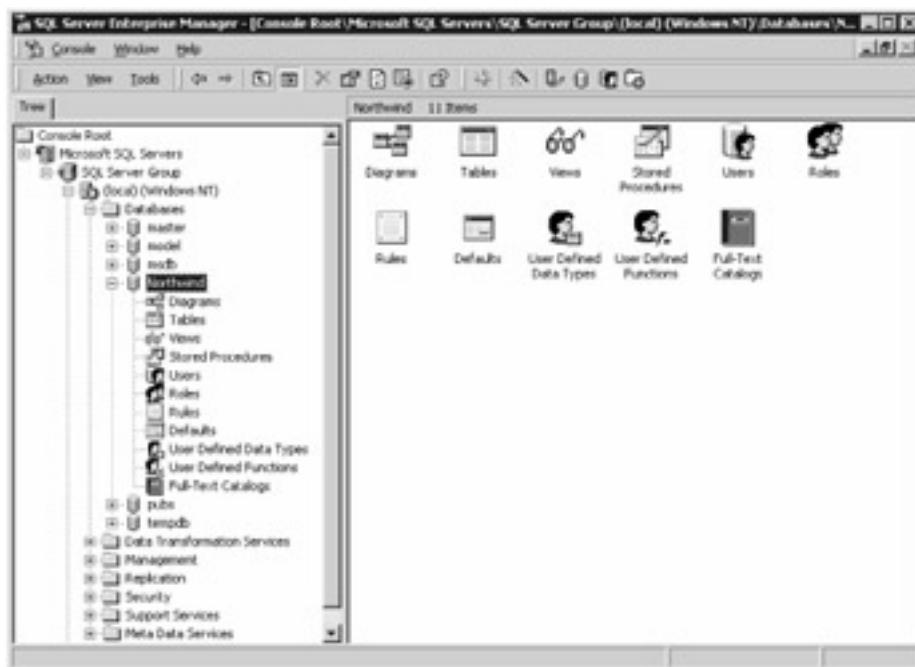
**Figure 2.1:** The Service Manager

You select the name of the server computer on which SQL Server is running in the Server drop-down list box. To start SQL Server, you click the Start/Continue button. To stop SQL Server, you click the Stop button. You can also use the Service Manager to pause SQL Server, and select whether you want to automatically start SQL Server when the operating system (OS) starts.

Once you've started SQL Server, other programs can access the databases managed by that SQL Server installation.

### Using Enterprise Manager

To administer a database, you use the Enterprise Manager tool. You can create databases, create and edit tables, create and edit users, and so on, using Enterprise Manager. To open the Enterprise Manager, you select Start > Programs > Microsoft SQL Server > Enterprise Manager. The Enterprise Manager is shown in [Figure 2.2](#).



**Figure 2.2:** The Enterprise Manager

On the left pane of Enterprise Manager, you'll see a tree that shows the accessible SQL Server installations. The contents of the right pane of Enterprise Manager display different information based on what you select in the left pane. For example, I selected the Databases folder and the North-wind database in the left pane when preparing [Figure 2.2](#). As you can see, the right pane displays icons that allow you to edit the items stored in that database.

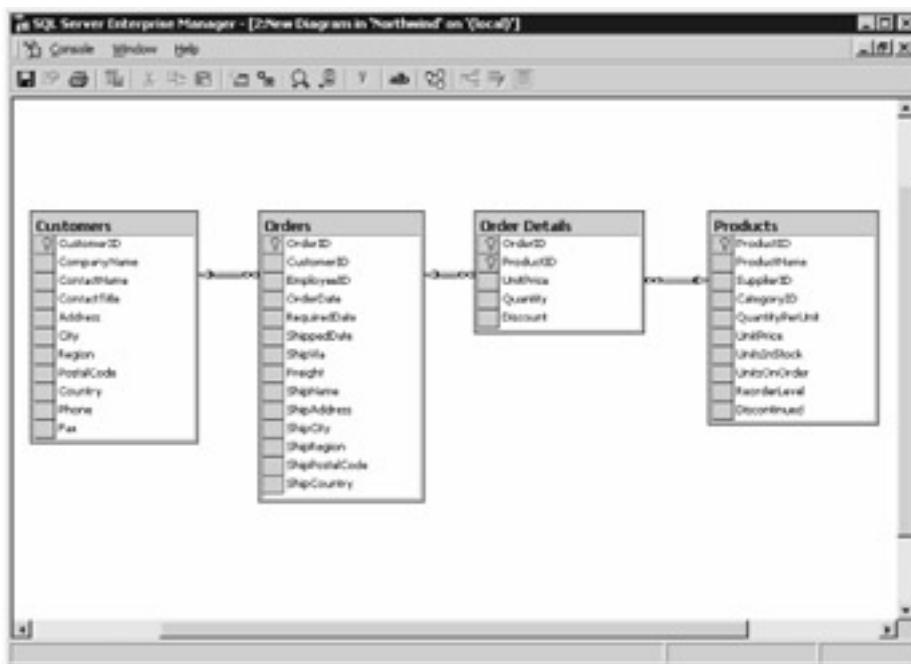
Each SQL Server installation contains the following seven folders shown in the left pane:

- **Databases** Contains tools that allow you to access the databases managed by SQL Server.
- **Data Transformation Services** Provides access to tools that allow you to move data from one database to another. You can also programmatically modify the data as it is moved. For example, you might want to move data from SQL Server database to an Oracle database, or vice versa.
- **Management** Contains tools that allow you to back up your databases, monitor current database activity, and other tasks.
- **Replication** Provides access tools that allow you to copy information from one database to another in near real time using a process known as *replication*. For example, you might want to move data from a database running at a branch office of a company to a database at headquarters.
- **Security** Contains tools that allow you to manage logins and built-in roles that contain permissions. You can also manage *linked* servers and *remote* servers. Linked servers are databases that you can access over a network. These databases don't have to be SQL Server databases; they could also be Oracle databases, for example. The only limitation is that there must be an OLE DB (Object Linking and Embedding for Databases) provider for that database. Remote servers are SQL Server databases that you can access over a network and run stored procedures on.
- **Support Services** Provides access to tools that allow you to manage the Distributed Transaction Coordinator, Full-Text Search, and SQL Mail services. The Distributed Transaction Coordinator service allows you to manage transactions that use more than one database. The Full Text Search service allows you to perform searches for phrases through large amounts of text. The SQL Mail service allows you to send electronic mail from SQL Server.
- **Meta Data Services** Contains tools that allow you to manage the information stored in the local repository. This information contains details about databases, users, tables, columns, views, stored procedures, and so on. This information is primarily used by data-warehousing applications.

**Note** Since this is a book on database programming, I won't cover too many details on database administration; I'll just focus on the Databases folder. Typically, your organization will have a database administrator, or DBA, who takes care of administering your databases and will use the other folders to perform their tasks. If you need more details on administering SQL Server, I recommend the book *Mastering SQL Server 2000* by Mike Gunderloy and Joseph L. Jorden (Sybex, 2000).

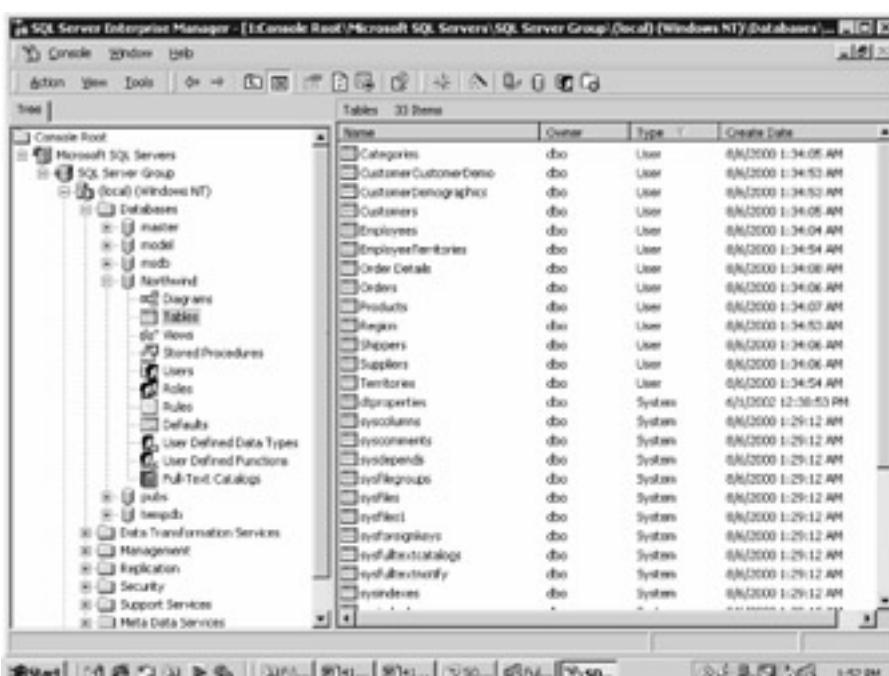
Let's take a closer look at the Databases folder, which contains the databases that are managed by a particular SQL Server installation. For example, my SQL Server installation manages six databases named master, model, msdb, Northwind, pubs, and tempdb. When you expand the Databases folder for a database, you'll see the following nodes:

- **Diagrams** You use a diagram to store a visual representation of the tables in a database. For example, the Northwind database contains many tables, four of which are named *Customers*, *Orders*, *Order Details*, and *Products*. [Figure 2.3](#) illustrates how these tables are related. The columns for each table are shown within each box in the diagram. For example, the *Customers* table contains 11 columns: CustomerID, CompanyName, ContactName, ContactTitle, Address, City, Region, PostalCode, Country, Phone, and Fax. As you'll learn in the "[Table Relationships and Foreign Keys](#)" section, the lines that connect the tables show the relationships between the various tables.



**Figure 2.3:** The Customers, Orders, Order Details, and Products tables

- **Tables** You use a table to store rows that are divided into columns. [Figure 2.4](#) shows a list of the tables stored in the Northwind database.



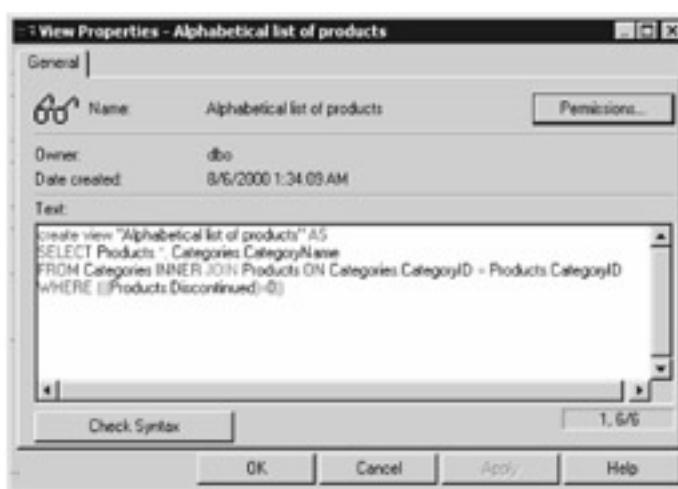
**Figure 2.4:** The tables of the Northwind database

You can create new tables, view the properties of a table, and query the rows stored in a table. You'll learn how to create a new table later in the "[Creating a Table](#)" section. To view the properties of a table, you select the table from the list in the right pane, click the right mouse button, and select Properties from the context-sensitive pop-up menu. You can also double-click the table to display the properties, and [Figure 2.5](#) shows the properties of the `Customers` table. You'll learn the meaning of these properties as this chapter progresses.



**Figure 2.5:** The `Customers` table properties

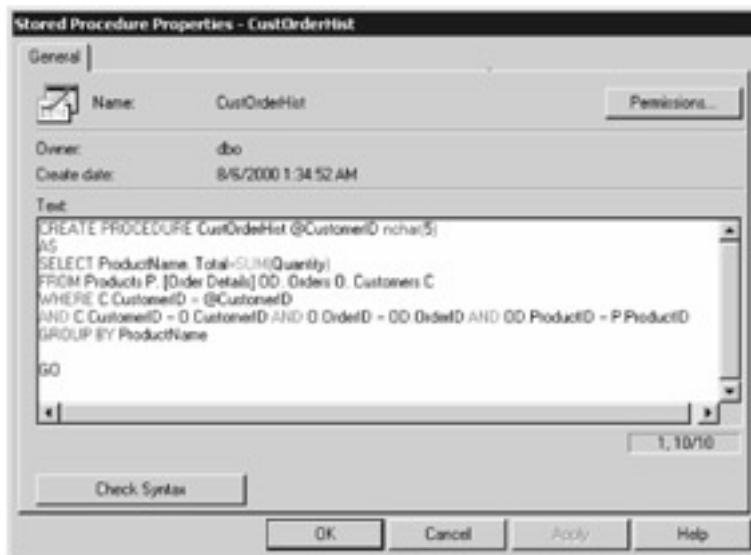
- **Views** You use a view to retrieve a set of columns from one or more tables. You can think of a view as a more flexible way of examining the rows stored in the tables. For example, one of the views of the Northwind database retrieves an alphabetical list of products, and retrieves the product name and the category name, among other columns. This information comes from both the `Products` and `Categories` tables. You can create new views, examine the properties of a view, and query the rows through a view. To examine the properties of a view, you select the view, click the right mouse button, and select Properties. You can also double-click the view to examine the Properties. [Figure 2.6](#) shows the properties for the alphabetical list of products view. The text of the view is written in SQL, which you'll learn more about in [Chapter 3](#), along with how to use the view in that chapter.



**Figure 2.6:** The alphabetical list of products view properties

- **Stored Procedures** You use a stored procedure to run a sequence of statements in the database. In SQL Server, stored procedures are written in Transact-SQL, which you'll learn about in [Chapter 4](#). Stored procedures are saved in the database, and are typically used when you need to perform a task that intensively uses the database, or you want to centralize a function in the database that any user can call rather than have each user write their own program to perform the same task. For example, one of the stored procedures in the Northwind database is named `CustOrdHist`, which returns the product name and the sum of the quantity of

products ordered by a particular customer, who is passed as a parameter to the procedure. [Figure 2.7](#) shows the properties for the `CustOrdHist` stored procedure.



**Figure 2.7:** The `CustOrdHist` stored procedure properties

- **Users** Every time you access the database, you connect to a particular user account in the database. Every SQL Server database comes with two default users named `dbo` and `guest`. The `dbo` user owns the database and has the permissions to do anything in the database, such as create new tables, edit tables, and so on. The `guest` user has more limited permissions that allow access to the contents of the tables, but not the ability to create or edit tables, and so on. [Figure 2.8](#) shows the properties of the `dbo` user. You'll notice that the `dbo` user has been granted two roles, `public` and `db_owner`. You'll learn about roles next. You can view all the permissions assigned to the `dbo` user by clicking the Permissions button.



**Figure 2.8:** The `dbo` user properties

- **Roles** A role is a named set of permissions that you can assign to a user. It is useful when you need to assign the same set of permissions to more than one user. That way, if you need to change the set of permissions, you need to change only the permissions assigned to the role, rather than the permissions assigned to each user. For example, you saw in the previous figure that the `dbo` user has been granted the `public` and `db_owner` roles. [Figure 2.9](#) shows the properties of the `public` role. You'll notice that the `public` role has also been granted to the `guest` user. If no `public` role was used, then the set of permissions would have to be added by hand to both the `dbo` and `guest` users.

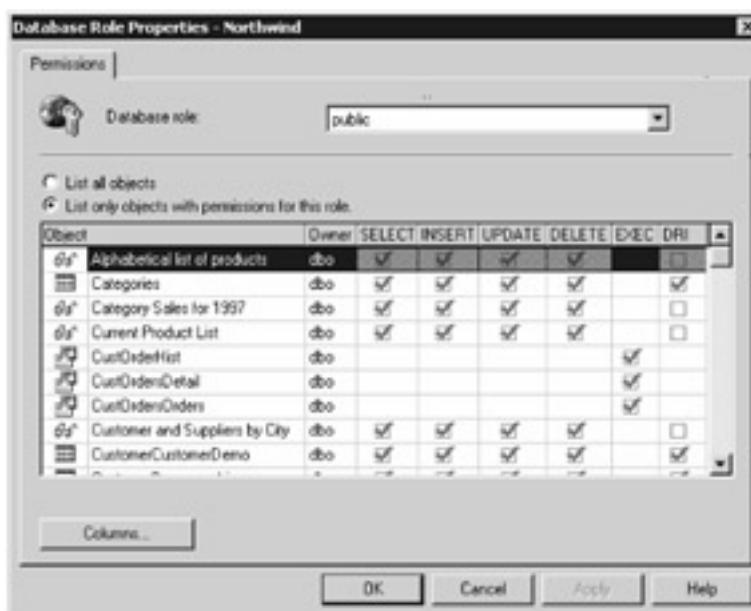


**Figure 2.9:** The public role properties

You can view the permissions assigned to a role by clicking the Permissions button. [Figure 2.10](#) shows the properties assigned to the public role, and [Table 2.2](#) lists the meaning of the available permissions.

**Table 2.2: MEANING OF THE AVAILABLE PERMISSIONS**

PERMISSION	MEANING
SELECT	Allows retrieval of rows from a table or view.
INSERT	Allows addition of rows into a table or view.
UPDATE	Allows modification of rows in a table or view.
DELETE	Allows removal of rows from a table or view.
EXEC	Allows execution of a stored procedure.
DRI	Allows the addition or removal of declarative referential integrity (DRI) constraints to a table. The constraints ensure that proper actions are taken when adding, modifying, or removing foreign key values. Foreign keys specify that a column in one table is related to a column in another table. You'll learn more about foreign keys later in the " <a href="#">Table Relationships and Foreign Keys</a> " section.



**Figure 2.10:** The public role permissions

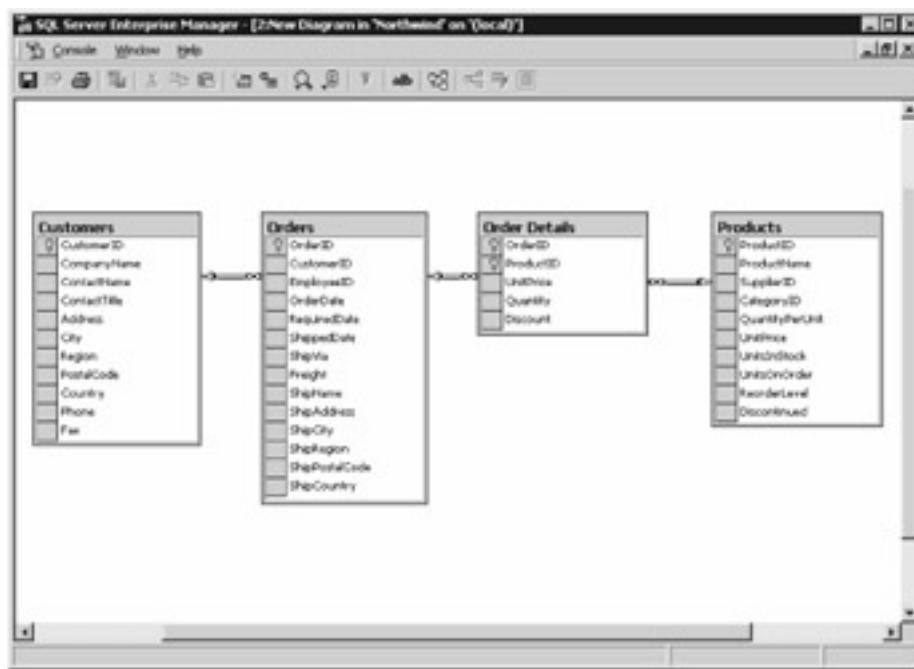
- **Rules** A *rule* is an expression that evaluates to either `true` or `false` and determines whether you are able to assign a particular value to a column. For example, you might define a rule that specifies a range of values, and if a supplied value is beyond that range, then you cannot assign that value to the column. Rules are provided for compatibility with older versions of SQL Server and are now replaced by constraints. You'll learn more about constraints in the "[Creating a Constraint](#)" section later.
- **Defaults** A *default* value is an initial value that is set when you add a new row to a table. Defaults are provided for compatibility with older versions of SQL Server and are now replaced by the default value of a column. You'll learn more about default values in the "[Creating a Table](#)" section later.
- **User-Defined Data Types** User-defined data types allow you to create your own types based on the existing SQL Server types. For example, say you wanted to store a United States ZIP code in several tables of your database; you could create a type that stores a string of five characters. If you then wanted to increase the length from five to eight to store an extended ZIP code, then all you need to do is modify your type and the change will be reflected in all the tables where you used that type.
- **User-Defined Functions** User-defined functions allow you to create your own functions. For example, say you wanted to compute the distance traveled by an object, then you could create a function to do that.
- **Full-Text Catalogs** Full-text catalogs allow you to create a full-text index, which enables you to perform searches for phrases through large amounts of text.

In the [next chapter](#), you'll see that Visual Studio .NET's Server Explorer also allows you to use many of the same features contained in the Databases folder of Enterprise Manager. Specifically, Server Explorer allows you to view, create, and edit the following items: database diagrams, tables, views, stored procedures, and user-defined functions.

In the following section, you'll learn what is meant by the term *relational* in the context of a relational database, and you'll explore some of the tables in the Northwind database.

## Exploring the Northwind Database

A database may have many tables, some of which are related to each other. For example, the Northwind database contains many tables, four of which are named `Customers`, `Orders`, `Order Details`, and `Products`. [Figure 2.11](#) is a repeat of the diagram shown earlier that illustrates how these tables are related.



**Figure 2.11:** Relationships between the `Customers`, `Orders`, `Order Details`, and `Products` tables

The columns for each table are shown within each box. For example, the `Customers` table contains 11 columns:

- `CustomerID`
- `CompanyName`
- `ContactName`
- `ContactTitle`
- `Address`
- `City`
- `Region`
- `PostalCode`
- `Country`
- `Phone`
- `Fax`

In the next few sections, you'll learn some database theory, and then you'll learn how each of the previous columns is defined in the `Customers` table. You'll also explore the `Orders`, `Order Details`, and `Products` tables.

## Primary Keys

Typically, each table in a database has one or more columns that uniquely identify each row in the table. This column is known as the **primary key** for the table. A primary key can be composed of more than one column. In such cases, it is known as a **composite key**.

**Note** The value for the primary key in each row of a table must be unique.

In the case of the `Customers` table, the primary key is the `CustomerID` column. The key icon shown to the left of the `CustomerID` column in [Figure 2.11](#) indicates that this column is the primary key for the `Customers` table. Similarly, the primary key for the `Orders` table is `OrderID`. The primary key for the `Order Details` table is composed of two columns: `OrderID` and `ProductID`. The primary key for the `Products` table is `ProductID`.

## Table Relationships and Foreign Keys

The lines that connect the tables in [Figure 2.11](#), shown earlier, display the relationships between the tables. The infinity sign ( $\infty$ ) at the end of each line indicates a **one-to-many** relationship between two tables, meaning that a row in one table can be related to one or more rows in the other table.

For example, the `Customers` table has a one-to-many relationship with the `Orders` table. Each customer can place many orders. Similarly, the one-to-many relationship between the `Orders` and `Order Details` table means that each order can be made up of many order details (you can think of an order detail as a line in a purchase order list, with each line referring to a specific product that is ordered). Finally, the one-to-many relationship between the `Products` and `Order Details` table means that each product can appear in many order details.

One-to-many relationships are modeled using **foreign keys**. For example, the `Orders` table has a column named `CustomerID`. This column is related to the `CustomerID` column in the `Customers` table through a foreign key. This means that every row in the `Orders` table must have a corresponding row in the `Customers` table with a matching value for the `CustomerID` column. For example, if a row in the `Orders` table has a `CustomerID` of `ALFKI`, then there must also be a row in the `Customers` table with a `CustomerID` of `ALFKI`. Since the relationship between the `Customers` and `Orders` table is one-to-many, this means that there can be many rows in the `Orders` table with the same `CustomerID` column. Conceptually, you can think of the foreign key as a pointer from the `Orders` table to the `Customers` table.

Often, the table containing the foreign key is known as the **child table**, and the table with the column referenced by the foreign key is known as the **parent table**. For example, the `Orders` table is the child table, and the `Customers` table is the parent table. Foreign key relationships are often known as **parent-child relationships**.

**Note** The relational term from "relational database" comes from the fact that tables can be related to each other through foreign keys.

You can manage the relationships for a table from Enterprise Manager by selecting the table from the Tables node, clicking the right mouse button, and selecting Design Table. You then click the Manage Relationships button on the toolbar of the table designer. For example, [Figure 2.12](#) shows the relationship between the `Customers` and `Orders` tables.



**Figure 2.12:** Relationship between the `Customers` and `Orders` table

The `Customers` and `Orders` tables are related through the `CustomerID` column. The `CustomerID` column in the `Orders` table is the foreign key. The relationship between the two tables is named `FK_Orders_Customers`.

## Null Values

Databases must also provide the ability to handle values that are not set, or are otherwise unknown. Unknown values are called *null values*, and a column is defined as allowing or disallowing null values. When a column allows null values, that column is defined as null; otherwise it is defined as not-null. A not-null column in a row must always have value stored in it. If you tried to add a row but didn't supply a value to a not-null column, then the database would display an error and wouldn't add your new row.

## Indexes

When looking for a particular topic in a book, you can either scan the whole book looking for your topic, or you can use the book's index to find the exact location of the topic directly. An index for a database table is similar in concept to a book index, except that database indexes are used to find specific rows in a table. The downside of indexes is that when a row is added to the table, additional time is required to update the index for the new row.

Generally, you should only create an index on a column when you find that you are retrieving a small number of rows from a table containing many rows. A good rule of thumb is that an index is useful when you expect any single query to retrieve 10 percent or less of the total rows in a table. This means that the candidate column for an index should be used to store a wide range of values. A good candidate for indexing would be a column containing a unique number for each record, while a poor candidate for indexing would be a column that contains only a small range of numeric codes such as 1, 2, 3, or 4. This consideration applies to all database types, not just numbers.

**Note** SQL Server automatically creates an index for the primary key *column of a table*.

Normally, the DBA is responsible for creating indexes, but as an application developer, you probably know more about your application than the DBA and will be able to recommend which columns are good candidates for indexing.

You can manage the indexes for a table from Enterprise Manager by selecting the table from the Tables node, clicking the right mouse button, and selecting All Tasks > Manage Indexes. For example, [Figure 2.13](#) shows the indexes for the `Customers` table. You can also manage indexes from the table designer by clicking the Manage Indexes/Keys button.

**Figure 2.13:** Indexes for the Customers table

The `Customers` table has five indexes: one each on the `CustomerID`, `City`, `CompanyName`, `PostalCode`, and `Region` columns.

You'll learn how to add an index to a table in the "[Creating an Index](#)" section later.

## Column Types

Each column in a table has a specific database type. This type is similar to the type of a variable in C#, except that a database type applies to the kind of value you can store in a table column. [Table 2.3](#) lists the SQL Server database types.

**Table 2.3: SQL SERVER DATABASE TYPES**

TYPE	DESCRIPTION
<code>bigint</code>	Integer value from $-2^{63}$ (-9,223,372,036,854,775,808) to $2^{63}-1$ (9,223,372,036,854,775,807).
<code>int</code>	Integer value from $-2^{31}$ (-2,147,483,648) to $2^{31}-1$ (2,147,483,647).
<code>smallint</code>	Integer value from $2^{15}$ (-32,768) to $2^{15}-1$ (32,767).
<code>tinyint</code>	Integer value from 0 to 255.
<code>bit</code>	Integer value with either a 1 or 0 value.
<code>decimal</code>	Fixed precision and scale numeric value from $-10^{38}+1$ to $10^{38}-1$ .
<code>numeric</code>	Same as <code>decimal</code> .
<code>money</code>	Monetary data value from $-2^{63}$ (-922,337,203,685,477.5808) to $2^{63}-1$ (922,337,203,685,477.5807), with an accuracy to one ten-thousandth of a monetary unit.
<code>smallmoney</code>	Monetary data value from -214,748.3648 to 214,748.3647, with an accuracy to one ten-thousandth of a monetary unit.
<code>float</code>	Floating-point value from $-1.79E+308$ to $1.79E+308$ .
<code>real</code>	Floating-point value from $-3.40E + 38$ to $3.40E + 38$ .
<code>datetime</code>	Date and time value from January 1, 1753, to December 31, 9999, with an accuracy of three-hundredths of a second (3.33 milliseconds).
<code>smalldatetime</code>	Date and time value from January 1, 1900 to June 6, 2079 with an accuracy of one minute.

<code>char</code>	Fixed-length non-Unicode characters with a maximum length of 8,000 characters.
<code>varchar</code>	Variable-length non-Unicode characters with a maximum of 8,000 characters.
<code>text</code>	Variable-length non-Unicode characters with a maximum length of $2^{31}-1$ (2,147,483,647) characters.
<code>nchar</code>	Fixed-length Unicode characters with a maximum length of 4,000 characters.
<code>nvarchar</code>	Variable-length Unicode characters with a maximum length of 4,000 characters.
<code>ntext</code>	Variable-length Unicode characters with a maximum length of $2^{30}-1$ (1,073,741,823) characters.
<code>binary</code>	Fixed-length binary data with a maximum length of 8,000 bytes.
<code>varbinary</code>	Variable-length binary data with a maximum length of 8,000 bytes.
<code>image</code>	Variable-length binary data with a maximum length of $2^{31}-1$ (2,147,483,647) bytes.
<code>cursor</code>	Reference to a cursor, which is a set of rows.
<code>sql_variant</code>	Can store values of various SQL Server types except <code>text</code> , <code>ntext</code> , <code>timestamp</code> , and <code>sql_variant</code> .
<code>table</code>	Stores a set of rows.
<code>timestamp</code>	Unique binary number that is updated every time you modify a row. You can only define one <code>timestamp</code> column in a table.
<code>uniqueidentifier</code>	Globally unique identifier (GUID).

Okay, enough theory! Let's take a closer look at the `Customers`, `Orders`, `Order Details`, and `Products` tables.

## The *Customers* Table

The `Customers` table contains rows that store the details of a company that might place orders with the Northwind Company. [Figure 2.14](#) shows some of the rows and columns stored in the `Customers` table.



The screenshot shows a Microsoft SQL Server Enterprise Manager window displaying the data from the 'Customers' table. The table has 13 columns: CustomerID, CompanyName, ContactName, ContactTitle, Address, City, Region, Phone, Fax, Email, SupportRepID, and Notes. The data consists of 42 rows, each representing a different company. Some key entries include:

CustomerID	CompanyName	ContactName	ContactTitle	Address	City
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obernstr. 57	Berlin
ANATR	Ana Trujillo Emparedados y helados	Aña Trujillo	Owner	Ayda de la Constitución 2222	México D.F.
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Manzana 2512	México D.F.
APOLI	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvägen 8	Luleå
BLAUS	Blaauw See Delicatesse	Karina Blauw	Sales Representative	Festvärk 57	Mannheim
BLONP	Blondel pâtisseries	Fridérik Céleste	Marketing Manager	24, place Kléber	Strasbourg
BOLID	Bólido Comidas preparadas	Martin Sommer	Owner	C/ Araquil, 67	Madrid
BONAP	Bon app	Laurence Lebihan	Owner	12, rue des Bouchers	Marseille
BOTTR	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsamakos Blvd.	Tsawassen
ESBRY	Els Beverage	Victoria Ashworth	Sales Representative	Fauntroy Court	London
CACTU	Cactus Comidas para llevar	Patricia Sedlacek	Sales Agent	Centro 333	Buenos Aires
CENITC	Centro comercial Holtxaurrea	Francisco Chang	Marketing Manager	Serrras de Granada 9990	México D.F.
CHOPS	Chop-suey Chinese	Yang Wang	Owner	Hausplatz, 29	Bern
COMAR	Comerio Minero	Pedro Alonso	Sales Associate	Ave. dos Lusíadas, 23	Sao Paulo
COMSH	Consolidated Holdings	Elizabeth Keyley	Sales Representative	Berkeley Gardens 12, Brewery	London
DRACD	Drachenblut Delikatessen	Ivan Orlitzky	Order Administrator	Wibergweg 22	Aachen
DURUN	Du monde entier	Jeanne Labrune	Owner	67, rue des Cinquante Otages	Nantes
EASTC	Eastern Connection	Ann Devitt	Sales Agent	35 King George	London
ERNSH	Ernst Handel	Roland Mendel	Sales Manager	Kirchgasse 6	Graz
FAMBA	Familia Argumedo	Aria Cruz	Marketing Assistant	Rua Orós, 92	Sao Paulo
FISSA	FISSA Fabrica Inter. Sánchez S.A.	Diego Roel	Accounting Manager	C/ Monseñor, 86	Madrid
FOUGS	Folies gourmandes	Marie Renault	Assistant, Sales Agent	184, chaussee de Tournai	Lille
FOUKO	Folk och fä HB	Maria Larsson	Owner	Allergatan 24	Bräcke
FRANK	Frankenversand	Peter Prankler	Marketing Manager	Berliner Platz 43	München
FRANR	France restaurant	Carine Schnitt	Marketing Manager	54, rue Royale	Nantes
FRANS	Franchi S.p.A.	Pietro Acciari	Sales Representative	Via Monte Bianco 24	Torino
FURIB	Furia Brasileira e Frutos do Mar	Lino Rodriguez	Sales Manager	Jardim das rosas n. 32	Lisboa
GALED	Galeria del gastrónomo	Eduardo Saaevedra	Marketing Manager	Rambla de Cataluña, 23	Barcelona
GODOS	Godis Coca Típics	Josep Pedro Freyre	Sales Manager	C/ Romero, 33	Sevilla
GOURL	Gourmet Lanchonetes	André Ponsca	Sales Associate	Av. Brasil, 442	Campinas

**Figure 2.14:** Rows from the `Customers` table

As you can see, the first row displayed is for a customer with the name `Alfreds Futterkiste`; this name is stored in the `CompanyName` column of the `Customers` table.

The `CustomerID` for the first row is `ALFKI`, and as you can see, the `CustomerID` is unique for each row. As mentioned earlier, the primary key for the `Customers` table is the `CustomerID` column. If you tried to add a row with a primary key already used by a row, then the database would reject your new row. For example, if you tried to add a row to the `Customers` table with a `CompanyID` of `ALFKI`, then that row would be rejected because `ALFKI` is already used by the first row in the table.

**Tip** You can view the rows from a table yourself by selecting the table in Enterprise Manager, clicking the right mouse button, and selecting Open Table > Return all rows. You'll learn more about viewing rows from tables later in the "Building Queries" section.

## Definition of the *Customers* Table

[Table 2.4](#) shows the definition for the columns of the `Customers` table. This table shows the column name, database type, length, and whether the column allows null values.

**Table 2.4: DEFINITION FOR THE COLUMNS OF THE *Customers* TABLE**

COLUMN NAME	DATABASE TYPE	LENGTH	Allows Null Values?
CustomerID	nchar	5	No
CompanyName	nvarchar	40	No
ContactName	nvarchar	30	Yes
ContactTitle	nvarchar	30	Yes
Address	nvarchar	60	Yes
City	nvarchar	15	Yes
Region	nvarchar	15	Yes
PostalCode	nvarchar	10	Yes
Country	nvarchar	15	Yes
Phone	nvarchar	24	Yes
Fax	nvarchar	24	Yes

In the [next section](#), you'll learn about the `Orders` table.

## The *Orders* Table

The `Orders` table contains rows that store the orders placed by customer. [Figure 2.15](#) shows some of the rows and columns stored in the `Orders` table.

The screenshot shows a database table named 'Orders' from the 'Northwind' database. The table has 13 columns: OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, and ShipName. The data consists of 42 rows, each representing an order placed by a customer through an employee. The first few rows show orders for customer ALFKI, followed by others like ANTON, ANTON, and BERGS.

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName
10453	ALFKI	6	8/25/1997	9/22/1997	9/21/1997	1	29.46	Alfreds Futterkiste
10456	ALFKI	4	9/3/1997	9/30/1997	9/3/1997	2	68.02	Alfreds Futterkiste
10703	ALFKI	4	10/3/1997	10/24/1997	10/21/1997	1	23.94	Alfreds Futterkiste
10835	ALFKI	1	1/15/1998	2/12/1998	1/23/1998	3	69.53	Alfreds Futterkiste
10952	ALFKI	1	2/1/1998	2/27/1998	2/24/1998	1	40.42	Alfreds Futterkiste
13311	ALFKI	3	4/9/1998	5/7/1998	4/13/1998	1	1.21	Alfreds Futterkiste
10928	ANATR	4	3/8/1998	3/15/1998	3/13/1998	3	39.92	Ana Trujillo Emparedados y helados
10759	ANATR	3	11/20/1997	12/26/1997	12/22/1997	3	11.99	Ana Trujillo Emparedados y helados
10625	ANATR	3	8/8/1997	9/5/1997	8/14/1997	1	43.9	Ana Trujillo Emparedados y helados
10308	ANATR	7	9/13/1998	10/16/1998	9/24/1998	3	1.61	Ana Trujillo Emparedados y helados
10365	ANTON	3	11/27/1997	12/25/1997	12/21/1997	2	22	Antonio Moreno Taqueria
10573	ANTON	7	6/18/1997	7/17/1997	6/20/1997	3	94.04	Antonio Moreno Taqueria
10507	ANTON	7	4/15/1997	5/13/1997	4/22/1997	1	47.45	Antonio Moreno Taqueria
10525	ANTON	4	5/13/1997	6/10/1997	5/13/1997	2	15.64	Antonio Moreno Taqueria
10462	ANTON	3	9/25/1997	10/23/1997	9/30/1997	2	36.13	Antonio Moreno Taqueria
10577	ANTON	1	10/20/1997	10/26/1997	10/26/1997	3	4.03	Antonio Moreno Taqueria
10596	ANTON	3	1/28/1998	2/25/1998	2/10/1998	2	58.43	Antonio Moreno Taqueria
10864	APARL	4	2/21/1998	3/2/1998	2/28/1998	2	3.04	Around the Horn
10768	APARL	5	1/20/1997	1/26/1998	1/21/1997	2	146.37	Around the Horn
10793	APARL	3	3/24/1998	4/21/1998	3/30/1998	3	4.52	Around the Horn
10920	APARL	4	3/3/1998	3/21/1998	3/19/1998	2	29.61	Around the Horn
10953	APARL	9	3/1/1998	3/20/1998	3/25/1998	2	23.72	Around the Horn
13196	APARL	9	4/3/1998	5/8/1998	4/13/1998	2	33.8	Around the Horn
10741	APARL	4	11/14/1997	11/20/1997	11/19/1997	3	10.96	Around the Horn
10743	APARL	1	11/17/1997	12/15/1997	11/21/1997	2	23.72	Around the Horn
10767	APARL	6	10/16/1997	10/30/1997	10/23/1997	3	23.74	Around the Horn
10558	APARL	1	6/6/1997	7/2/1997	6/10/1997	2	72.97	Around the Horn
10355	APARL	6	1/15/1998	1/21/1998	1/20/1998	1	41.95	Around the Horn
10383	APARL	8	3/24/1998	4/1/1998	3/30/1998	3	34.24	Around the Horn
10453	APARL	1	2/21/1997	3/2/1997	2/26/1997	2	25.36	Around the Horn
10444	BERGS	3	2/12/1997	3/12/1997	2/23/1997	3	3.5	Berglunds snabbköp

**Figure 2.15:** Rows from the Orders table

The primary key for the `Orders` table is the `OrderID` column, meaning that the value for this column must be unique for each row. If you look closely at the first six rows in the `Orders` table, you'll see that the `CustomerID` column is equal to `ALFKI`, which is the same as the `CustomerID` column for the first row in the `Customers` table shown earlier in [Figure 2.12](#).

You can now see how foreign keys relate information. The `CustomerID` column of the `Orders` table is a foreign key that references the `CustomerID` column of the `Customers` table. In this example, the `Orders` table is the child table, and the `Customers` table is the parent table. You can think of the foreign key as a pointer from the `Orders` table to the `Customers` table. [Table 2.5](#) shows the definition for the columns of the `Orders` table.

**Table 2.5: DEFINITION FOR THE COLUMNS OF THE Orders TABLE**

COLUMN NAME	DATABASE TYPE	LENGTH	Allows Null Values?
OrderID	int	4	No
CustomerID	nchar	5	Yes
EmployeeID	int	4	Yes
OrderDate	datetime	8	Yes
RequiredDate	datetime	8	Yes
ShippedDate	datetime	8	Yes
ShipVia	int	4	Yes
Freight	money	8	Yes
ShipName	nvarchar	40	Yes
ShipAddress	nvarchar	60	Yes
ShipCity	nvarchar	15	Yes
ShipRegion	nvarchar	15	Yes
ShipPostalCode	nvarchar	10	Yes
ShipCountry	nvarchar	15	Yes

In the [next section](#), you'll learn about the `Order Details` table.

## The Order Details Table

The Order Details table contains rows that store the details of each order. In [Figure 2.16](#), I've restricted the rows retrieved from the Order Details table to those where the OrderID column is equal to 10643 (this is the same as the OrderID column for the first row in the Orders table shown earlier in [Figure 2.15](#)).

OrderID	ProductID	UnitPrice	Quantity	Discount
10643	28	45.6	15	0.25
10643	39	18	21	0.25
10643	46	12	2	0.25

**Figure 2.16:** Restricted rows from the Order Details table

The primary key for the Order Details table is the combination of the OrderID and CustomerID columns, meaning that the combination of the values in these two columns must be unique for each row.

Also, the OrderID column of the Order Details table is a foreign key that references the OrderID column of the Orders table. The ProductID column of the Order Details table is a foreign key that references the ProductID column of the Products table. [Table 2.6](#) shows the definition for the columns of the Order Details table. You'll learn about the Products table next.

**Table 2.6: DEFINITION FOR THE COLUMNS OF THE Order Details TABLE**

COLUMN NAME	DATABASE TYPE	LENGTH	Allows Null Values?
OrderID	int	4	Yes
ProductID	int	4	Yes
UnitPrice	money	8	Yes
Quantity	smallint	2	Yes
Discount	real	4	Yes

## The Products Table

The Products table contains rows that store the details of each product sold by the Northwind Company. In [Figure 2.17](#), I've restricted the rows retrieved from the Products table to those where the ProductID column is equal to 22, 39, and 46 (these are the same as the values for the ProductID column for the rows in the Order Details table shown earlier in [Figure 2.16](#)).

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel
22	Gulács körtekrém	9	5	24 - 500 g pélcs.	21	104	0	25
39	Chutney verte	18	1	750 cc per bottle	18	69	0	5
46	Spagetti	21	8	4 - 450 g glasses	12	95	0	0

**Figure 2.17:** Restricted rows from the Products table

The primary key for the Products table is the ProductID column. The CategoryID column of the Products table is a foreign key that references the CategoryID column of the Categories table. The Categories table contains the various categories of products.

The `SupplierID` column of the `Products` table is a foreign key that references the `SupplierID` column of the `Suppliers` table. The `Suppliers` table contains the suppliers of products to the Northwind Company. [Table 2.7](#) shows the definition for the columns of the `Products` table.

**Table 2.7: DEFINITION FOR THE COLUMNS OF THE Products TABLE**

COLUMN NAME	DATABASE TYPE	LENGTH	Allows Null Values?
<code>ProductID</code>	<code>int</code>	4	No
<code>ProductName</code>	<code>nvarchar</code>	40	No
<code>SupplierID</code>	<code>int</code>	4	Yes
<code>CategoryID</code>	<code>int</code>	4	Yes
<code>QuantityPerUnit</code>	<code>nvarchar</code>	20	Yes
<code>UnitPrice</code>	<code>money</code>	8	Yes
<code>UnitsInStock</code>	<code>smallint</code>	2	Yes
<code>UnitsOnOrder</code>	<code>smallint</code>	2	Yes
<code>ReorderLevel</code>	<code>smallint</code>	2	Yes
<code>Discontinued</code>	<code>bit</code>	1	Yes

In the [next section](#), you'll learn how to build queries to retrieve rows from tables.

**Team LiB**

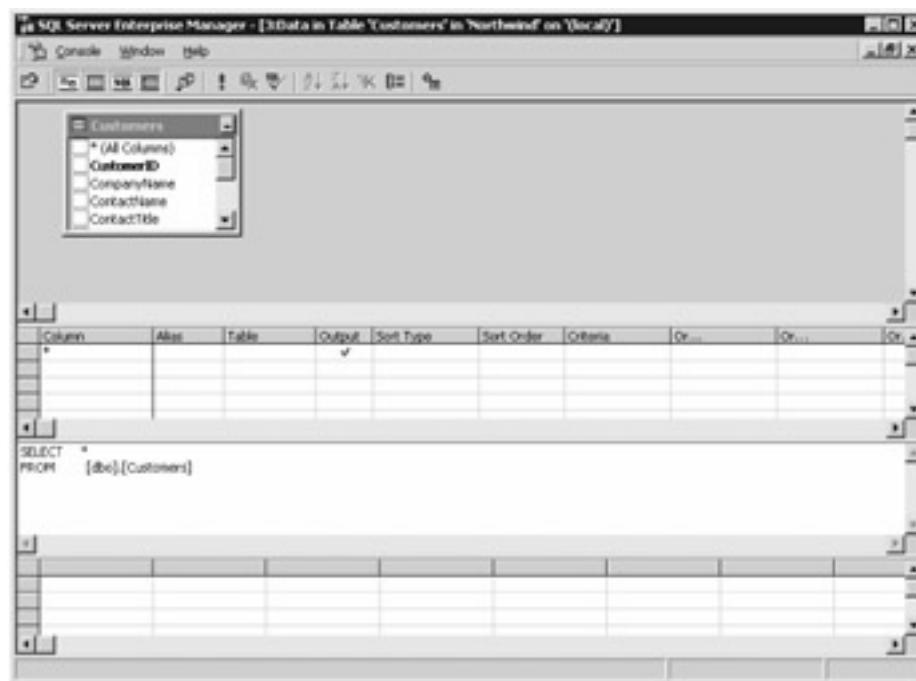
[◀ PREVIOUS](#) [NEXT ▶](#)

## Building Queries Using Enterprise Manager

You can build your own queries to examine rows in tables using Enterprise Manager. In this section, you'll learn how to build and run a query to view the orders placed by the customer with a `CustomerID` of ALFKI, along with the order details and products for the order with an `OrderID` of 10643. Specifically, you'll be selecting the following columns:

- The `CustomerID` and `CompanyName` columns from the `Customers` table
- The `OrderID` and `OrderDate` columns from the `Orders` table
- The `ProductID` and `Quantity` columns from the `Order Details` table

To start building the query, select the `Customers` table in Enterprise Manager from the Tables node of the Databases folder for the Northwind database. Click the right mouse button and select Open Table > Query. This opens the query builder, as shown in [Figure 2.18](#).



**Figure 2.18:** The query builder

The upper pane is called the Diagram Pane, and it shows the tables that are used in the query. As you can see, the `Customers` table is initially shown in the Diagram Pane. The pane below is called the Grid Pane, and it shows the details for the columns and rows to be retrieved from the tables. Initially, all rows are to be retrieved from the `Customers` table, as indicated by the asterisk (\*) in the Grid Pane. Below the Grid Pane is the SQL Pane, and it shows the SQL statement for the query.

**Note** SQL is a text-based language for accessing a database, and you'll learn all about SQL in the [next chapter](#). For now, you can click the SQL button on the toolbar to hide the SQL Pane-unless you want to view the SQL statement that is constructed by the query builder.

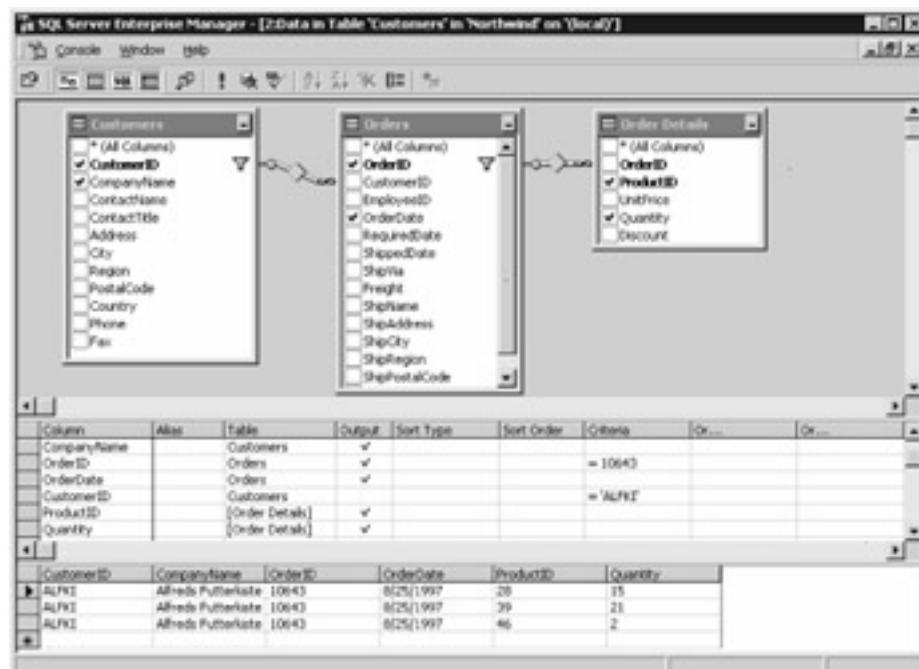
Below the SQL Pane is the Results Pane, which shows any rows retrieved by the query. This is initially empty because no query has yet been run. Use the following steps to build the query:

1. Remove the asterisk (\*) from the Grid Pane by clicking the right mouse button on the box on the left of the row containing the asterisk and selecting Delete. This stops all columns from being retrieved from the `Customers` table.
2. Click the right mouse button in the Diagram Pane, and select Add Table. Add the `Orders` and `Order Details` tables so that you can query these tables. You can also click the Add table button on the toolbar to add tables. You'll notice that after you add the tables, they appear in the Diagram Pane along with lines that connect the parent and child tables through the foreign key. For example, the `Customers` and `Orders` tables are connected through the `CustomerID` column. Similarly, the `Orders` and

Order Details tables are connected through the OrderID column.

3. Select the CustomerID and CompanyName columns from the Customers table by selecting the check boxes to the left of the column names in the Diagram Pane.
4. Select the OrderID and OrderDate columns from the Orders table.
5. Select the ProductID and Quantity columns from the Order Details table.
6. In the Grid Pane, set the Criteria for the CustomerID column to = 'ALFKI'. This causes the query to retrieve only the rows from the Customers table where the CustomerID column is equal to ALFKI.
7. In the Grid Pane, set the Criteria for the OrderID column to = 10643. This causes the query to retrieve only the rows from the Orders table where the OrderID column is equal to 10643.
8. Run the query by clicking the Run button on the toolbar.

[Figure 2.19](#) shows the final result of building and running the query.



**Figure 2.19:** Building and running a query

As you'll see in the [next chapter](#), you can also build and run queries using Visual Studio .NET. In the [next section](#), you'll learn how to create a table using Enterprise Manager.

## Creating a Table

You can use Enterprise Manager to add a table to a database. In this section, you'll add a table to the Northwind database to store the details of a person. This table will be called **Persons**, and will contain the columns shown in [Table 2.8](#).

**Table 2.8: DEFINITION FOR THE COLUMNS OF THE Persons TABLE**

COLUMN NAME	DATABASE TYPE	LENGTH	ALLOWS NULL VALUES?
PersonID	int	4	No
FirstName	nvarchar	15	No
LastName	nvarchar	15	No
DateOfBirth	datetime	8	Yes
Address	nvarchar	50	Yes
EmployerID	nchar	5	No

To create a table in the Northwind database, you select the Tables node of the Northwind database in Enterprise Manager and select Action ➤ New Table. You'll then see the table designer. Add the columns as shown in [Table 2.8](#) to the table, as shown in [Figure 2.20](#).



**Figure 2.20:** Adding a new table

**Note** The length of some of the data types is fixed. For example, the *int* type always uses 4 bytes of storage space, so you can't change the length of an *int* column from 4. Similarly, the *datetime* type always uses 8 bytes of storage space. You can change the length of *nchar* and *nvarchar* columns because those types are designed to store variable-length data.

Click the Save button on the toolbar to save the table. In the Choose Name dialog, enter **Persons** as the name, and click OK to save your table, as shown in [Figure 2.21](#).



**Figure 2.21:** Entering the name of the table

**Note** Once you've saved your table, you can return to the table designer at any time by selecting the table in the Tables node of Enterprise Manager, right-clicking the table, and selecting Design Table.

In the rest of this chapter, you'll learn how to:

- Get additional information about the columns in a table using the Columns tab.
- Set the primary key of a table.
- Set the permissions that allow access to the contents of a table.
- Create a relationship between tables.
- Create an index to allow faster access to the information in a table.
- Create a constraint to restrict values that may be stored in a column.

## The Columns Tab

In the area beneath the grid, you'll notice a tab named Columns. The Columns tab contains additional information about the currently selected column in the grid, and [Figure 2.20](#), shown earlier, shows the information on the PersonID column. As you change your selected column, the information in the Columns tab will change.

You can enter an optional description for a column in the Description field of the Columns tab. The Default Value field allows you to supply an initial value when a new row is added to the table; you can of course supply your own value to a column that will override the default value.

The Precision field shows the maximum number of digits that may be used to store a number, including those that might be stored to the right of a decimal point. The Scale field shows the maximum number of digits to the right of a decimal point. For example, the precision and scale of an `int` column are 10 and 0, meaning that an `int` column can store up to 10 digits, with no digits to the right of a decimal point—no digits to the right because an `int` is an integral number. The precision and scale for a `money` column are 19 and 4, meaning that a `money` column can store up to 19 digits, with up to four of those digits to the right of a decimal point.

The Identity field allows you specify whether SQL Server should automatically assign a value to a field. If you set the Identity field to Yes, then you can also specify values for the Identity Seed and Identity Increment fields. You use the Identity Seed field to set the initial value for the column, and you use the Identity Increment field to specify the increment for value. For example, if you set the Identity Seed to 1 and the Identity Increment to 1, then the first value for the column would be 1, the next would be 2, and so on. The `ProductID` column of the `Products` table is an example of a column that uses an identity to set its value.

The `IsRowGuid` field specifies whether a `uniqueidentifier` column is a globally unique identifier known as a GUID.

**Tip** SQL Server doesn't automatically supply a value for a GUID. If you want SQL Server to generate a GUID, you can use the SQL Server `NEWID()` function. The `NEWID()` function always returns a different value. You can then use the output from this function as the Default Value for your `uniqueidentifier` column. For example, you would set the Default Value field to `[NEWID()]`. You'll learn more about SQL Server functions in the [next chapter](#).

The Formula field allows you to set a formula that is used to assign a value to a column.

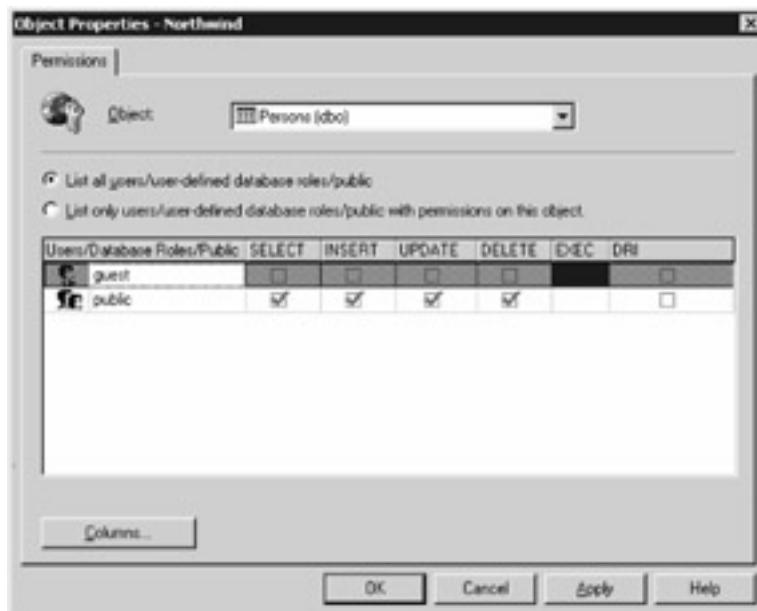
The Collation field specifies the rules that are used to sort and compare characters. You might need to set this when working with foreign languages. For further details, consult the SQL Server Books Online documentation.

## Setting the Primary Key

Next, you'll set the primary key for the `Persons` table to `PersonID`. To do this, click on the first row in the grid containing the `PersonID` column, and click the Set primary key button on the toolbar. Once you've done this, you'll see a small key icon to the left of `PersonID`.

## Setting the Permissions

To set the permissions for your table, click the Show permissions button on the toolbar of the table designer. Grant `SELECT`, `INSERT`, `UPDATE`, and `DELETE` permissions to the `public` role, as shown in [Figure 2.22](#). These permissions allow public users to retrieve, add, modify, and remove rows from the `Persons` table.



**Figure 2.22:** Setting the permissions

Click OK to continue.

## Creating the Relationship

You'll be creating a relationship between your `Persons` table and the `Customers` table. To view the relationships screen, click the Manage Relationships button on the toolbar of the table designer. Click New to start creating the relationship. Pick the `Customers` table as the primary key table and pick the `CustomerID` column from this table. Make sure `Persons` is selected as the foreign key table, and pick the `EmployerID` column from this table. [Figure 2.23](#) shows this. You'll notice that the relationship name is automatically set to `FK_Persons_Customers`.



**Figure 2.23:** Creating the relationship

The check boxes at the bottom the page are as follows:

- **Check existing data on creation** This applies your constraint to data that might already exist in the database when you add your relationship to the foreign key table.
- **Enforce relationship for replication** Replication allows you to copy information to a different database. When you enable Enforce relationship for replication, your constraint is applied to the foreign key table when that table is copied to a different database during replication.
- **Enforce relationship for INSERTs and UPDATEs** This applies your constraint to rows that are added, modified, or removed from the foreign key table. It also prevents a row in the primary key table from being deleted when there is a matching row in your foreign key table.
- **Cascade Update Related Fields** This causes SQL Server to automatically update the foreign key values of your relationship when the primary key value is modified.
- **Cascade Delete Related Fields** This causes SQL Server to automatically remove rows from the foreign key table whenever the referenced row in the primary key table is removed. Click Close to continue.

## Creating an Index

An index allows the database to quickly locate a row when you request retrieval of that row based on a particular column value. In this section, you'll create an index on the `Lastname` column of your `Persons` table.

To view the indexes for your `Persons` table, click the Manage Indexes/Keys button on the toolbar of the table designer. Click New to start creating a new index. Set the index name as `IX_LastName_Persons`, pick the `Lastname` column, and set the order as ascending. [Figure 2.24](#) shows this.



**Figure 2.24:** Creating an index

You won't change any of the other fields and check boxes when creating your index, but just so you know what they are, this is what the fields mean:

- **Index Filegroup** The index filegroup is the filegroup in which you want to store your index. A filegroup is made up of one or more physical files on a computer's hard disk. SQL Server uses filegroups to store the actual information that makes up a database.
- **Create UNIQUE** The Create UNIQUE option allows you to create a unique constraint or index for the selected database table. You indicate whether you are creating a unique constraint or index by selecting either the Constraint or Index radio button.
- **Ignore duplicate key** If you create a unique index, you can select this option to ignore duplicate key values.
- **Fill factor** Unless you are an advanced SQL Server user, you should leave the fill factor in the default setting. The smallest unit of storage in a SQL Server database is a *page*, which can hold up to 8,096 bytes of data. The data for tables and indexes are stored in pages. You can specify how full each index page can be by setting the fill factor. For example, if you set the fill factor to 60 percent, then the page will contain up to 60 percent data and 40 percent empty space. The amount of empty space on an index page is important because when an index page fills up, SQL Server must split the page to make room for new index data. By reducing the fill factor, therefore, you can increase the performance of your database because SQL Server won't have to split pages so often. Reducing the fill factor, however, also causes the index to take up more hard disk space because there will be more empty space in each page. If you don't specify a fill factor, then the database's default fill factor is used.
- **Pad Index** Unless you are an advanced SQL Server user, you shouldn't enable the Pad Index option. If you specify a fill factor of more than zero percent and you're creating a unique index through the Create UNIQUE option, then you can enable the Pad Index option. This informs SQL Server that it is to use the same percentage you specified in the fill factor field as the space to leave open on each leaf node of the binary tree that makes up the index. You can learn more about this option in the SQL Server Books Online documentation.
- **Create as CLUSTERED** You use the Create as CLUSTERED option to indicate that your index is clustered. A clustered index is one that contains the actual table rows, rather than pointers to the table rows. Clustered indexes allow faster retrieval of rows, but require more time when inserting new rows. You can learn more about this option in the SQL Server Books Online documentation.
- **Do not automatically recompute statistics** You typically shouldn't use this option as it might reduce performance. When you create an index, SQL Server automatically stores statistical information regarding the distribution of values in your indexed columns. SQL Server uses these statistics to estimate the cost of using the index for a query. You use the Do Not Automatically Recompute Statistics option to indicate that SQL Server should use previously created statistics, which means that the statistics are not necessarily up to date and could reduce performance. You can learn more about this option in the SQL Server Books Online documentation.

Click Close to continue.

## Creating a Constraint

A constraint allows you to define a limit on the value that may be stored in a column. In this section, you'll be creating a constraint on the DateOfBirth column of your Persons table. This constraint will ensure that you can place only dates between January 1, 1950, and December 31, 2050, in the DateOfBirth column.

To view the constraints for your Persons table, click the Manage Constraints button on the toolbar of the table designer. Click New to start creating a new constraint. Set the constraint expression as follows:

```
( [DateOfBirth] >= '1/1/1950' and [DateOfBirth] <= '12/31/2050' )
```

Set the constraint name as CK\_DateOfBirth\_Persons. [Figure 2.25](#) shows this.



**Figure 2.25:** Creating a constraint

You won't change any of the check boxes when creating your constraint, but just so you know what they are, this is what the fields mean:

- **Check existing data on creation** Use this option to ensure that data that currently exists in the table satisfies your constraint.
- **Enforce constraint for replication** Use this option to enforce your constraint when the table is copied to another database through replication.
- **Enforce constraint for INSERTs and UPDATEs** Use this option to enforce your constraint when rows are added or modified in the table.

Click Close to continue. Save the table and close the table designer.

**Team LiB**

◀ PREVIOUS ▶ NEXT ▶

## Summary

In this chapter, you learned the basics of databases and SQL Server. A database is an organized collection of information. A *relational database* is a collection of related information that has been organized into structures known as *tables*. Each table contains *rows* that are further organized into *columns*.

The system used to manage the information in the database is known as the *database management system*. In the case of an electronic database in a computer, the database management system is the software that manages the information in the computer's memory and files. One example of such software is SQL Server. You saw how to start a SQL Server database, and how to use Enterprise Manager to explore the Northwind database.

Typically, each table in a database has one or more columns that uniquely identify each row in the table. This column is known as the *primary key* for the table. Tables can be related to each other through *foreign keys*. You saw how to query the rows in a table and create a new table using Enterprise Manager.

In the [next chapter](#), you'll learn how to use the Structured Query Language.

**Team LiB**[◀ PREVIOUS](#)[NEXT ▶](#)

## Chapter 3: Introduction to Structured Query Language (SQL)

### Overview

In this chapter, you'll learn how to use Structured Query Language (SQL) to access a database, using two tools to enter and run queries: the Query Analyzer and Visual Studio .NET. This chapter shows you how to use the SQL Server Northwind database, which contains the information for the fictitious Northwind Company. You'll see how you use SQL to interact with the Northwind database to retrieve and manipulate information and to create, modify, and delete tables in that database.

Featured in this chapter:

- Using SQL
- [Accessing a database using Visual Studio .NET](#)

**Team LiB**[◀ PREVIOUS](#)[NEXT ▶](#)

## Using SQL

SQL (pronounced *sequel*) is the standard language for accessing relational databases. As you'll see in this chapter, SQL is easy to learn and use. With SQL, you tell the database what data you want to access, and the database software figures out exactly how to get that data.

There are many types of SQL statements, but the most commonly used types of SQL statements are these:

- Data Manipulation Language (DML) statements
- Data Definition Language (DDL) statements

DML statements allow you to retrieve, add, modify, and delete rows stored in the database. DDL statements allow you to create database structures such as tables.

Before you learn the basics of DML statements, you need to know how you can enter and run SQL statements. You can enter and run SQL statements against a SQL Server database using the Query Analyzer tool, and you'll learn about this next.

**Note** As you'll see later in the "[Accessing a Database Using Visual Studio .NET](#)" section, you can also use Visual Studio .NET to create SQL statements. Visual Studio .NET enables you to create SQL statements visually, as well as entering them manually.

## Using Query Analyzer

You use Query Analyzer to enter and run SQL statements. You start Query Analyzer by selecting Start > Microsoft SQL Server > Query Analyzer. In the following sections, you'll learn how to connect to a SQL server instance, enter and run a SQL statement, save a SQL statement, and load one.

### Connecting to a SQL Server Instance

When you start Query Analyzer, the first thing it displays is the Connect to SQL Server dialog box, as shown in [Figure 3.1](#). In the SQL Server field, you enter the name of the SQL Server instance to which you want to connect. You can click the drop-down list and select an instance of SQL Server, or you can click the ellipsis button (three dots ...) to the right of the drop-down list to display a list of SQL Server instances running on your network.



**Figure 3.1:** Connecting to a SQL Server database

If you select the Windows authentication radio button, then SQL Server will use the Windows 2000/NT user information to validate your request to connect to SQL Server. If you select the SQL Server authentication radio button, then you will need to enter a login name and password.

In [Figure 3.1](#), I've entered localhost in the SQL Server field; this corresponds to the instance of SQL Server installed on the local computer.

I've also selected the SQL Server authentication radio button, and entered **sa** in the Login Name field and **sa** in the Password field (this is the password I used when installing SQL Server). These details are then used to connect to SQL Server. If you have an instance of SQL Server running on your local computer or on your network, you may enter the relevant details and click the OK button to connect to SQL Server.

Now that you've seen how to connect to the database, let's take a look at how you enter and run a SQL statement.

## Entering and Running a SQL Statement

Once you've connected to SQL Server using Query Analyzer, you can use the Object Browser to view the parts of a database, and you enter and run SQL statements using a Query window. [Figure 3.2](#) shows the Object Browser and an example Query window, along with the results of retrieving the CustomerID and CompanyName columns from the Customers table.



**Figure 3.2:** Viewing database items using the Object Browser and executing a `SELECT` statement using the Query window

As you can see from [Figure 3.2](#), you enter SQL statements into the top part of the Query window, and the results retrieved from the database are displayed in the bottom part. You specify the database to access with the `USE` statement, and you retrieve rows from the database using the `SELECT` statement.

**Tip** You can also specify the database to access by using the drop-down list on the toolbar.

If you want to follow along with this example, go ahead and enter the following `USE` statement into your Query window:

```
USE Northwind
```

This `USE` statement indicates that you want to use the Northwind database. Next, on a separate line, enter the following `SELECT` statement:

```
SELECT CustomerID, CompanyName FROM Customers;
```

This `SELECT` statement indicates that you want to retrieve the `CustomerID` and `CompanyName` columns from the `Customers` table.

**Note** `SELECT` and `FROM` are SQL keywords. Although SQL isn't case sensitive, I use uppercase when specifying SQL keywords and mixed case when specifying column and table names. You may terminate a SQL statement using a semicolon (;), although this isn't mandatory.

You can run the SQL statement entered in the Query window in five ways:

- Selecting Execute from the Query menu

- Clicking the Execute Query button (green triangle) on the toolbar
- Pressing the F5 key on the keyboard
- Pressing Ctrl+E on the keyboard
- Pressing Alt+X on the keyboard

Once you run the SQL statement, your statement is sent to the database for execution. The database runs your statement and sends results back. These results are then displayed in the bottom of your Query window.

## Saving and Loading a SQL Statement

You can save a SQL statement previously entered into Query Analyzer as a text file. Later, you can load and run the SQL statement saved in that file. You can save a SQL statement by

- Selecting Save or Save As from the File menu
- Clicking the Save Query/Result button (disk) on the toolbar
- Pressing Ctrl+S on the keyboard

When you do any of these, the Query Analyzer opens the Save Query dialog box. Let's say you save the file as `CustomerSelect.sql`. Once you've saved the file, you can open it by

- Selecting Open from the File menu
- Clicking the Load SQL Script button (open folder) on the toolbar
- Pressing Ctrl+Shift+P on the keyboard

When you do any of these, the Query Analyzer opens the Open Query File dialog box. Let's say you open `CustomerSelect.sql`. Once you've opened a query file, you can run it using one of the techniques described earlier.

## Understanding Data Manipulation Language (DML) Statements

As mentioned earlier, DML statements enable you to retrieve, add, modify, and delete rows stored in database tables. There are four types of DML statements:

- **SELECT** Retrieves rows from one or more tables.
- **INSERT** Adds one or more new rows to a table.
- **UPDATE** Modifies one or more rows in a table.
- **DELETE** Removes one or more rows from a table.

You'll learn how to use these four statements in the following sections.

## Retrieving Rows From a Single Table

You use the **SELECT** statement to retrieve rows from tables. The **SELECT** statement has many forms, and the simplest version allows you to specify a list of columns and the table name. For example, the following **SELECT** statement retrieves the `CustomerID`, `CompanyName`, `ContactName`, and `Address` columns from the `Customers` table:

```
SELECT CustomerID, CompanyName, ContactName, Address
FROM Customers;
```

The columns to retrieve are specified after the **SELECT** keyword, and the table is specified after the **FROM** keyword.

If you want to retrieve all columns from a table, specify the asterisk character (\*) immediately after the **SELECT** keyword.

**Tip** To avoid retrieving more information than you need, rather than use \*, list only the columns you actually want.

For example, the following **SELECT** statement retrieves all the columns from the **Customers** table using \*:

```
SELECT *
FROM Customers;
```

[Figure 3.3](#) shows the results of this **SELECT** statement.

The screenshot shows the SQL Server Management Studio interface with a query window titled 'Query - [PRD03-LAP-Northwind].[master]'. The query is: 'SELECT \* FROM Customers;'. The results grid displays 13 rows of data from the Customers table, including columns like CustomerID, CompanyName, ContactTitle, Address, City, Region, and PostalCode. The data includes entries for customers like ALFKI, ANATR, and ANTON, among others.

	CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode
1	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Ober str. 57	Berlin	NW	12200
2	ANATR	Anne Trujillo Emp... ...er	Anne Trujillo	Owner	Ave. 5 de la C...	Mexico D.F.	NW	00001
3	ANTON	Antonio Moreno T...	Antonio Moreno	Owner	Barrio António	Mexico D.F.	NW	00001
4	AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Buffay Sq.	London	SW	SE1 8BP
5	HERB	Berglunds snabbköp	Christina Berglund	Sales Representative	Berguvsgatan 8	Luleå	N	991 02
6	BLAUS	Blauer See Delic...	Manuela Diaz	Sales Representative	Pfortenstr. 37	Mannheim	NW	68306
7	BLONP	Blondel pâtisseries	François BLONDEAU	Marketing Manager	24, place Kléber	Strasbourg	NW	67000
8	BOLID	Bolido Comidas p...	Martín Sommer	Owner	C/ Araquil, 47	Madrid	NW	28009
9	BONAP	Bon App'	Laurence Le... ...	Owner	12, rue des B...	Reuil-Malmaison	NW	92500
10	BOTTR	Bottom-Dodd Mat...	Elizabeth Abbott	Accounting Manager	29 Tasmania...	Tasmania	E	T12 9BA
11	BREVY	B'rs Beverages	Viktoria K...	Sales Representative	Frontline Closus London	NW	EC2 3RT	
12	CACTU	Cactus Comidas Fin...	Genesha Agent	Delivery Agent	Ortizla 33	Santiago ALM	NW	0010
13	CENTC	Centro comercial Mo...	Fernando Pérez	Marketing Manager	El Centro de M...	Mexico D.F.	NW	00001
14	CHINW	Chinatown Chocolates	Tang Wong	Owner	Brigata 12	Buenos Aires	S	3012
15	COMER	Comercio Wholes...	Pedro Adáns	Sales Associate	Av. dos Comis...	Sao Paulo	S	05412-043
16	CONBE	Consolidated BG...	Ellen Hartman	Sales Representative	Drumley Wood...	London	SW	921 0LT
17	GRACI	Graceland Delic...	Bernard Oberholzer	Order Administrator	Wellington 21	Aachen	NW	52046
18	FRANR	Frans依能	Jeanne Werner	Owner	67, rue des C...	Strasbourg	NW	67000
19	FAUST	Fausto Comida...	Ana Davids	Sales Agent	35 King George	London	SW	SE1 5EW
20	FRER	Frédéric Mendel	Holger Mandel	Sales Manager	Erlingsgate 4	Oslo	NW	0010

**Figure 3.3:** Using a **SELECT** statement to retrieve rows from the **Customers** table

To retrieve rows from a table containing a space in its name, you place that table name in square brackets. For example, the following **SELECT** statement retrieves rows from the **Order Details** table:

```
SELECT *
FROM [Order Details];
```

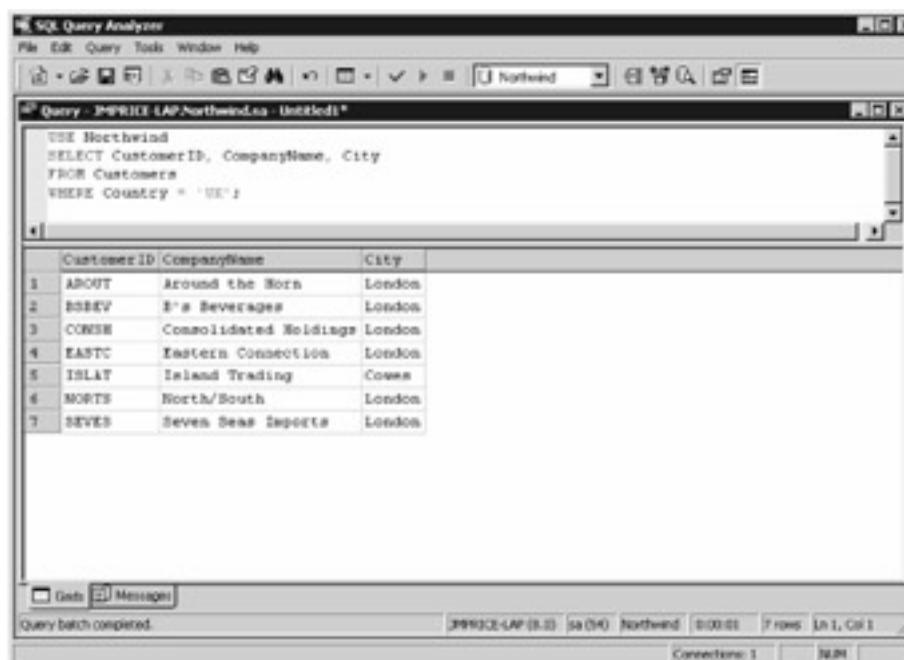
**Note** You can also use square brackets when you have a column with a name that contains a space.

## Restricting Retrieved Rows

You use the **WHERE** clause to restrict the rows retrieved by a **SELECT** statement. For example, the following **SELECT** statement uses a **WHERE** clause to restrict the rows retrieved from the **Customers** table to those where the **Country** column is equal to 'UK':

```
SELECT CustomerID, CompanyName, City
FROM Customers
WHERE Country = 'UK';
```

[Figure 3.4](#) shows the results of this **SELECT** statement.



**Figure 3.4:** Using a WHERE clause to restrict rows from the Customers table to those where Country is equal to 'UK'

The next SELECT statement uses a WHERE clause to restrict the row retrieved from the Products table to the one where ProductID is equal to 10:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductID = 10;
```

The equal operator (=) is not the only operator you can use in a WHERE clause. [Table 3.1](#) shows other mathematical operators you can use.

**Table 3.1: SQL MATHEMATICAL OPERATORS**

OPERATOR	DESCRIPTION
=	Equal
<> or !=	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

The following SELECT statement uses the less-than-or-equal operator (<=) to retrieve the rows from the Products table where the ProductID column is less than or equal to 10:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductID <= 10;
```

The next SELECT statement uses the not-equal operator (<>) to retrieve the rows from the Products table where the ProductID column is not equal to 10:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductID <> 10;
```

## Performing Pattern Matching

You use the `LIKE` operator in a `WHERE` clause to perform pattern matching. You specify one or more wildcard characters to use in your pattern matching string. [Table 3.2](#) lists the wildcard characters.

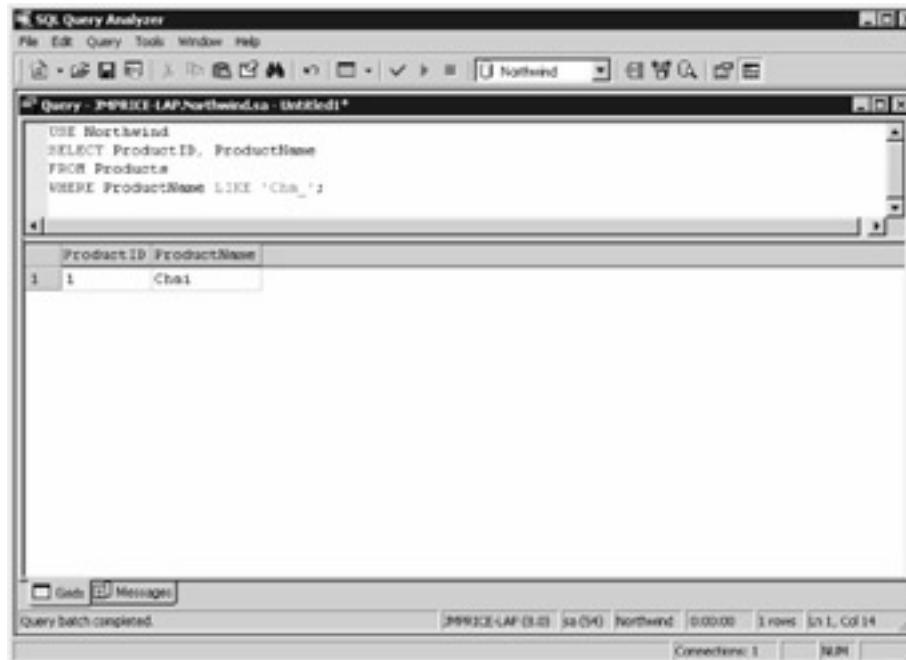
**Table 3.2: WILDCARD CHARACTERS**

CHARACTERS	DESCRIPTION
_	Matches any one character. For example, <code>J_y</code> matches <code>Joy</code> and <code>Jay</code> .
%	Matches any number of characters. For example, <code>%wind</code> matches <code>Northwind</code> and <code>Southwind</code> ; <code>%fire%</code> matches <code>starfire</code> , <code>firestarter</code> , and <code>fireman</code> .
[ ]	Matches any one character in the brackets. For example, <code>[sm]ay</code> matches <code>say</code> and <code>may</code> .
[ ^ ]	Matches any one character not in the brackets. For example, <code>[ ^a ]</code> matches any character except <code>a</code> .
[ - ]	Matches a range of characters. For example, <code>[a-c]bc</code> matches <code>abc</code> , <code>bbc</code> , and <code>cbc</code> .
#	Matches any one number. For example, <code>A#</code> matches <code>A1</code> through <code>A9</code> .

Let's take a look at some examples that use some of the wildcard characters shown in [Table 3.2](#). The following `SELECT` statement uses the `LIKE` operator to retrieve products where the `ProductName` column is like '`Cha_`':

```
SELECT ProductID, ProductName
FROM Products
WHERE ProductName LIKE 'Cha_';
```

[Figure 3.5](#) shows the results of this `SELECT` statement. `LIKE 'Cha_'` matches products with names that start with `Cha` and end with any one character.



**Figure 3.5:** Products where `ProductName` is like '`Cha_`'

The next `SELECT` statement uses the `LIKE` operator to retrieve products where the `ProductName` column is like '`Cha%`':

```
SELECT ProductID, ProductName
FROM Products
WHERE ProductName LIKE 'Cha%';
```

[Figure 3.6](#) shows the results of this `SELECT` statement. `LIKE 'Cha%'` matches products with names that start with `Cha` and end with any number of characters.

The screenshot shows the SQL Query Analyzer interface. The query window contains:

```
USE Northwind
SELECT ProductID, ProductName
FROM Products
WHERE ProductName LIKE 'Cha%'
```

The results grid shows the following data:

	ProductID	ProductName
1	1	Chai
2	2	Chang
3	39	Chartreuse verte

Below the results, the status bar shows "Query batch completed." and connection information.

**Figure 3.6:** Products where ProductName is like 'Cha%'

The next SELECT statement uses the LIKE operator to retrieve products where the ProductName column is like '[ABC]%' :

```
SELECT ProductID, ProductName
FROM Products
WHERE ProductName LIKE '[ABC]%' ;
```

[Figure 3.7](#) shows the results of this SELECT statement. LIKE '[ABC]%' matches products with a name that starts with any of the letters A, B, or C, and ends with any number of characters.

The screenshot shows the SQL Query Analyzer interface. The query window contains:

```
USE Northwind
SELECT ProductID, ProductName
FROM Products
WHERE ProductName LIKE '[ABC]%'
```

The results grid shows the following data:

	ProductID	ProductName
1	17	Alice Mutton
2	3	Aniseed Syrup
3	40	Boston Crab Meat
4	60	Camembert Pierrot
5	18	Carnarvon Tigers
6	1	Chai
7	2	Chang
8	39	Chartreuse verte
9	4	Chef Anton's Cajun Specialty G...
10	5	Chef Anton's Gumbo Mix
11	48	Chocolate
12	50	Côte de Blaye

Below the results, the status bar shows "Query batch completed." and connection information.

**Figure 3.7:** Products where ProductName is like '[ABC]%'

The next SELECT statement uses the LIKE operator to retrieve products where the ProductName column is like '[^ABC]%' :

```
SELECT ProductID, ProductName
FROM Products
WHERE ProductName LIKE '[^ABC]%' ;
```

## Using SQL

[Figure 3.8](#) shows the results of this SELECT statement. LIKE '[^ABC]%' matches products with names that don't start with any of the letters A, B, or C, and end with any number of characters.

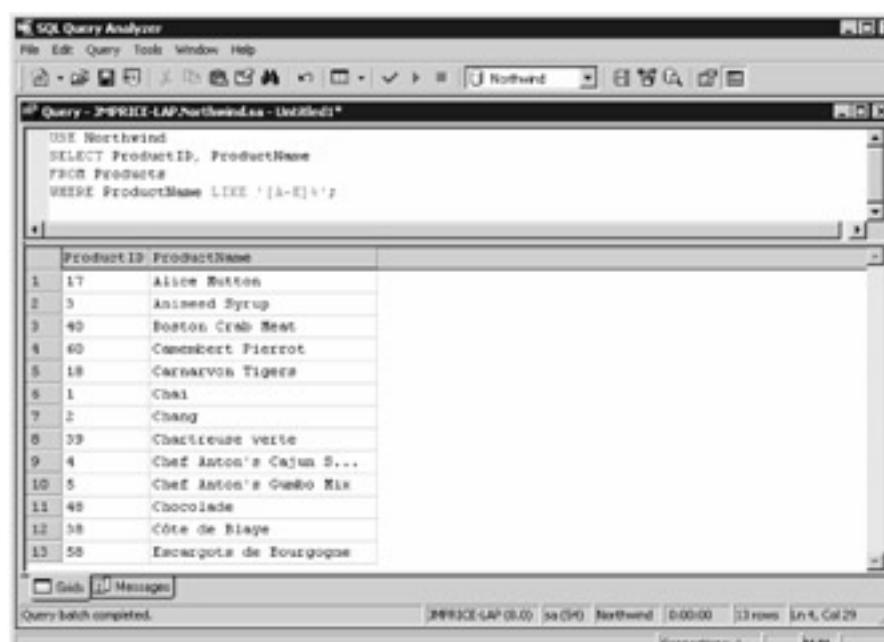


**Figure 3.8:** Products where ProductName is like '[^ABC]%'

The next `SELECT` statement uses the `LIKE` operator to retrieve products where the `ProductName` column is like '[A-E]%'.

```
SELECT ProductID, ProductName  
FROM Products  
WHERE ProductName LIKE '[A-E]%'
```

[Figure 3.9](#) shows the results of this SELECT statement. LIKE '[A-E]%' matches products with names that start with any of the letters A through E, and end with any number of characters.



**Figure 3.9:** Products where Product Name is like '[A-E]%

## Specifying a List of Values

You use the `IN` operator in a `WHERE` clause to retrieve rows with columns that contain values in a specified list. For example, the following

SELECT statement uses the `IN` operator to retrieve products with a `ProductID` of 1, 2, 5, 15, 20, 45, or 50:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductID IN (1, 2, 5, 15, 20, 45, 50);
```

Here's another example that displays the `OrderID` column from the `Orders` table for the rows where the `CustomerID` column is in the list retrieved by a *subquery*; the subquery retrieves the `CustomerID` column from the `Customers` table where the `CompanyName` is like '`Fu%`':

```
SELECT OrderID
FROM Orders
WHERE CustomerID IN (
    SELECT CustomerID
    FROM Customers
    WHERE CompanyName LIKE 'Fu%'
);
```

The results of the subquery are used in the outer query.

## Specifying a Range of Values

You use the `BETWEEN` operator in a `WHERE` clause to retrieve rows with columns that contain values in a specified range. For example, the following `SELECT` statement uses the `BETWEEN` operator to retrieve products with a `ProductID` between 1 and 12:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductID BETWEEN 1 AND 12;
```

Here's another example that displays the `OrderID` column for the rows from the `Orders` table where the `OrderDate` is between '`1996-07-04`' and '`1996-07-08`':

```
SELECT OrderID
FROM Orders
WHERE OrderDate BETWEEN '1996-07-04' AND '1996-07-08';
```

## Reversing the Meaning of an Operator

You use the `NOT` keyword with an operator in a `WHERE` clause to reverse the meaning of that operator. For example, the following `SELECT` statement uses the `NOT` keyword to reverse the meaning of the `BETWEEN` operator:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductID NOT BETWEEN 1 AND 12;
```

**Note** You can use the `NOT` keyword to reverse other operators, for example, `NOT LIKE`, `NOT IN`.

## Retrieving Rows with Columns Set to Null

Earlier, I mentioned that columns can contain null values. A null value is different from a blank string or zero: A null value represents a value that hasn't been set, or is unknown. You can use the `IS NULL` operator in a `WHERE` clause to determine if a column contains a null value. For example, the following `SELECT` statement uses the `IS NULL` operator to retrieve customers where the `Fax` column contains a null value:

```
SELECT CustomerID, CompanyName, Fax
FROM Customers
WHERE Fax IS NULL;
```

[Figure 3.10](#) shows the results of this `SELECT` statement.

The screenshot shows the SQL Query Analyzer interface. The query window contains the following code:

```
USE Northwind
SELECT CustomerID, CompanyName, Fax
FROM Customers
WHERE Fax IS NULL;
```

The results grid displays 13 rows of customer data, all showing NULL in the Fax column. The columns are CustomerID, CompanyName, and Fax.

CustomerID	CompanyName	Fax
1	ANTON	Antonio Moreno Taqueria
2	BEEV	B's Beverages
3	CHOPS	Chop-suey Chinese
4	COMER	Comercio Mineiro
5	FAMIL	Familia Arquibaldo
6	FOLIO	Folk och få KF
7	GODOS	Godos Cocina Tipica
8	GOURL	Gourmet Lanchonetes
9	GREAL	Great Lakes Food Market
10	ISLAT	Island Trading
11	JASCH	Test
12	KOENE	Koniglich Essen
13	LETSS	Let's Stop N Shop

**Figure 3.10:** Using the `IS NULL` operator to retrieve customers where `Fax` contains a null value

As you can see, null values are displayed as `NULL` in the Query Analyzer.

## Specifying Multiple Conditions

You can use the logical operators shown in [Table 3.3](#) to specify multiple conditions in a `WHERE` clause.

**Table 3.3: LOGICAL OPERATORS**

OPERATOR	DESCRIPTION
<code>a AND b</code>	Evaluates to true when <code>a</code> and <code>b</code> are both true
<code>a OR b</code>	Evaluates to true when either <code>a</code> or <code>b</code> are true
<code>NOT a</code>	Evaluates to true if <code>a</code> is false, and false if <code>a</code> is true

For example, the following `SELECT` statement uses the `AND` operator to retrieve products where the `UnitsInStock` column is less than 10 and the `ReorderLevel` column is less than or equal to 20:

```
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
WHERE UnitsInStock < 10
AND ReorderLevel <= 20;
```

[Figure 3.11](#) shows the results of this `SELECT` statement.

The screenshot shows the SQL Server Query Analyzer interface. A query window titled "Query - JBRUCE-LAP.Northwind.mdf - Untitled1" is open. The query is:

```
USE Northwind
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
WHERE UnitsInStock < 10
AND ReorderLevel <= 20;
```

The results grid displays 11 rows of product information. The columns are ProductID, ProductName, UnitsInStock, and ReorderLevel. The data includes items like Chef Anton's Gumbo Mix, Northwoods Cranberry Sauce, Alice Mutton, Sir Rodney's Sooces, Thuringer Rostbratwurst, Gorgonzola Telino, Rogede sild, Perth Pastries, Louisiana Hot Spiced Okra, Scottish Longbreads, and Longlife Tofu.

	ProductID	ProductName	UnitsInStock	ReorderLevel
1	5	Chef Anton's Gumbo Mix	0	0
2	8	Northwoods Cranberry Sauce	6	0
3	17	Alice Mutton	9	0
4	21	Sir Rodney's Sooces	3	5
5	29	Thuringer Rostbratwurst	0	0
6	31	Gorgonzola Telino	0	20
7	45	Rogede sild	5	15
8	53	Perth Pastries	0	0
9	66	Louisiana Hot Spiced Okra	4	20
10	68	Scottish Longbreads	6	15
11	74	Longlife Tofu	4	5

Below the results, the status bar shows "Query batch completed." and other connection details.

**Figure 3.11:** Using the AND operator to retrieve products where UnitsInStock is less than 10 and ReorderLevel is less than or equal to 20

In the next example, the SELECT statement uses the OR operator to retrieve products where either the UnitsInStock column is less than 10 or the ReorderLevel column is less than or equal to 20:

```
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
WHERE UnitsInStock < 10
OR ReorderLevel <= 20;
```

[Figure 3.12](#) shows the results of this SELECT statement.

The screenshot shows the SQL Server Query Analyzer interface. A query window titled "Query - JBRUCE-LAP.Northwind.mdf - Untitled1" is open. The query is:

```
USE Northwind
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
WHERE UnitsInStock < 10
OR ReorderLevel <= 20;
```

The results grid displays 15 rows of product information. The columns are ProductID, ProductName, UnitsInStock, and ReorderLevel. The data includes items like Chai, Chef Anton's Cajun Seasoning, Chef Anton's Gumbo Mix, Thalia Bob's Organic Dried Pears, Northwoods Cranberry Sauce, Kimchi Kobe Niku, Dhura, Queso Manchego La Pastor, Koski, Tofu, and Genen Shousyu.

	ProductID	ProductName	UnitsInStock	ReorderLevel
1	1	Chai	39	10
2	4	Chef Anton's Cajun Seasoning	51	0
3	5	Chef Anton's Gumbo Mix	0	0
4	7	Thalia Bob's Organic Dried Pears	15	10
5	8	Northwoods Cranberry Sauce	6	0
6	9	Kimchi Kobe Niku	29	0
7	10	Dhura	31	0
8	12	Queso Manchego La Pastor	66	0
9	13	Koski	24	5
10	14	Tofu	35	0
11	15	Genen Shousyu	39	5

Below the results, the status bar shows "Query batch completed." and other connection details.

**Figure 3.12:** Using the OR operator to retrieve products where either UnitsInStock is less than 10 or ReorderLevel is less than or equal to 20

The next SELECT statement uses the NOT operator to retrieve products where the UnitsInStock column is not less than 10:

```
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
```

```
WHERE NOT (UnitsInStock < 10);
```

## Sorting Rows

You can use the `ORDER BY` clause to sort rows retrieved from the database. You specify the column (or columns) to sort in the `ORDER BY` clause. By default, rows are sorted in ascending order. For example, the following `SELECT` statement orders the rows using the `ProductName` column:

```
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
ORDER BY ProductName;
```

[Figure 3.13](#) shows the results of this `SELECT` statement. As you can see, the rows are ordered in ascending order using the `ProductName` column.

The screenshot shows the SQL Query Analyzer interface. In the top pane, there is a query window with the following content:

```
USE Northwind
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
ORDER BY ProductName;
```

Below the query window is a results grid. The grid has four columns: ProductID, ProductName, UnitsInStock, and ReorderLevel. The data is as follows:

	ProductID	ProductName	UnitsInStock	ReorderLevel
1	17	Alice Mutton	0	0
2	3	Aniseed Syrup	10	25
3	40	Boston Crab Meat	123	30
4	60	Camembert Pierrot	19	0
5	18	Carcassonne Tigrae	42	0
6	1	Chai	39	10
7	2	Chang	17	25
8	39	Chartreuse Vecte	69	5
9	4	Chef Anton's Cajun Seasoning	52	0
10	5	Chef Anton's Gumbo Mix	0	0
11	48	Chocolade	15	25

At the bottom of the interface, there is a status bar showing "Query Batch completed." and other connection information.

**Figure 3.13:** Using the `ORDER BY` clause to order products by ascending `ProductName`

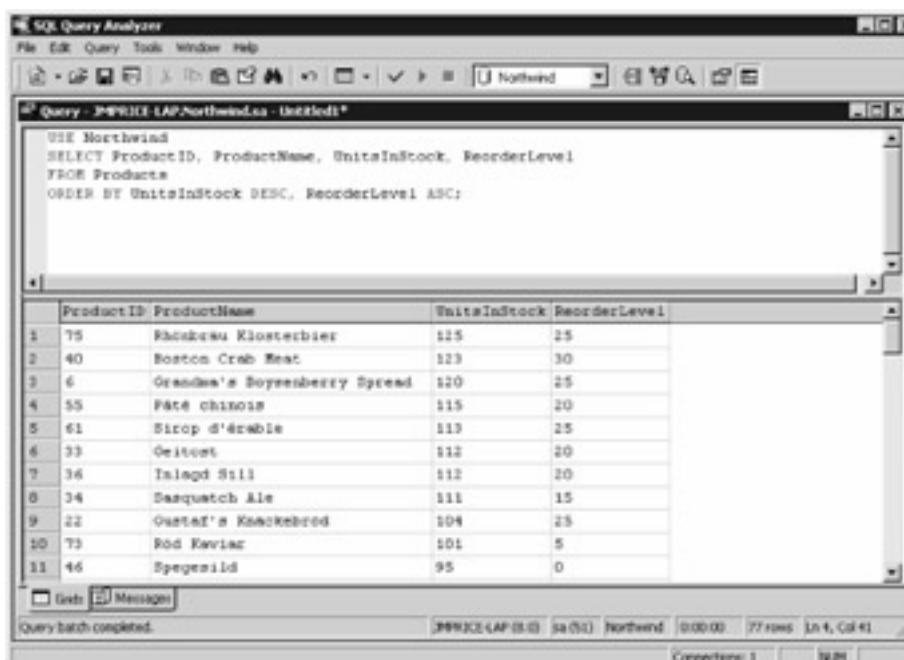
You can explicitly state the order for a column using the `ASC` or `DESC` keyword. `ASC` orders the columns in ascending order (smallest item first), and `DESC` orders the columns in descending order (largest item first). For example, the following `SELECT` statement orders the products in descending order using the `ProductName` column:

```
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
ORDER BY ProductName DESC;
```

You can specify multiple columns in an `ORDER BY` clause. For example, the following `SELECT` statement orders the rows using both the `UnitsInStock` and `ReorderLevel` columns:

```
SELECT ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
ORDER BY UnitsInStock DESC, ReorderLevel ASC;
```

[Figure 3.14](#) shows the results of this `SELECT` statement. As you can see, the rows are ordered by the `UnitsInStock` column first (in descending order), and then by the `ReorderLevel` column (in ascending order).



**Figure 3.14:** Using the DESC and ASC keywords to order products by descending UnitsInStock and ascending ReorderLevel

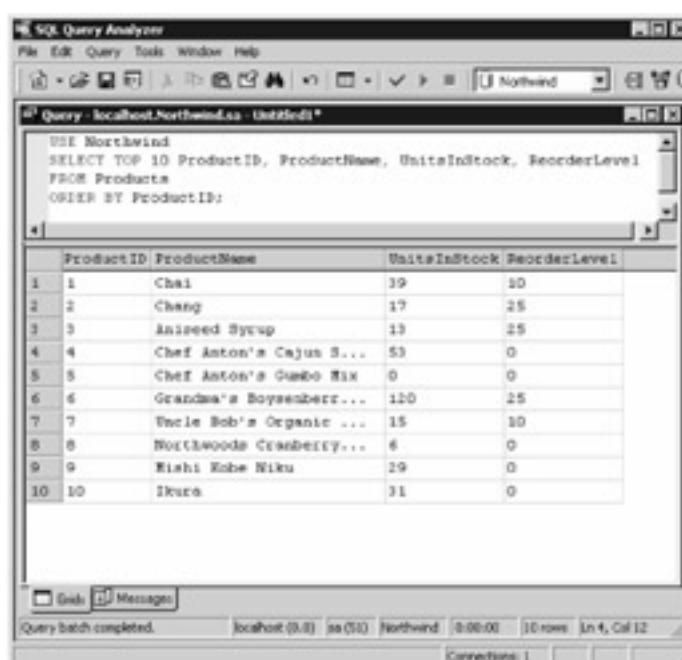
## Retrieving the Top N Rows

You use the TOP keyword to just retrieve the top *N* rows from a SELECT statement. For example, the following SELECT statement uses the TOP keyword to retrieve the top 10 rows from the Products table, ordered by the ProductID column:

```
SELECT TOP 10 ProductID, ProductName, UnitsInStock, ReorderLevel
FROM Products
ORDER BY ProductID;
```

**Note** I've also used the optional ORDER BY clause in this example SELECT statement to order the rows by the *ProductID* column.

[Figure 3.15](#) shows the results of this SELECT statement.



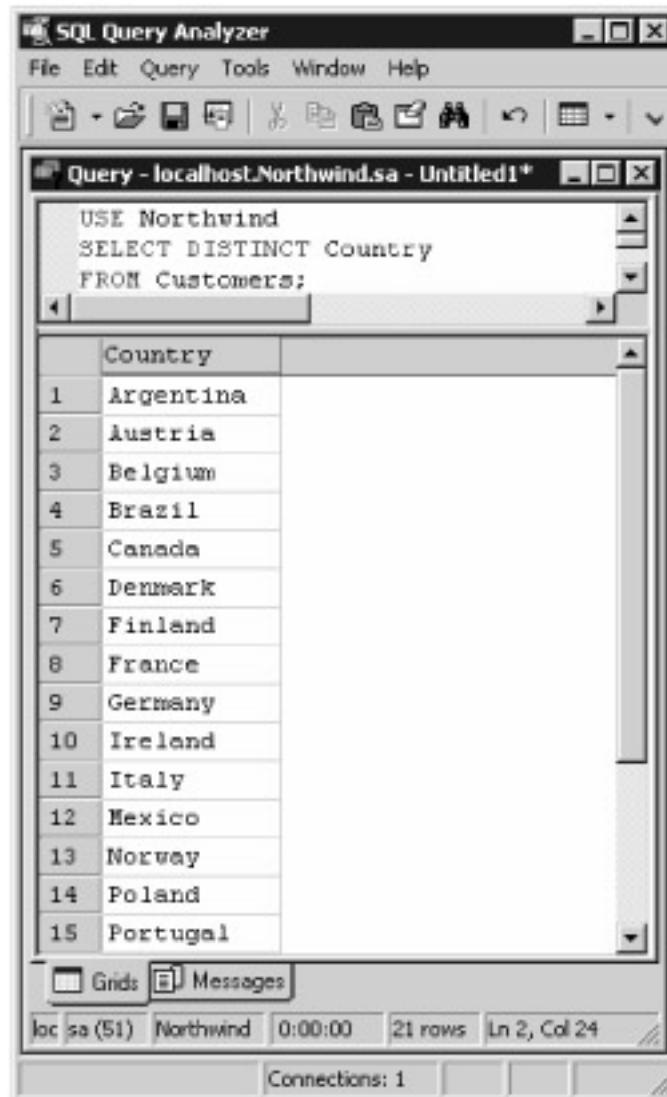
**Figure 3.15:** Using the TOP keyword to retrieve the top 10 products by ProductID

## Eliminating Duplicate Rows

You use the `DISTINCT` keyword to eliminate duplicate rows retrieved by a `SELECT` statement. For example, the following `SELECT` statement uses the `DISTINCT` keyword to retrieve the distinct `Country` column values from the `Customers` table:

```
SELECT DISTINCT Country
FROM Customers;
```

[Figure 3.16](#) shows the results of this `SELECT` statement.



**Figure 3.16:** Using the `DISTINCT` keyword to retrieve distinct `Country` column values

As you can see, the `SELECT` statement only displays `Country` column values that are unique: duplicate values are eliminated. If you didn't include the `DISTINCT` keyword, then all the `Country` column values would be displayed.

## Combining Retrieved Rows From `SELECT` Statements

You use the `UNION` operator to combine retrieved rows from `SELECT` statements into one set of rows. For example, the following `SELECT` statement uses the `UNION` operator to combine the retrieved rows from two `SELECT` statements that retrieve rows from the `Products` table; the first retrieves rows where the `ProductID` is less than or equal to 5, and the second retrieves rows where the `ProductName` starts with `Queso`:

```
(SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductID <= 5)
UNION
```

## Using SQL

```
(SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductName LIKE 'Queso%');
```

[Figure 3.17](#) shows the results of this statement.

The screenshot shows the SQL Server Query Analyzer interface. The query window contains the following code:

```
USE Northwind
(SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductID <= 5)
UNION
(SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
WHERE ProductName LIKE 'Queso%'));
```

Below the code, the results are displayed in a grid:

ProductID	ProductName	QuantityPerUnit	UnitPrice
1	Chai	10 boxes x 20 bags	18.0000
2	Chang	24 - 12 oz bottles	19.0000
3	Aniseed Syrup	12 - 550 ml bottles	10.0000
4	Chef Anton's Gourmet ...	48 - 6 oz jars	22.0000
5	Chef Anton's Gumbo Mix	36 boxes	21.3500
11	Queso Cabrales	1 kg pkg.	21.0000
12	Queso Manchego La Pa...	10 - 500 g packages	16.0000

**Figure 3.17:** Using the UNION operator to combine retrieved rows from two SELECT statements

## Dividing Retrieved Rows into Blocks

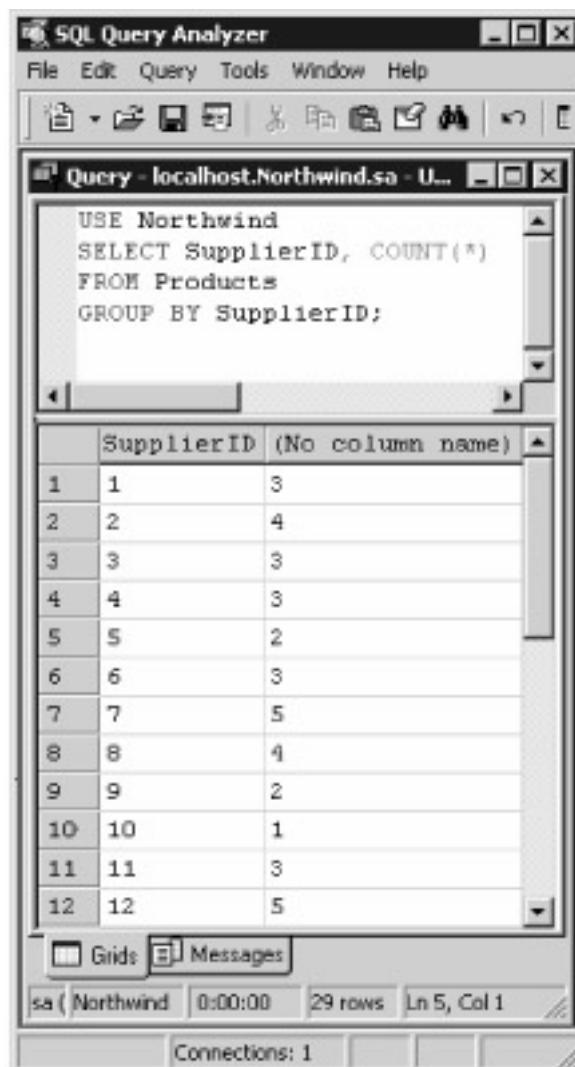
You use the `GROUP BY` clause to divide retrieved rows into *blocks*. You can think of a block as a group of rows that have been condensed into one row. For example, let's say you grouped the `SupplierID` column of the rows from the `Products` table. You would get one row for every row that had the same `SupplierID` column value. The following `SELECT` statement uses the `GROUP BY` clause to divide the `SupplierID` column values into blocks:

```
SELECT SupplierID
FROM Products
GROUP BY SupplierID;
```

This `SELECT` statement displays one row for each group of rows that have the same `SupplierID` column value. You can get the number of rows in each block using the `COUNT( )` function. `COUNT( )` is one of the functions that come built into SQL Server, and is known as an *aggregate function* because it can operate on more than one row at a time. You use `COUNT( * )` to get the number of rows, as shown in the following example that retrieves the `SupplierID` and number of rows for each group of `SupplierID` column values:

```
SELECT SupplierID, COUNT( * )
FROM Products
GROUP BY SupplierID;
```

[Figure 3.18](#) shows the results of this `SELECT` statement.



**Figure 3.18:** Using the GROUP BY clause to divide rows into blocks

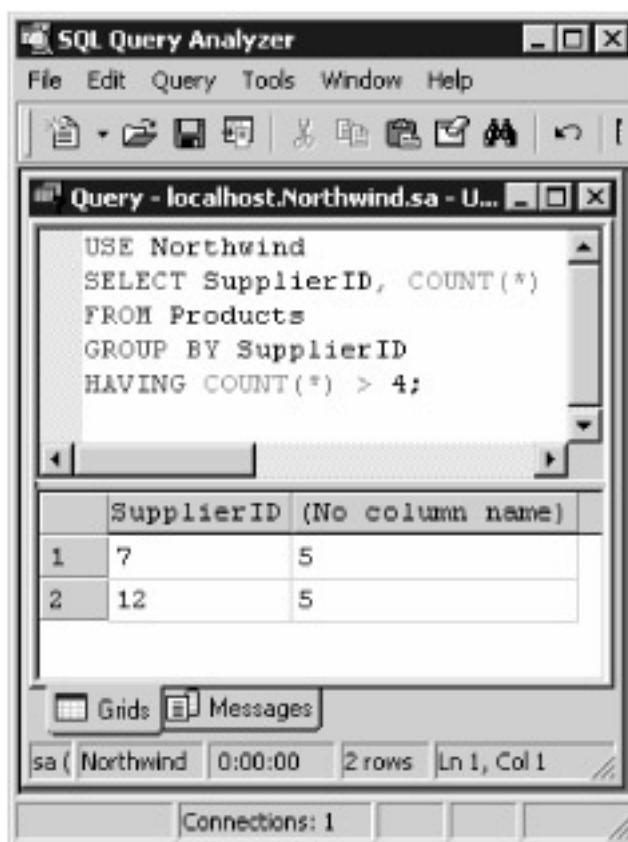
You'll learn more about the various SQL Server functions in the [next chapter](#).

## Restricting Retrieved Groups of Rows

You use the HAVING clause to restrict the groups of rows retrieved by the GROUP BY clause. For example, the following SELECT statement uses the HAVING clause to restrict the group of rows returned to those that have more than 4 rows in each group:

```
SELECT SupplierID, COUNT(*)
FROM Products
GROUP BY SupplierID
HAVING COUNT(*) > 4;
```

[Figure 3.19](#) shows the results of this SELECT statement.



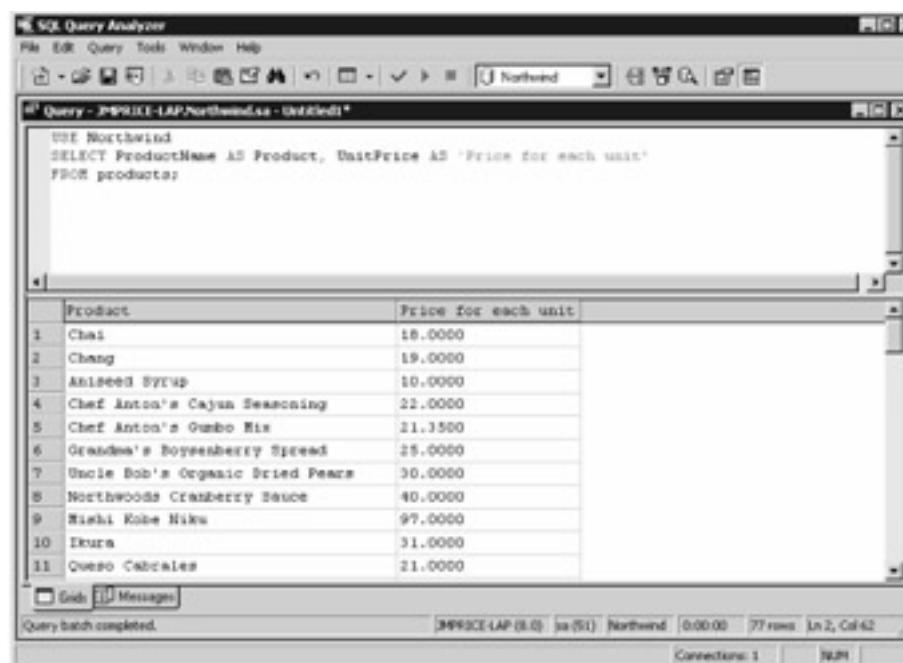
**Figure 3.19:** Using the HAVING clause to restrict retrieved groups of rows

## Specifying the Display Name for a Column and Aliasing a Table

You can use the AS clause to specify the name of a column when it is displayed in the output from a SELECT statement. You might want to do this when you need to display more friendly names or descriptive names for columns. For example, the following SELECT statement uses the AS clause to set the display name of the ProductName column to Product, and the UnitPrice column to Price for each unit:

```
SELECT ProductName AS Product, UnitPrice AS 'Price for each unit'
FROM products;
```

[Figure 3.20](#) shows the results of this SELECT statement.



**Figure 3.20:** Using the AS clause to specify the display name for columns

You can also use the AS clause to alias a table. You might want to do this if your table names are long. The following example uses the AS clause to alias the Customers and Orders tables as Cust and Ord respectively:

```
SELECT Cust.CustomerID, CompanyName, Address, OrderID, ShipAddress
FROM Customers AS Cust, Orders AS Ord
WHERE Cust.CustomerID = Ord.CustomerID
AND Cust.CustomerID = 'ALFKI';
```

## Performing Computations Based on Column Values

You typically use calculated fields to perform computations based on column values. For example, you might want to use a calculated field to compute the effect of increasing the UnitPrice column of the Products table by 20 percent. The following SELECT statement shows this:

```
SELECT UnitPrice * 1.20
FROM Products
WHERE ProductID = 1;
```

This example returns 21.600000. The new unit price is calculated using UnitPrice \* 1.20. This is an increase of 20 percent over the current unit price.

The next example concatenates the ContactName and ContactTitle columns from the Customers table for the row where the CustomerID equals ALFKI:

```
SELECT ContactName + ', ' + ContactTitle
FROM Customers
WHERE CustomerID = 'ALFKI';
```

This example returns Maria Anders, Sales Representative.

## Retrieving Rows From Multiple Tables

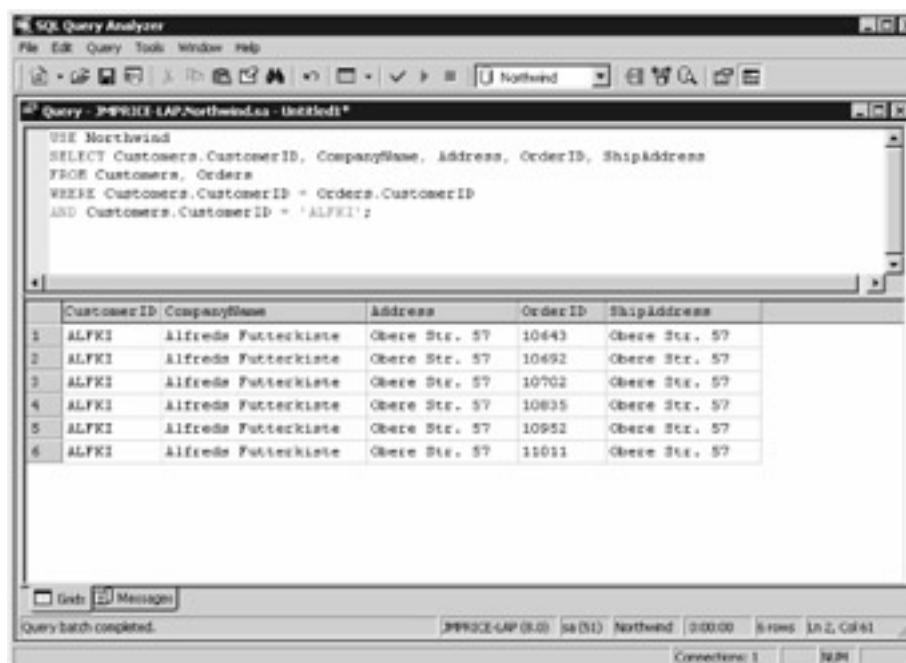
So far, you've seen SELECT statements that retrieve rows from only one table at a time. You'll often need to retrieve rows from multiple tables using the same SELECT statement.

For example, you might want to see all the orders placed by a customer. To do this, you must specify both the Customers and the Orders tables after the FROM keyword in the SELECT statement and use a *table join* in the WHERE clause. You must also specify the name of the table when referencing columns of the same name in both tables. The following SELECT statement shows this and retrieves the orders placed by the customer with a CustomerID of ALFKI:

```
SELECT Customers.CustomerID, CompanyName, Address, OrderID, ShipAddress
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
AND Customers.CustomerID = 'ALFKI';
```

Notice that the Customers and Orders tables are specified after the FROM keyword, and because both tables contain a column named CustomerID, the table name is placed before each reference to the respective column in each table. The table join is done on the CustomerID column of each table (Customers.CustomerID = Orders.CustomerID).

[Figure 3.21](#) shows the results of this SELECT statement.



**Figure 3.21:** Using a multitable SELECT statement to retrieve orders placed by a specific customer

The previous `SELECT` statement used the SQL standard format for joining tables. With SQL Server, you can also use the `JOIN` keyword for joining tables. The advantage of the `JOIN` keyword is you can use it to perform outer joins, which you'll learn about shortly. Here's an example that rewrites the previous `SELECT` statement using the `JOIN` keyword:

```
SELECT Customers.CustomerID, CompanyName, Address, OrderID, ShipAddress
FROM Customers
JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
AND Customers.CustomerID = 'ALFKI';
```

This `SELECT` statement returns the same results as the previous example.

The disadvantage of the previous two `SELECT` statements is that they return rows only where the join columns both contain a value, that is, neither column contains a null. This can be a problem if you have rows that have a null value in either of the columns used in the join and you need to actually retrieve those rows. Outer joins solve this problem. There are three types of outer joins:

- **LEFT OUTER JOIN** The `LEFT OUTER JOIN` (usually shortened to `LEFT JOIN`) returns all the rows from the table on the left of the join, including those with a column that contains a null value.
- **RIGHT OUTER JOIN** The `RIGHT OUTER JOIN` (usually shortened to `RIGHT JOIN`) returns all the rows from the table on the right of the join, including those with a column that contains a null value.
- **FULL OUTER JOIN** The `FULL OUTER JOIN` (usually shortened to `FULL JOIN`) returns all the rows from the tables on the left and right of the join, including those with a column that contains a null value.

Let's take a look at a couple of examples. First, perform the following `INSERT` to add a row to the `Products` table:

```
INSERT INTO Products (ProductName, SupplierID)
VALUES ('DVD Player', NULL);
```

**Note** You'll learn the details of the `INSERT` statement later in this chapter.

You don't need to specify the `ProductID` column because SQL Server will automatically supply a value using an identity. This identity was established when the `Products` table was created, and the identity generates a series of values that start with 1 and are incremented by 1 each time it is used. For example, the `ProductID` column initially contains a series of values from 1 to 77, therefore the next `INSERT` statement that adds a row to the `Products` table will set the `ProductID` column to 78 for that row—the next identity value.

You'll notice that the `SupplierID` column in the `INSERT` statement is null. If you now perform the following `SELECT` statement, you

won't see the new row because the `SupplierID` column of the new row is null and the `JOIN` won't return that row:

```
SELECT ProductID
FROM Products
JOIN Suppliers
ON Products.SupplierID = Suppliers.SupplierID;
```

To see the new row, you use `LEFT JOIN` in the `SELECT` statement to retrieve all rows from the table on the left of the join (in this case, the table on the left is the `Products` table):

```
SELECT ProductID
FROM Products
LEFT JOIN Suppliers
ON Products.SupplierID = Suppliers.SupplierID;
```

You can also use `LEFT JOIN` with `IS NULL` in the same `SELECT` statement to retrieve just the new row:

```
SELECT ProductID
FROM Products
LEFT JOIN Suppliers
ON Products.SupplierID = Suppliers.SupplierID
WHERE Products.SupplierID IS NULL;
```

## Retrieving Rows From a View

You use a view to retrieve a set of columns from one or more tables. You can think of a view as a more flexible way of examining the rows stored in the tables. For example, one of the views of the Northwind database retrieves an alphabetical list of products, and retrieves the product name and category name, among other columns. This information comes from both the `Products` and `Categories` tables. This view is named `Alphabetical list of products` and the `SELECT` statement that makes up this view is as follows:

```
SELECT Products.*, Categories.CategoryName
FROM Categories INNER JOIN Products ON
    Categories.CategoryID = Products.CategoryID
WHERE ((Products.Discontinued)=0));
```

You can retrieve all columns and rows from the underlying tables referenced by this view using the following `SELECT` statement:

```
SELECT *
FROM [Alphabetical list of products];
```

You can also retrieve individual columns from a view. For example, the following `SELECT` statement retrieves just the `ProductName` and `CategoryName` columns from the view:

```
SELECT ProductName, CategoryName
FROM [Alphabetical list of products];
```

## Adding a New Row to a Table

You use the `INSERT` statement to add a new row to a table. When adding a new row, you specify the name of the table, the optional column names, and the values for those columns. For example, the following `INSERT` statement adds a new row to the `Customers` table:

```
INSERT INTO Customers (
    CustomerID, CompanyName, ContactName, ContactTitle, Address,
    City, Region, PostalCode, Country, Phone, Fax
) VALUES (
    'JPCOM', 'Jason Price Company', 'Jason Price', 'Owner', '1 Main Street',
    'New York', NULL, '12345', 'USA', '(800)-555-1212', NULL
);
```

The `CustomerID` column is the primary key of the `Customers` table, therefore the new row must contain a unique value for this column. You'll notice that the `INSERT` statement specifies a null value for the `Region` and `Fax` columns (this is specified using the

NULL keyword).

You can use the Query Analyzer to enter `INSERT` statements. [Figure 3.22](#) shows the previous `INSERT`, along with a `SELECT` statement that retrieves the new row.

The screenshot shows the SQL Server Management Studio interface. In the top pane, a query window titled 'Query - JPCOM\JAP\Northwind.mdf - Untitled1' contains the following T-SQL code:

```

USE Northwind
INSERT INTO Customers (
    CustomerID, CompanyName, ContactName, ContactTitle, Address,
    City, Region, PostalCode, Country, Phone, Fax
) VALUES (
    'JPCOM', 'Jason Price Company', 'Jason Price', 'Owner',
    '1 Main Street', 'New York', NULL,
    '12345', 'USA', '(800)-555-1212', NULL
);
SELECT *
FROM Customers
WHERE CustomerID = 'JPCOM';

```

In the bottom pane, a results grid displays the inserted row:

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region
JPCOM	Jason Price Company	Jason Price	Owner	1 Main Street	New York	NULL

The status bar at the bottom indicates 'Query batch completed.'

**Figure 3.22:** Using an `INSERT` statement to add a new row to the `Customers` table

**Note** You must supply values for all columns that are defined as *NOT NULL* in a table. Also, the number of columns in the `INSERT` and `VALUES` lists must match, and the data type of each column in the `INSERT` and `VALUES` lists must also match.

When supplying values to all columns in a row, you may omit the column names and just supply the values for each column. For example:

```

INSERT INTO Customers VALUES (
    'CRCOM', 'Cynthia Red Company', 'Cynthia Red', 'Owner', '2 South Street',
    'New York', NULL, '12345', 'USA', '(800)-555-1212', NULL
);

```

## Modifying Rows in a Table

You use the `UPDATE` statement to update rows in a table. When updating a row, you specify the name of the table, the columns to update, and the new values for the columns.

**Warning** Typically, you should also use a `WHERE` clause to restrict the rows being updated. If you don't supply a `WHERE` clause, then all the rows in the specified table will be updated. In many cases, you'll specify the value for the primary key in your `WHERE` clause.

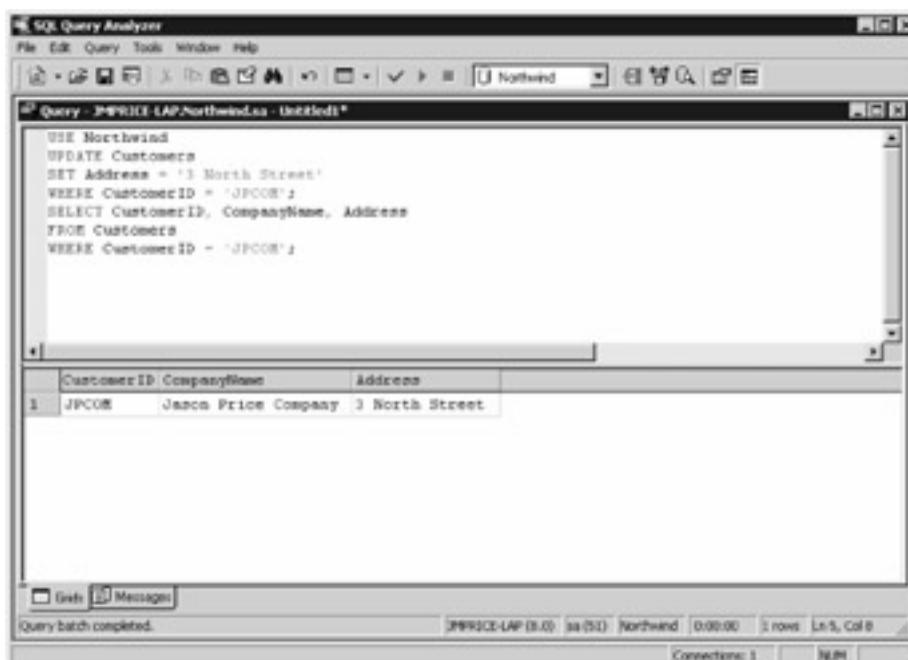
The following `UPDATE` statement modifies the `Address` column for the row in the `Customers` table with a `CustomerID` of `JPCOM`:

```

UPDATE Customers
SET Address = '3 North Street'
WHERE CustomerID = 'JPCOM';

```

[Figure 3.23](#) shows this `UPDATE` statement, along with a `SELECT` statement that retrieves the modified row.



**Figure 3.23:** Using an UPDATE statement to modify the Address column of a row in the Customers table

You can use an UPDATE statement to modify multiple columns. For example, the following UPDATE statement modifies the Address and ContactTitle columns:

```
UPDATE Customers
SET Address = '5 Liberty Street', ContactTitle = 'CEO'
WHERE CustomerID = 'JPCOM';
```

## Removing Rows From a Table

You use the DELETE statement to remove rows from a table. When removing a row, you specify the name of the table and the rows to delete using a WHERE clause.

**Warning** If you omit the *WHERE* clause in a *DELETE* statement, all rows from the table will be deleted. Make sure you provide a *WHERE* clause if you don't want to remove all the rows from the table. Typically, you'll specify the value for the primary key in your *WHERE* clause.

The following DELETE statement removes the row from the Customers table where the CustomerID is CRCOM:

```
DELETE FROM Customers
WHERE CustomerID = 'CRCOM';
```

[Figure 3.24](#) shows this DELETE statement, along with a SELECT statement that demonstrates that the row has been removed.



**Figure 3.24:** Using an UPDATE statement to remove a row from the `Customers` table

In the [next section](#), you'll learn how the database software maintains the integrity of the information stored in the database.

## Maintaining Database Integrity

The database software ensures that the information stored in the tables is consistent. In technical terms, it maintains the integrity of the information. Two examples of this are the following:

- The primary key of a row always contains a unique value.
- The foreign key of a row in the child table always references a value that exists in the parent table.

Let's take a look at what happens when you try to insert a row into a table with a primary key that already exists. The following `INSERT` statement attempts to add a row to the `Customers` table with a `CustomerID` of `ALFKI` (a row with this primary key already exists in the `Customers` table):

```
INSERT INTO Customers (
    CustomerID, CompanyName, ContactName, ContactTitle, Address,
    City, Region, PostalCode, Country, Phone, Fax
) VALUES (
    'ALFKI', 'Jason Price Company', 'Jason Price', 'Owner', '1 Main Street',
    'New York', NULL, '12345', 'USA', '(800)-555-1212', NULL
);
```

If you attempt to run this `INSERT` statement, you'll get the following error message from the database:

```
Violation of PRIMARY KEY constraint 'PK_Customers'.
Cannot insert duplicate key in object 'Customers'.
The statement has been terminated.
```

This `INSERT` statement fails because an existing row in `Customers` table already contains the primary key value `ALFKI`. The message tells you that the primary key specified in the `INSERT` statement already exists in the `Customers` table. The constraint name `PK_Customers` is the name of the table constraint assigned to the primary key when the `Customers` table was originally created. At the end, the message indicates that the statement has been terminated, meaning that the `INSERT` statement has not been performed.

Let's take a look at what happens when you try to modify a primary key in a parent table with a value that is referenced in a foreign key in a child table. The following `UPDATE` statement attempts to modify the `CustomerID` from `ALFKI` to `ALFKZ` in the parent `Customers` table (this row is referenced by rows in the child `Orders` table):

## Using SQL

```
UPDATE Customers
SET CustomerID = 'ALFKZ'
WHERE CustomerID = 'ALFKI';
```

If you attempt to run this UPDATE statement, you'll get the following error message:

```
UPDATE statement conflicted with COLUMN REFERENCE constraint
'FK_Orders_Customers'. The conflict occurred in database
'Northwind', table 'Orders', column 'CustomerID'.
The statement has been terminated.
```

This UPDATE statement fails because the row containing the primary key value ALFKI is referenced by rows in the Orders table. The message tells you that the new value for the CustomerID column violates the foreign key constraint on the CustomerID column of the Orders table. This constraint is named FK\_Orders\_Customers.

Also, you can't remove a row from a parent table that is referenced by a row in a child table. For example, the following DELETE statement attempts to remove the row from the Customers table where the CustomerID column equals ALFKI (this row is referenced by rows in the Orders table):

```
DELETE FROM Customers
WHERE CustomerID = 'ALFKI';
```

If you attempt to run this DELETE statement, you'll get the same error message that was shown for the previous UPDATE statement. This DELETE statement fails because the Orders table contains rows that reference the row in the Customers table, and removing this row would make the database inconsistent because the rows in the Orders table wouldn't reference a valid row.

## Grouping SQL Statements

By default, when you run an INSERT, UPDATE, or DELETE statement, SQL Server permanently records the results of the statement in the database. This might not always be your desired result. For example, in the case of a banking transaction, you might want to withdraw money from one account and deposit it into another. If you had two separate UPDATE statements that performed the withdrawal and deposit, then you would want to make the results of each UPDATE statement permanent only as one unit. If either UPDATE failed for some reason, then you would want to undo the results of both UPDATE statements.

**Note** Permanently recording the results of SQL statements is known as a commit, or committing the SQL statements. Undoing the results of SQL statements is known as a *rollback*, or *rolling back* the SQL statements.

You can group SQL statements into a *transaction*. You can then commit or roll back the SQL statements in that transaction as one unit. For example, the two UPDATE statements in the previous banking example could be placed into a transaction, and then you could commit or roll back that transaction as one unit, depending on whether both of the UPDATE statements succeeded.

You start a transaction using the BEGIN TRANSACTION statement or the shorthand version, BEGIN TRANS. You then perform your SQL statements that make up the transaction. To commit the transaction, you perform a COMMIT TRANSACTION statement or one of the shorthand versions, COMMIT TRANS or COMMIT. To roll back the transaction, you perform a ROLLBACK TRANSACTION statement or one of the shorthand versions, ROLLBACK TRANS or ROLLBACK.

**Note** By default, transactions are rolled back. You should always explicitly commit or roll back a transaction to indicate what you want to do.

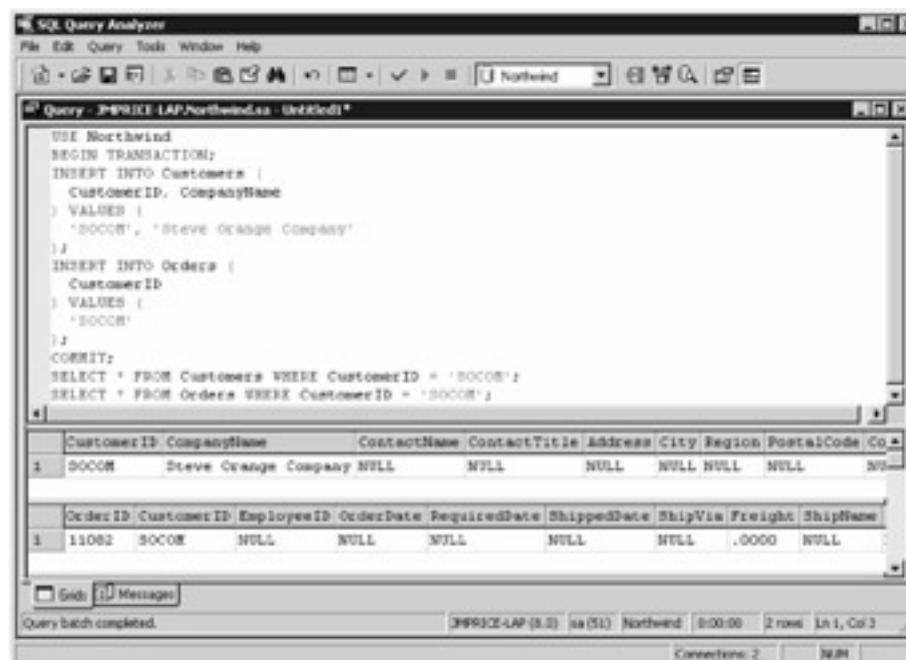
Let's take a look at an example. The following transaction consists of two INSERT statements: the first adds a row to the Customers table, and the second adds a row to the Orders table. At the end, the transaction is committed using a COMMIT statement:

```
BEGIN TRANSACTION;
INSERT INTO Customers (
    CustomerID, CompanyName
) VALUES (
    'SOCOM', 'Steve Orange Company'
);
INSERT INTO Orders (
```

## Using SQL

```
CustomerID  
) VALUES ('  
    'SOCOM'  
) ;  
COMMIT;
```

[Figure 3.25](#) shows this transaction, along with two SELECT statements that show the two new rows.



**Figure 3.25:** Using a transaction

The next transaction consists of similar INSERT statements, except this time the transaction is rolled back using a ROLLBACK statement.

```
BEGIN TRANSACTION;  
INSERT INTO Customers (CustomerID, CompanyName  
) VALUES ('  
    'SYCOM', 'Steve Yellow Company'  
) ;  
INSERT INTO Orders (CustomerID  
) VALUES ('  
    'SYCOM'  
) ;  
ROLLBACK;
```

Because the transaction is rolled back, the two rows added by the INSERT statements are undone.

You should check for errors in a transaction before deciding to perform a COMMIT or ROLLBACK because errors do not always stop the next line from processing. To do this in SQL Server, you use the @@ERROR function. This function returns zero whenever a statement is executed and doesn't cause an error. If @@ERROR returns a nonzero value, you know an error occurred. If @@ERROR returns 0, you perform a COMMIT, otherwise you perform a ROLLBACK.

You can also assign a name to your transaction in the BEGIN TRANSACTION statement. This is useful as it shows which transaction you are working on.

The following example shows the naming of a transaction, along with the use of the @@ERROR function to determine whether to perform a COMMIT or ROLLBACK:

```
BEGIN TRANSACTION MyTransaction;
```

```

INSERT INTO Customers (
    CustomerID, CompanyName
) VALUES (
    'SYCOM', 'Steve Yellow Company'
);
INSERT INTO Orders (
    CustomerID
) VALUES (
    'SYCOM'
);
IF @@Error = 0
    COMMIT TRANSACTION MyTransaction;
ELSE
    ROLLBACK TRANSACTION MyTransaction;

```

Notice that the name of the transaction is `MyTransaction`, and that this name is used in the `COMMIT` and `ROLLBACK` statements.

**Note** You use the `IF` statement to conditionally execute a statement. You'll learn more about this in [Chapter 4](#), "Introduction to Transact-SQL Programming."

## Introducing Data Definition Language (DDL) Statements

As mentioned earlier, DDL statements allow you to create database structures such as tables and indexes. In this section, you'll learn how to create, alter, and drop a table, and create and drop an index.

### Creating a Table

You create a table using the `CREATE TABLE` statement. For example, let's say you wanted to store the details for a number of persons in the database. Assume you want to store a person's first name, last name, and date of birth. Let's call this table `Persons`. You also want to uniquely identify each row in the `Persons` table using a numeric ID, which acts as the primary key for the table. The following `CREATE TABLE` statement creates the `Persons` table:

```

CREATE TABLE Persons (
    PersonID int CONSTRAINT PK_Persons PRIMARY KEY,
    FirstName nvarchar(15) NOT NULL,
    LastName nvarchar(15) NOT NULL,
    DateOfBirth datetime
);

```

You use the `CONSTRAINT` clause to restrict the values stored in a table or column. You'll notice that the `CONSTRAINT` clause is used to specify the primary key for the table using the keywords `PRIMARY KEY`. The primary key is the `PersonID` column, and this constraint is named `PK_Persons`. The `ID` column is an `int`, meaning that it stores integers. Every row in the `Persons` table must have a unique number for the `PersonID` column.

The `FirstName` and `LastName` columns are `nvarchar` columns that may store up to 15 characters. Both of these columns are defined using the `NOT NULL` constraint. `NOT NULL` indicates that you must supply a value to the column. The default is `NULL`, meaning that you don't have to supply a value to the column.

**Note** Primary keys always require a value, and are therefore implicitly `NOT NULL`.

The `DateOfBirth` column is a `datetime`, meaning that it can store a date and time. This column doesn't have a `NOT NULL` constraint and therefore uses the default of `NULL`.

### Altering a Table

You alter an existing table using the `ALTER TABLE` statement. You can add or drop a column, and add or drop a constraint using the `ALTER TABLE` statement. For example, the following `ALTER TABLE` statement adds a column named `Address` to the `Persons` table:

```
ALTER TABLE Persons
ADD Address nvarchar(50);
```

The `Address` column is an `nvarchar` that can store up to 50 characters.

The next example drops the `Address` column from the `Persons` table:

```
ALTER TABLE Persons
DROP COLUMN Address;
```

The next example adds a column named `EmployerID` to the `Persons` table, which records the company that a person works for:

```
ALTER TABLE Persons
ADD EmployerID nchar(5) CONSTRAINT FK_Persons_Customers REFERENCES
Customers(CustomerID);
```

The `EmployerID` column is a foreign key to the `CustomerID` column of the `Customers` table. The constraint is named `FK_Persons_Customers`.

## Dropping a Table

You drop a table from the database using the `DROP TABLE` statement. For example, the following statement drops the `Persons` table:

```
DROP TABLE Persons;
```

## Creating an Index

You add an index to a table using the `CREATE INDEX` statement. An index allows you to potentially find a row more rapidly when you use the column with the index in a `WHERE` clause. For example, the following `CREATE INDEX` statement adds an index to the `LastName` column of the `Persons` table:

```
CREATE INDEX LastNameIndex
ON Persons(LastName);
```

**Tip** If you frequently use a column in a `WHERE` clause, you should consider adding an index to that column.

Generally, you should create an index on a column only when you find that you are retrieving a small number of rows from a table containing many rows. A good rule of thumb is that an index is useful when you expect any single query to retrieve 10 percent or less of the total rows in a table. This means that the candidate column for an index should be used to store a wide range of values. A good candidate for indexing would be a column containing a unique number for each record, while a poor candidate for indexing would be a column that contains only a small range of numeric codes, such as 1, 2, 3, or 4. This consideration applies to all database types, not just numbers.

Normally, the DBA is responsible for creating indexes, but as an application developer, you might know more about the application than the DBA and be able to recommend which columns are good candidates for indexing.

## Dropping an Index

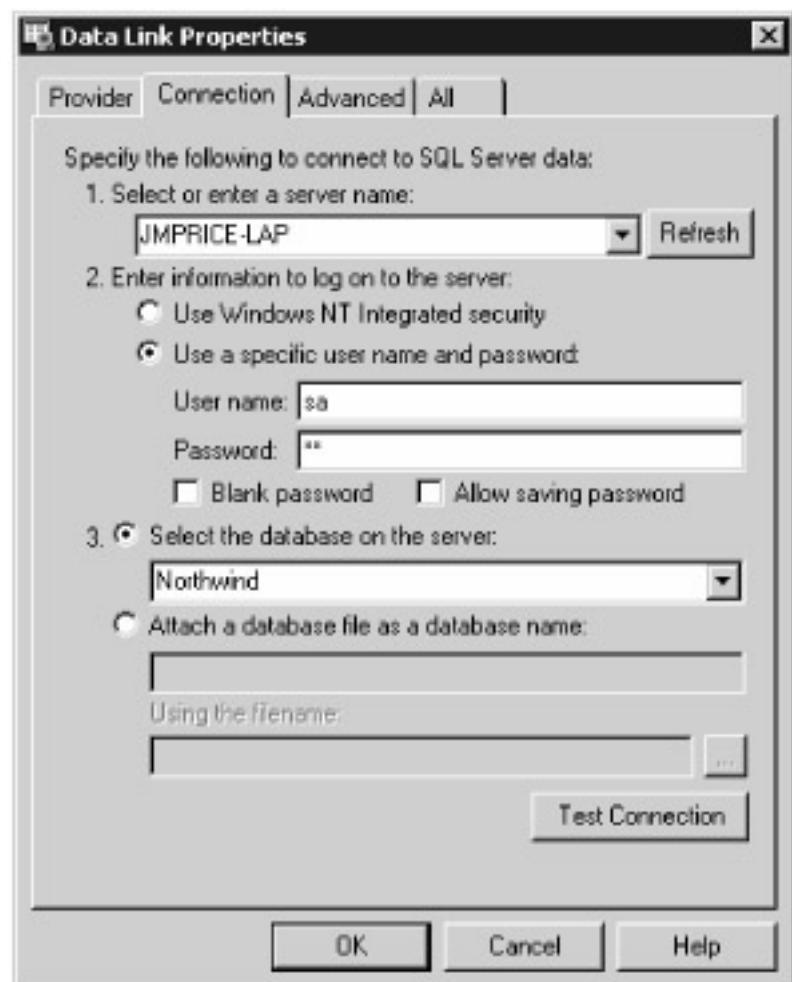
You drop an index from a table using the `DROP INDEX` statement. For example, the following `DROP INDEX` statement drops `LastNameIndex` from the `Persons` table:

```
DROP INDEX Persons.LastNameIndex;
```

## Accessing a Database Using Visual Studio .NET

Visual Studio .NET's Server Explorer allows you to use a subset of the features contained in the Databases folder of Enterprise Manager. Specifically, Server Explorer allows you to view, create, and edit databases, database diagrams, tables, views, stored procedures, and user-defined functions. In this section, you'll be introduced to Server Explorer and some of its functionality. As you'll see, Server Explorer operates in a similar way to Enterprise Manager, which was covered in [Chapter 2](#). Because of the similarity between Server Explorer and Enterprise Manager, I'll only briefly cover Server Explorer here. You should feel free to experiment with Server Explorer yourself.

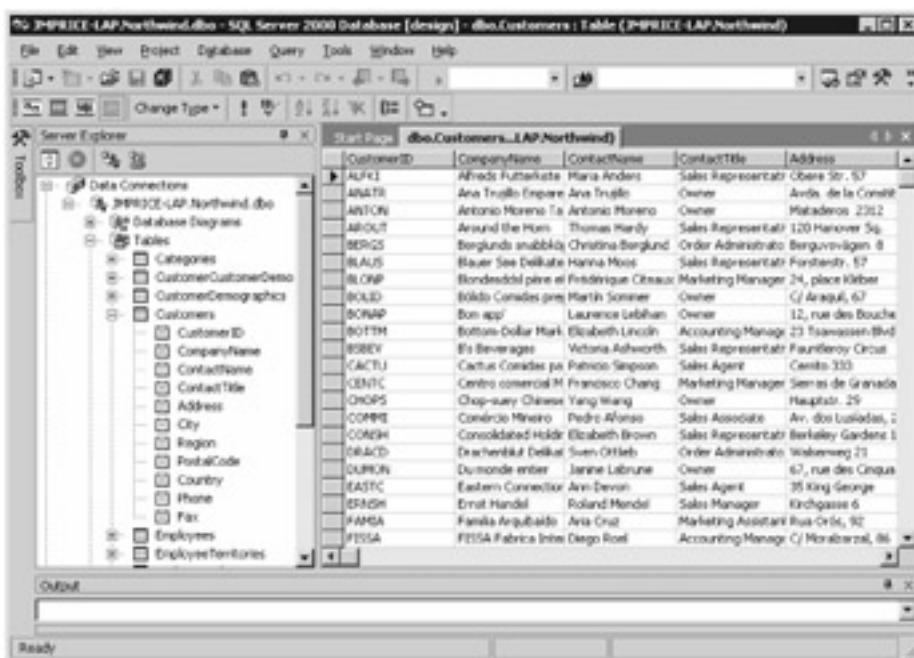
Your first step is to connect to a database. To do this, you select Tools > Connect To Database. This displays the Data Link Properties dialog box. [Figure 3.26](#) shows this dialog box with appropriate entries to connect to the Northwind database running on the computer JMPRICE-LAP.



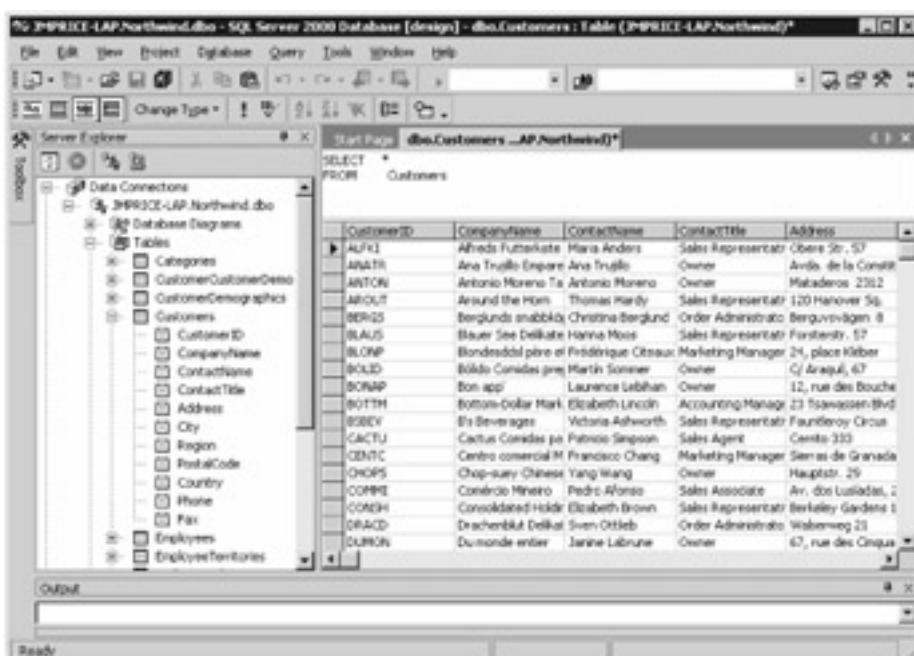
**Figure 3.26:** Entering database details using the Data Link Properties dialog box

Once you've entered your database details, your second step is to click the Test Connection button to verify the database connection details. Click the OK button once your test succeeds.

Once you've connected to the database, you can view things such as the tables. You can also retrieve and modify rows in the tables. You can drill down to the tables in the database by clicking the Add icon in the tree in Server Explorer, and you can retrieve the rows from a table by clicking the right mouse button on the table in the tree and selecting Retrieve Data From Table in the pop-up window. [Figure 3.27](#) shows the rows from the Customers table.

**Figure 3.27:** Viewing the rows in the Customers table using the Server Explorer

You can enter SQL statements by clicking the Show SQL Pane button in the toolbar, as shown in [Figure 3.28](#).

**Figure 3.28:** Entering a SQL statement

You can build SQL statements visually by clicking the Show Diagram button in the toolbar and selecting columns from the table, as shown in [Figure 3.29](#). As you can see, I've selected the ContactName, CompanyName, and CustomerID columns from the Customers table.



**Figure 3.29:** Building a SQL statement visually

You can view the properties of a column in a table by clicking the right mouse button over the column and selecting Properties from the pop-up window. [Figure 3.30](#) shows the properties of the `CustomerID` column of the `Customers` table.



**Figure 3.30:** Properties of the `CustomerID` columns

You've only scratched the surface of the Server Explorer in this section. If you have VS .NET, you should feel free to experiment with the Server Explorer—that's the best way to learn.

## Summary

In this chapter, you learned how to use SQL (pronounced **sequel**) to access a database. SQL is the standard language for accessing relational databases. With SQL, you tell the database what data you want to access, and the database software figures out exactly how to get that data. You can enter and run SQL statements against a SQL Server database using the Query Analyzer tool.

There are two main types of SQL statements: Data Manipulation Language (DML) statements, and Data Definition Language (DDL) statements. DML statements allow you to retrieve, add, modify, and delete rows stored in the database. DDL statements allow you to create database structures such as tables and indexes.

You use a SQL `SELECT` statement to retrieve rows, an `INSERT` statement to add rows, an `UPDATE` statement to modify rows, and a `DELETE` statement to remove rows.

You examined Visual Studio .NET's Server Explorer tool that allows you to connect to a database. Server Explorer contains a subset of the functionality offered by Enterprise Manager.

In the [next chapter](#), you'll be introduced to Transact-SQL programming.

## Chapter 4: Introduction to Transact-SQL Programming

### Overview

Transact-SQL is Microsoft's implementation of SQL, and it contains additional programming constructs. (It's often shortened to *T-SQL*, a convention you'll see in this chapter.) T-SQL enables you to write programs that contain SQL statements, along with standard programming constructs such as variables, conditional logic, loops, procedures, and functions.

Featured in this chapter:

- Fundamentals of Transact-SQL
- [Using functions](#)
- Creating user-defined functions
- [Introducing stored procedures](#)
- Introducing triggers

## Fundamentals of Transact-SQL

In this section, you'll learn some of the essential programming constructs available in T-SQL. Specifically, you'll see how to use variables, comments, and conditional logic. You'll also see how to use a number of statements that allow you to perform jumps and loops. Finally, you'll examine cursors, which allow you to process rows returned from the database one at a time.

Let's start by looking at variables.

### Using Variables

A **variable** allows you to store a value in the memory of a computer. Each variable has a **type** that indicates the kind of value that will be stored in that variable. You can use any of the types shown earlier in [Table 2.3](#) of [Chapter 2](#), "Introduction to Databases."

You declare a variable using the `DECLARE` statement, followed by the variable name and the type. You place an at character (@) before the start of the variable name. The following syntax illustrates the use of the `DECLARE` statement:

```
DECLARE @name type
```

Where `name` is the name of your variable, and `type` is the variable type.

For example, the following statements declare two variables named `MyProductName` and `MyProductID`:

```
DECLARE @MyProductName nvarchar(40)
DECLARE @MyProductID int
```

As you can see, `MyProductName` is of the `nvarchar` type, and `MyProductID` is of the `int` type.

You can place more than one variable declaration on the same line. For example:

```
DECLARE @MyProductName nvarchar(40), @MyProductID int
```

Variables are initially set to null. You set a variable's value using the `SET` statement. For example, the following statements set `MyProductName` to `Chai` and `MyProductID` to 7:

```
SET @MyProductName = 'Chai'
SET @MyProductID = 7
```

The following `SELECT` statement then uses these variables in the `WHERE` clause:

```
SELECT ProductID, ProductName, UnitPrice
FROM Products
WHERE ProductID = @MyProductID
OR ProductName = @MyProductName;
```

You can execute T-SQL using Query Analyzer, and [Figure 4.1](#) shows the output from the examples shown in this section.



**Figure 4.1:** Executing T-SQL using Query Analyzer

## Using Comments

You add **comments** to describe your code, making it more understandable for both yourself and other programmers. You might think you understand your own code inside out, but when you return to it for maintenance six months later, you might have forgotten the intricacies of your own creation! The point is that you should add comments to your code to aid understanding, but don't think you have to comment every line. Use comments judiciously.

You need to mark your comments with specific characters so SQL Server ignores them and doesn't try to process them as code. There are two types of comments: single-line and multi-line. A single-line comment uses two negative signs (--) and may span only one line, as shown here:

```
-- A single-line comment may only span one line.
```

The -- tells SQL Server to ignore everything up to the end of that line.

A multi-line comment begins with an open comment mark /\* and ends with a close comment mark \*/:

```
/* A multi-line comment
   may span more than one
   line. */
```

The /\* tells SQL Server to ignore everything up to the next \*/ mark, no matter how many lines forward it is. If you were to use single-line comments in this example, you would have to add -- characters at the beginning of every line that made up the comment.

Multi-line comments can of course also span only one line:

```
/* Another comment */
```

## Using Conditional Logic

Conditional logic allows you to execute different branches of code based on the Boolean true or false value of a given expression. For example, you might want to check if an error condition is true and display a message. You use the IF and optional ELSE keywords to perform conditional logic. The following syntax illustrates the use of conditional logic:

```
IF condition
    statement1
[ELSE
    statement2]
```

Where *condition* is a Boolean expression that evaluates to true or false. If *condition* is true, then *statement1* is executed, otherwise *statement2* is executed.

**Note** You can replace a single statement with multiple statements by placing those statements within `BEGIN` and `END` statements. This rule applies to all T-SQL programming constructs.

The following syntax shows the replacement of single statements with a block of statements placed within `BEGIN` and `END`:

```
IF condition
BEGIN
    statements1
END
ELSE
BEGIN
    statements2
END
```

Where `statements1` and `statements2` are multiple statements. You can also use an optional `ELSE` statement to execute a different branch of code if the condition is false.

**Note** You can nest `IF` statements to any level.

The following example displays the `ProductID`, `ProductName`, and `UnitPrice` columns for any rows from the `Products` table that have a `UnitPrice` of less than \$5. You'll notice the use of the `PRINT` statement to output a line in this example.

```
IF (SELECT COUNT(*) FROM Products WHERE UnitPrice < 5) > 0
BEGIN
    PRINT 'The following products have a UnitPrice of less than $5:'
    SELECT ProductID, ProductName, UnitPrice
    FROM Products
    WHERE UnitPrice < 5
END
ELSE
BEGIN
    PRINT 'There are no products that have a UnitPrice of less than $5'
END
```

## Using `CASE` Statements

You use the `CASE` statement to compare a value against a list of values and execute one or more statements when a match is found. For example, the following `CASE` statement returns `Massachusetts`:

```
CASE 'MA'
    WHEN 'CA' THEN 'California'
    WHEN 'MA' THEN 'Massachusetts'
    WHEN 'NY' THEN 'New York'
END
```

The next example uses a `SELECT` statement to retrieve the value `Massachusetts` returned by the `CASE` statement:

```
DECLARE @State nchar(2)
SET @State = 'MA'
DECLARE @StateName nvarchar(15)
SELECT CASE @State
    WHEN 'CA' THEN 'California'
    WHEN 'MA' THEN 'Massachusetts'
    WHEN 'NY' THEN 'New York'
END
```

You can store the value retrieved by the `SELECT` statement in a variable, as shown in the next example:

```
DECLARE @State nchar(2)
SET @State = 'MA'
```

```

DECLARE @StateName nvarchar(15)
SELECT @StateName =
CASE @State
    WHEN 'CA' THEN 'California'
    WHEN 'MA' THEN 'Massachusetts'
    WHEN 'NY' THEN 'New York'
END
PRINT @StateName

```

The output from this example is as follows:

```
Massachusetts
```

You can also compare a column value in a CASE statement. For example:

```

SELECT Price =
CASE
    WHEN UnitPrice IS NULL THEN 'Unknown'
    WHEN UnitPrice < 10 THEN 'Less than $10'
    WHEN UnitPrice = 10 THEN '$10'
    ELSE 'Greater than $10'
END
FROM Products

```

You'll notice from this example that you can also supply a catchall ELSE condition in a CASE statement.

## Using WHILE Loops

You use a WHILE loop to run one or more statements multiple times. A WHILE loop runs until a specified condition evaluates to false. The syntax for a WHILE loop is as follows:

```
WHILE condition
    statement
```

The following example shows a WHILE loop:

```

DECLARE @count int
SET @count = 5
WHILE (@count > 0)
BEGIN
    PRINT 'count = ' + CONVERT(nvarchar, @count)
    SET @count = @count -1
END

```

This loop runs until the count variable reaches 0, and the output from this code is as follows:

```
count = 5
count = 4
count = 3
count = 2
count = 1
```

The CONVERT( ) function is used to convert a value from one type to another. For example, CONVERT (nvarchar, @count) converts the count variable to the nvarchar type, which can then be used with the PRINT statement.

## CONTINUE Statement

You use the CONTINUE statement to start the next iteration of a WHILE loop immediately, skipping over any remaining code in the loop. The CONTINUE statement causes execution to jump back to the start of the loop.

The following example shows a `WHILE` loop that uses the `CONTINUE` statement to start the next iteration of the loop if the `count` variable is equal to 2:

```
DECLARE @count int
SET @count = 5
WHILE (@count > 0)
BEGIN
    PRINT 'count = ' + CONVERT(nvarchar, @count)
    SET @count = @count -1
    IF (@count = 2)
        BEGIN
            SET @count = @count -1
            CONTINUE
        END
    END
END
```

The output from this code is as follows:

```
count = 5
count = 4
count = 3
count = 1
```

You'll notice that the display of `count = 2` is missing. This is because the `CONTINUE` statement skips that iteration.

## **BREAK Statement**

You use the `BREAK` statement to end a `WHILE` loop immediately. The `BREAK` statement causes execution to jump out of the loop and continue executing any statements after the loop.

The following example shows a `WHILE` loop that uses the `BREAK` statement to end the loop if the `count` variable is equal to 2:

```
DECLARE @count int
SET @count = 5
WHILE (@count > 0)
BEGIN
    PRINT 'count = ' + CONVERT(nvarchar, @count)
    SET @count = @count -1
    IF (@count = 2)
        BEGIN
            BREAK
        END
    END
END
```

The output from this code is as follows:

```
count = 5
count = 4
count = 3
```

## **Using Labels and the *GOTO* Statement**

You use the `GOTO` statement to jump to a specified *label* in your code; you use a label to identify a statement in your code. You must define the label before issuing the `GOTO` to that label. Before I show you the details of the `GOTO` statement, you should be aware that its use is considered poor programming practice, and you should avoid it if at all possible. It is usually possible to structure code so that you don't need to use the `GOTO` statement. Having said that, I've included it in this chapter for completeness.

As mentioned, the `GOTO` statement requires that you create a label in your program. You do this by placing an identifier containing the label name in your code, followed by a colon (:). The following example creates a label named `myLabel`:

```
myLabel:
```

You may then use the GOTO statement to jump to that label, for example:

```
GOTO myLabel
```

The following example shows the use of a label and the GOTO statement:

```
DECLARE @count int
SET @count = 5
myLabel:
PRINT 'count = ' + CONVERT(nvarchar, @count)
SET @count = @count -1
IF (@count > 0)
BEGIN
    GOTO myLabel
END
```

The output from this code is as follows:

```
count = 5
count = 4
count = 3
count = 2
count = 1
```

## Using ***RETURN*** Statements

You use the RETURN statement to exit from a stored procedure or group of statements. Any statements that follow your return are not executed. You can also return a value using the RETURN statement.

The syntax for the RETURN statement is as follows:

```
RETURN [int_expression]
```

Where *int\_expression* is any expression that evaluates to an int value.

**Note** You can return a value only when using the *RETURN* statement with a stored procedure. You'll see an example of that later in the "[Introducing Stored Procedures](#)" section.

The following example shows the use of the RETURN statement:

```
DECLARE @count int
SET @count = 5
WHILE (@count > 0)
BEGIN
    PRINT 'count = ' + CONVERT(nvarchar, @count)
    SET @count = @count -1
    IF (@count = 2)
        BEGIN
            RETURN
        END
    END
END
```

The output from this code is as follows:

```
count = 5
count = 4
count = 3
```

## Using ***WAITFOR*** Statements

There are times when you want your program to pause before running some code to perform a specific action, such as running a batch program at night to update customer records. You use the `WAITFOR` statement to specify a time interval or time to wait until continuing execution of code.

The syntax for the `WAITFOR` statement is as follows:

```
WAITFOR {DELAY 'time interval' | TIME 'actual time'}
```

You can specify the time interval to wait using the `DELAY` keyword, or you can specify the actual time to wait until using the `TIME` keyword. You can specify a time interval or an actual time in the format `HH:MM:SS`, where `HH` is the hour (in 24-hour format), `MM` is the minute, and `SS` is the second.

Here are some examples:

- `WAITFOR DELAY '00:00:05'` waits for a time interval of 5 seconds.
- `WAITFOR DELAY '23:10:25'` waits for a time interval of 23 hours, 10 minutes, and 25 seconds.
- `WAITFOR TIME '20:15:10'` waits until 10 seconds after 10:15 PM.

The following example prints a message after 5 seconds have elapsed:

```
BEGIN
    WAITFOR DELAY '00:00:05'
    PRINT '5 seconds have elapsed'
END
```

## Using `RAISERROR` Statements

You use the `RAISERROR` statement to generate an error message. You'll typically want to do this if an error occurs in one of your stored procedures, which you'll see how to use later in the section "Creating Stored Procedures."

The simplified syntax for the `RAISERROR` statement is as follows:

```
RAISERROR ( {number | description} {, severity, state})
```

Where `number` is the error number, which must be between 50,001 and 2,147,483,648. The `description` is a message that cannot exceed 400 characters. The `severity` is the degree of the error and must be between 0 and 18 (18 is the most severe error). The `state` is an arbitrary value that must be between 1 and 127, and represents information about the invocation state of the error.

The following examples show the use of the `RAISERROR` statement:

```
RAISERROR (50001, 15, 1)
RAISERROR ('No row with that ProductID was found', 10, 1)
```

## Using Cursors

When you execute a `SELECT` statement, all the rows are returned in one go. This might not always be appropriate. For example, you might want to take some action based on the column values retrieved for a particular row. To do this, you can use a *cursor* to process rows retrieved from the database one row at a time. A cursor allows you to step through the rows returned by a particular `SELECT` statement.

You follow these steps when using a cursor:

1. Declare variables to store the column values from the `SELECT` statement.
2. Declare the cursor, specifying your `SELECT` statement.
3. Open your cursor.
4. Fetch the rows from your cursor.
5. Close your cursor.

You'll learn the details of these steps in the following sections.

### Step 1: Declare Variables to Store the Column Values from the `SELECT` Statement

These variables must be compatible with the column types for the retrieved rows. For example, you'll want to use an `int` variable to store the value from an `int` column, and so on.

The following example declares three variables to store the `ProductID`, `ProductName`, and `UnitPrice` columns from the `Products` table:

```
DECLARE @MyProductID int
DECLARE @MyProductName nvarchar(40)
DECLARE @MyUnitPrice money
```

### Step 2: Declare the Cursor

A cursor declaration consists of a name that you assign to the cursor and the `SELECT` statement that you want to execute. This `SELECT` statement is not actually run until you open the cursor. You declare your cursor using the `DECLARE` statement.

The following example declares a cursor named `ProductCursor` with a `SELECT` statement that retrieves the `ProductID`, `ProductName`, and `UnitPrice` columns for the first 10 products from the `Products` table:

```
DECLARE ProductCursor CURSOR FOR
SELECT ProductID, ProductName, UnitPrice
FROM Products
WHERE ProductID <= 10
```

### Step 3: Open the Cursor

Now it's time to open your cursor, which runs the `SELECT` statement previously defined in the `DECLARE` statement. You open a cursor using the `OPEN` statement. The following example opens `ProductCursor`, and therefore also runs the `SELECT` statement that retrieves the rows from the `Products` table:

```
OPEN ProductCursor
```

### Step 4: Fetch the Rows from the Cursor

Now you must read each row from your cursor. To do this, you use the `FETCH` statement. Your cursor may contain many rows, and therefore a `WHILE` loop is required to read each row in turn. To determine when the loop is to end, you can use the

`@@FETCH_STATUS` function. This function returns one of the possible values shown in [Table 4.1](#).

**Table 4.1: RETURN VALUES FROM THE `@@FETCH_STATUS` FUNCTION**

VALUE	DESCRIPTION
0	FETCH statement successfully returned a row.
-1	FETCH statement failed or the requested row was outside the result set.
-2	Row fetched was missing.

The following example shows a loop that reads each row from `ProductCursor`:

```

FETCH NEXT FROM ProductCursor
INTO @MyProductID, @MyProductname, @MyUnitPrice
PRINT '@MyProductID = ' + CONVERT(nvarchar, @MyProductID)
PRINT '@MyProductName = ' + CONVERT(nvarchar, @MyProductName)
PRINT '@MyUnitPrice = ' + CONVERT(nvarchar, @MyUnitPrice)
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM ProductCursor
    INTO @MyProductID, @MyProductname, @MyUnitPrice
    PRINT '@MyProductID = ' + CONVERT(nvarchar, @MyProductID)
    PRINT '@MyProductName = ' + CONVERT(nvarchar, @MyProductName)
    PRINT '@MyUnitPrice = ' + CONVERT(nvarchar, @MyUnitPrice)
END

```

You'll notice that the condition `@@FETCH_STATUS = 0` is used in the `WHILE` loop to check that the `FETCH` statement successfully returned a row. When this condition is no longer `true`, the loop ends.

**Tip** You can get the number of rows stored in a cursor using the `@@CURSOR_ROWS` function. You'll learn more about functions later in the "[Using Functions](#)" section.

## Step 5: Close the Cursor

Close your cursor using the `CLOSE` statement. The following example closes `ProductCursor`:

```
CLOSE ProductCursor
```

You should also remove the reference to your cursor using the `DEALLOCATE` statement. This frees the system resources used by your cursor. The following example removes the reference to `ProductCursor` using the `DEALLOCATE` statement:

```
DEALLOCATE ProductCursor
```

The following section shows a complete example script that you may run using Query Analyzer. This script contains all five steps for using a cursor.

## Complete Example: *ProductCursor.sql*

[Listing 4.1](#) shows the `ProductCursor.sql` script. You can load this file into Query Analyzer and run it.

### Listing 4.1: USING CURSORS

---

```
/*
ProductCursor.sql uses a cursor to display
the ProductID, ProductName, and UnitPrice columns
from the Products table
*/
```

```
USE Northwind
```

```
-- step 1: declare the variables
DECLARE @MyProductID int
DECLARE @MyProductName nvarchar(40)
DECLARE @MyUnitPrice money

-- step 2: declare the cursor
DECLARE ProductCursor CURSOR FOR
SELECT ProductID, ProductName, UnitPrice
FROM Products
WHERE ProductID <= 10

-- step 3: open the cursor
OPEN ProductCursor

-- step 4: fetch the rows from the cursor
FETCH NEXT FROM ProductCursor
INTO @MyProductID, @MyProductName, @MyUnitPrice
PRINT '@MyProductID = ' + CONVERT(nvarchar, @MyProductID)
PRINT '@MyProductName = ' + CONVERT(nvarchar, @MyProductName)
PRINT '@MyUnitPrice = ' + CONVERT(nvarchar, @MyUnitPrice)
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM ProductCursor
    INTO @MyProductID, @MyProductName, @MyUnitPrice
    PRINT '@MyProductID = ' + CONVERT(nvarchar, @MyProductID)
    PRINT '@MyProductName = ' + CONVERT(nvarchar, @MyProductName)
    PRINT '@MyUnitPrice = ' + CONVERT(nvarchar, @MyUnitPrice)
END

-- step 5: close the cursor
CLOSE ProductCursor
DEALLOCATE ProductCursor
```

---

The output for the first two rows read by the cursor is as follows:

```
@MyProductID = 1
@MyProductName = Chai
@MyUnitPrice = 18.00
@MyProductID = 2
@MyProductName = Chang
@MyUnitPrice = 19.00
...
```

## Using Functions

SQL Server provides a number of functions you can use to get values from the database. For example, you can use the COUNT( ) function to get the number of rows in a table. The various functions are split into the categories shown in [Table 4.2](#).

**Table 4.2: FUNCTIONS**

FUNCTION CATEGORY	DESCRIPTION
Aggregate	Return information based on one or more rows in a table.
Mathematical	Perform calculations.
String	Perform string manipulations.
Date and time	Work with dates and times.
System	Return information on SQL Server.
Configuration	Return information on the configuration of the server.
Cursor	Return information on cursors.
Metadata	Return information on the database and the various database items, such as tables.
Security	Return information on the database users and roles.
System statistical	Return statistical information on SQL Server.
Text and image	Perform text and image manipulations.

You'll learn about the first five functions in the following sections. The other categories of functions are beyond the scope of this book, as they are of primary interest to database administrators. You can learn about those functions in the SQL Server Online Books documentation.

## Using Aggregate Functions

Earlier, you saw the use of the COUNT( ) aggregate function to get the number of rows. COUNT( ) and some other aggregate functions you can use with SQL Server are listed in [Table 4.3](#). The expression you may pass to the aggregate functions is typically a single column, but it can also be a calculated field. ALL means that the function is applied to all the column values, while DISTINCT means that the function is applied only to unique values. ALL is the default.

**Table 4.3: AGGREGATE FUNCTIONS**

FUNCTION	DESCRIPTION
AVG([ ALL   DISTINCT ] expression)	Returns the average of the values in a group.
COUNT([ ALL   DISTINCT ] expression   *)	Returns the number of rows in a group. COUNT( ) returns an int data type value.
COUNT_BIG([ ALL   DISTINCT ] expression   *)	Returns the number of values in a group. COUNT_BIG( ) returns a bigint data type value
MAX([ ALL   DISTINCT ] expression)	Returns the highest value.
MIN([ ALL   DISTINCT ] expression)	Returns the lowest value.
SUM([ ALL   DISTINCT ] expression)	Returns the sum of any non-null values. SUM( ) can be used only with numeric expressions.
STDEV(expression)	Returns the standard deviation for all the values.

<code>STDEVP(expression)</code>	Returns the standard deviation for the population of all the values.
<code>VAR(expression)</code>	Returns the variance for all the values.
<code>VARP(expression)</code>	Returns the variance for the population of all the values.

Let's consider examples that use some of the aggregate functions.

You use the `AVG()` function to get the average value. For example, the following statement gets the average of the `UnitPrice` column of the `Products` table using the `AVG()` function:

```
SELECT AVG(UnitPrice)
FROM Products;
```

This example returns 28.8663. Since `ALL` is the default used with functions, this example uses every row in the `Products` table when performing the calculation. If you wanted to just use unique values in the calculation, then you use the `DISTINCT` option, as shown in the following example:

```
SELECT AVG(DISTINCT UnitPrice)
FROM Products;
```

This example returns 31.4162, slightly higher than the previous result because only unique values are used this time.

In addition to passing a column to a function, you can also pass a calculated field. For example, the following statement passes the calculated field `UnitPrice * 1.20` to the `AVG()` function:

```
SELECT AVG(UnitPrice * 1.20)
FROM Products;
```

This example returns 34.639636; the average after the `UnitPrice` values have been increased 20 percent.

You can limit the rows passed to a function using a `WHERE` clause. For example, the following `SELECT` statement calculates the average `UnitPrice` value for the rows with a `CategoryID` of 1:

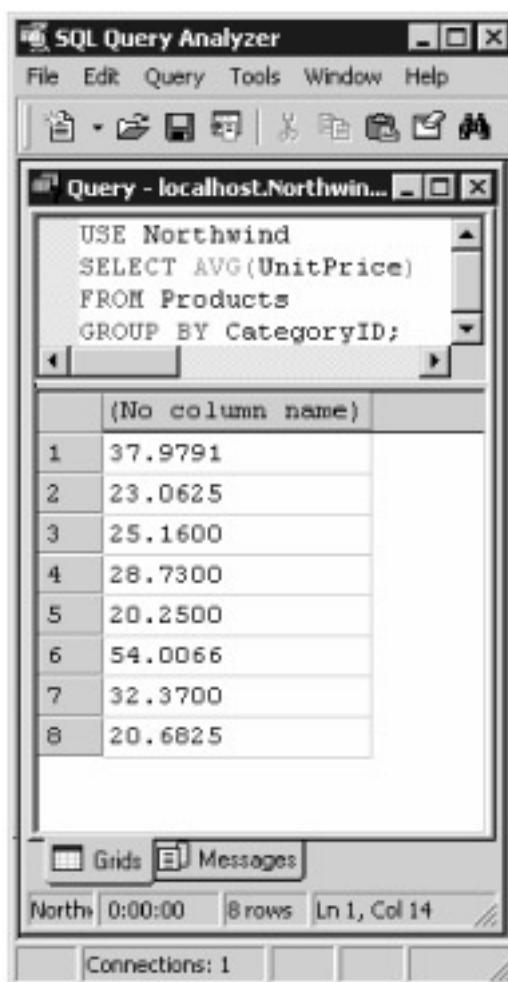
```
SELECT AVG(UnitPrice)
FROM Products
WHERE CategoryID = 1;
```

This example returns 37.9791.

You can combine a function with a `GROUP BY` clause to perform a calculation on each group of rows. For example, the following `SELECT` statement calculates the average `UnitPrice` value for each block of rows grouped by `CategoryID`:

```
SELECT AVG(UnitPrice)
FROM Products
GROUP BY CategoryID;
```

[Figure 4.2](#) shows the results of this `SELECT` statement.



**Figure 4.2:** Using the `AVG()` function to compute the average value of the `UnitPrice` column

You can also supply a `HAVING` clause to eliminate groups used in a `SELECT` statement. For example, the following statement adds a `HAVING` clause to the previous example to eliminate the groups that have an average value greater than 50:

```
SELECT AVG(UnitPrice)
FROM Products
GROUP BY CategoryID
HAVING AVG(UnitPrice) > 50;
```

This example returns 54.0066.

Let's take a look at some of the other aggregate functions. You get the total number of rows using the `COUNT()` function. For example, the following statement gets the total number of rows in the `Products` table using the `COUNT()` function:

```
SELECT COUNT(*)
FROM Products;
```

This example returns 77.

You use the `MAX()` and `MIN()` functions to get the maximum and minimum values. For example, the following statement uses these functions to get the maximum and minimum `UnitPrice`:

```
SELECT MAX(UnitPrice), MIN(UnitPrice)
FROM Products;
```

This example returns 263.5000 and 2.5000 for the respective maximum and minimum values.

You use the `SUM()` function to get the total of any non-null values. For example, the following statement gets the sum of the `UnitPrice` column values for each group of rows using the `SUM()` function:

```
SELECT SupplierID, SUM(UnitPrice) AS SumUnitPrice
FROM Products
```

```
GROUP BY SupplierID;
```

The GROUP BY clause of this example returns one row for each block of rows with identical SupplierID column values. The SUM( ) function then adds up the UnitPrice column values for all the rows within each block and returns a single value. For example, SUM( ) returns 47.0000 for the group where the SupplierID is 1. This is the sum of the UnitPrice column values for all the rows where the SupplierID is 1. Similarly, SUM( ) returns 81.4000 where the SupplierID is 2, and so on. The AS clause in this example names the results returned by the SUM( ) function as SumUnitPrice.

[Figure 4.3](#) shows the results of this SELECT statement.

The screenshot shows the SQL Query Analyzer interface. The query window contains the following T-SQL code:

```
USE Northwind
SELECT SupplierID, SUM(UnitPrice) AS SumUnitPrice
FROM Products
GROUP BY SupplierID;
```

The results grid displays the following data:

	SupplierID	SumUnitPrice
1	1	47.0000
2	2	81.4000
3	3	95.0000
4	4	138.0000
5	5	59.0000
6	6	44.7500
7	7	177.8500
8	8	112.7000
9	9	30.0000
10	10	4.5000
11	11	89.1300
12	12	223.3900
13	13	25.8900
14	14	79.3000

The status bar at the bottom shows the following information: Query batch: localhost (8.0), sa (51), Northwind, 0:00:00, 29 rows, Ln 4, Col 21. The 'Grids' tab is selected in the bottom-left corner.

**Figure 4.3:** Using the SUM( ) function to compute the total of the UnitPrice column

## Using Mathematical Functions

The mathematical functions allow you to perform numerical operations, such as getting the absolute value of a number. [Table 4.4](#) lists the mathematical functions available in SQL Server. The expression you may pass to the mathematical functions is typically a single column or value, but it can also be a calculated field.

**Table 4.4: MATHEMATICAL FUNCTIONS**

FUNCTION	DESCRIPTION
ABS( <i>expression</i> )	Returns the absolute value of <i>expression</i> . This is always a positive number.

<code>ACOS(expression)</code>	Returns the arccosine of <i>expression</i> .
<code>ASIN(expression)</code>	Returns the arcsine of <i>expression</i> .
<code>ATAN(expression)</code>	Returns the arctangent of <i>expression</i> .
<code>ATN2(expression1, expression2)</code>	Returns the arctangent of the angle between <i>expression1</i> and <i>expression2</i> .
<code>CEILING(expression)</code>	Returns the smallest integer greater than or equal to <i>expression</i> .
<code>COS(expression)</code>	Returns the cosine of <i>expression</i> .
<code>COT(expression)</code>	Returns the cotangent of <i>expression</i> .
<code>DEGREES(expression)</code>	Converts the supplied angle in radians to an angle in degrees.
<code>EXP(expression)</code>	Returns the exponential value of <i>expression</i> .
<code>FLOOR(expression)</code>	Returns the largest integer less than or equal to <i>expression</i> .
<code>LOG(expression)</code>	Returns the natural logarithm of <i>expression</i> .
<code>LOG10(expression)</code>	Returns the base-10 logarithm of <i>expression</i> .
<code>PI()</code>	Returns the mathematical constant Pi.
<code>POWER(expression, y)</code>	Returns the value of <i>expression</i> raised to the power <i>y</i> .
<code>RADIANS(expression)</code>	Converts the supplied angle in degrees to an angle in radians.
<code>RAND([expression])</code>	Returns a random floating-point number between 0 and 1. The <i>expression</i> is an optional seed value that you may use to generate the random number.
<code>ROUND(expression, length [, function])</code>	Returns the value of <i>expression</i> rounded or truncated to the number of decimal places specified by <i>length</i> . The optional <i>function</i> is used to specify the type of operation to perform: 0 (the default) rounds the number, and any other value truncates the number.
<code>SIGN(expression)</code>	Returns 1, 0, or -1 depending on the sign of <i>expression</i> . Returns 1 for a positive number, 0 for zero, or -1 for a negative number.
<code>SIN(expression)</code>	Returns the sine of <i>expression</i> .
<code>SQUARE(expression)</code>	Returns the square of <i>expression</i> .
<code>SQRT(expression)</code>	Returns the square root of <i>expression</i> .
<code>TAN(expression)</code>	Returns the tangent of <i>expression</i> .

Let's consider examples that use some of the mathematical functions. You use the `ABS()` function to get the absolute value. The following example returns 10 and 15:

```
SELECT ABS(-10), ABS(15);
```

You use the `ACOS()`, `ASIN()`, and `ATAN()` functions to get the arccosine, arcsine, and arctangent of a number. The following example returns 0.0, 1.5707963267948966, and 0.78539816339744828:

```
SELECT ACOS(1), ASIN(1), ATAN(1);
```

You use the `CEILING()` function to get the smallest integer greater than or equal to the value passed to it. The following example returns

## Using Functions

2 and -1:

```
SELECT CEILING(1.4), CEILING(-1.4);
```

You use the FLOOR( ) function to get the largest integer less than or equal to the value passed to it. The following example returns 1 and -2:

```
SELECT FLOOR(1.4), FLOOR(-1.4);
```

You use the PI( ) function to get the mathematical constant Pi. The following example returns 3.1415926535897931:

```
SELECT PI();
```

You use the POWER( ) function to get the value of a number raised to a specified power. The following example returns 8:

```
SELECT POWER(2, 3);
```

You use the ROUND( ) function to get the value of a number rounded or truncated to a specified length. The following example returns 1.23500, which is 1.23456 rounded to three decimal places:

```
SELECT ROUND(1.23456, 3);
```

The next example passes a non-zero number as the third parameter to ROUND( ), which indicates that the number is to be truncated rather than rounded, as was done in the previous example:

```
SELECT ROUND(1.23456, 3, 1);
```

This example returns 1.23400, which is 1.23456 truncated to three decimal places.

You use the SQUARE( ) function to get the square of a number. The following example returns 16.0:

```
SELECT SQUARE(4);
```

You use the SQRT( ) function to get the square root of a number. The following example returns 4.0:

```
SELECT SQRT(16);
```

## Using String Functions

The string functions allow you to manipulate strings. For example, you can replace specified characters in a string. [Table 4.5](#) lists the string functions available in SQL Server.

**Table 4.5: STRING FUNCTIONS**

FUNCTION	DESCRIPTION
ASCII( <i>charExpression</i> )	Returns the ASCII code for the leftmost character of <i>charExpression</i> .
CHAR( <i>intExpression</i> )	Returns the character that corresponds to the ASCII code specified by <i>intExpression</i> .
CHARINDEX ( <i>charExpression1</i> , <i>charExpression2</i> [, <i>start</i> ])	Returns the position of the characters specified by <i>charExpression1</i> in <i>charExpression2</i> , starting at the optional position specified by <i>start</i> .
DIFFERENCE ( <i>charExpression1</i> , <i>charExpression2</i> )	Returns the difference between the SOUNDEX values of the two character expressions. You use the SOUNDEX code to evaluate the phonetic similarity of two strings. The returned value is between 0 and 4; 4 indicates that the two expressions are phonetically identical.
LEFT( <i>charExpression</i> , <i>intExpression</i> )	Returns the leftmost characters specified by <i>intExpression</i> from <i>charExpression</i> .

<code>LEN(<i>charExpression</i>)</code>	Returns the number of characters in <i>charExpression</i> .
<code>LOWER(<i>charExpression</i>)</code>	Converts the characters in <i>charExpression</i> to lowercase and returns those characters.
<code>LTRIM(<i>charExpression</i>)</code>	Removes any leading spaces from the start of <i>charExpression</i> and returns the remaining characters.
<code>NCHAR(<i>intExpression</i>)</code>	Returns the Unicode character with the code specified by <i>intExpression</i> .
<code>PATINDEX('pattern%', <i>charExpression</i>)</code>	Returns the starting position of the first occurrence of pattern in <i>charExpression</i> . If pattern is not found then zeros are returned.
<code>REPLACE (<i>charExpression1</i>, <i>charExpression2</i>, <i>charExpression3</i>)</code>	Replaces all occurrences of <i>charExpression2</i> in <i>charExpression1</i> with <i>charExpression3</i> .
<code>QUOTENAME ('<i>charString</i>' [ , '<i>quoteChar</i>' ])</code>	Returns a Unicode string with the delimiters specified by <i>quoteChar</i> added to make <i>charString</i> a valid delimited identifier.
<code>REPLICATE (<i>charExpression</i>, <i>intExpression</i>)</code>	Repeats <i>charExpression</i> a total of <i>intExpression</i> times.
<code>REVERSE(<i>charExpression</i>)</code>	Reverses the characters in <i>charExpression</i> and returns those characters.
<code>RIGHT(<i>charExpression</i>, <i>intExpression</i>)</code>	Returns the rightmost characters specified by <i>intExpression</i> from <i>charExpression</i> .
<code>RTRIM(<i>charExpression</i>)</code>	Removes any trailing spaces from the end of <i>charExpression</i> and returns the remaining characters.
<code>SOUNDEX(<i>charExpression</i>)</code>	Returns the four-character SOUNDEX code. You use this code to evaluate the phonetic similarity of two strings.
<code>SPACE(<i>intExpression</i>)</code>	Returns a string of repeated spaces for a total specified by <i>intExpression</i> .
<code>STR(<i>floatExpression</i> [ , <i>length</i> [ , <i>decimal</i> ] ])</code>	Converts the number specified by <i>floatExpression</i> to characters; <i>length</i> specifies the total number of characters you want to see (including digits and spaces, plus the positive or negative sign and decimal point); <i>decimal</i> specifies the number of digits to the right of the decimal point. The number is rounded if necessary.
<code>STUFF (<i>charExpression1</i>, <i>start</i>, <i>length</i>, <i>charExpression2</i>)</code>	Deletes characters from <i>charExpression1</i> , starting at the position specified by <i>start</i> for a total of <i>length</i> characters, and then inserts the characters specified by <i>charExpression2</i> .
<code>SUBSTRING(<i>expression</i>, <i>start</i>, <i>length</i>)</code>	Returns part of a character, binary, text, or image <i>expression</i> .
<code>UNICODE('nCharExpression')</code>	Returns the Unicode value for the first character of the nchar or nvarchar expression <i>nCharExpression</i> .
<code>UPPER(<i>charExpression</i>)</code>	Converts the characters in <i>charExpression</i> to uppercase and returns those characters.

## Using Functions

Let's consider examples that use some of the string functions.

You use the `ASCII( )` function to get the ASCII code for the leftmost character of the supplied character expression. The following example returns 65 and 97:

```
SELECT ASCII('A'), ASCII('a');
```

You use the `CHAR( )` function to get the character that corresponds to the ASCII code of the supplied integer expression. The following example returns A and a:

```
SELECT CHAR(65), CHAR(97);
```

You use the `CHARINDEX( )` function to get the position of characters. The following example returns 16, which is the position where the word `ten` starts:

```
SELECT CHARINDEX('ten', 'Four-score and ten years');
```

You use the `DIFFERENCE( )` function to obtain the difference between the `SOUNDEX` values of two character expressions. The following example returns 4, indicating that `Brown` and `Browne` are phonetically identical:

```
SELECT DIFFERENCE('Brown', 'Browne');
```

You use the `LEFT( )` function to obtain the leftmost characters of a character expression. The following example returns `Four-score`, which are the 10 leftmost characters of `Four-score and ten years`:

```
SELECT LEFT('Four-score and ten years', 10);
```

You use the `RIGHT( )` function to obtain the rightmost characters of a character expression. The following example returns `years`, which are the five rightmost characters of `Four-score and ten years`:

```
SELECT RIGHT('Four-score and ten years', 5);
```

You use the `LEN( )` function to obtain the digits in a character expression. The following example returns 24:

```
SELECT LEN('Four-score and ten years');
```

You use the `LOWER( )` function to obtain the lowercase version of a character expression. The following example returns `four-score and ten years`:

```
SELECT LOWER('FOUR-SCORE AND TEN YEARS');
```

You use the `UPPER( )` function to obtain the uppercase version of a character expression. The following example returns `FOUR-SCORE AND TEN YEARS`:

```
SELECT UPPER('four-score and ten years');
```

You use the `LTRIM( )` and `RTRIM( )` functions to remove any spaces from the left and right of a character expression. The following example returns `FOUR-SCORE and AND TEN YEARS` (spaces removed):

```
SELECT LTRIM(' FOUR-SCORE'), RTRIM('AND TEN YEARS');
```

You use the `STR( )` function to convert a numeric value to a string consisting of numbers. The first parameter is the number to convert, the second is the total number of characters you want in your string, and the third is the number of digits after the decimal point. The following example returns 123.46:

```
SELECT STR(123.456, 6, 2);
```

The number 123.456 is converted to a string of six characters, with two digits after the decimal point, and rounded.

You use the `STUFF( )` function to replace characters. The first parameter is the string you want to replace characters in, the second is the starting position, the third is the total number of characters, and the fourth is the set of characters to insert. The following example returns `Five-score and ten`:

```
SELECT STUFF('Four-score and ten', 1, 4, 'Five');
```

Four is replaced with Five.

You use the SUBSTRING( ) function to obtain part of a string. The first parameter is the string, the second is the starting position, and the third is the total number of characters. The following example returns Four:

```
SELECT SUBSTRING('Four-score and ten', 1, 4);
```

You use the UNICODE( ) function to obtain the Unicode value for the first character. The following example returns 65 and 97:

```
SELECT UNICODE('A'), UNICODE('a');
```

## Using Date and Time Functions

The date and time functions allow you to manipulate dates and times. For example, you can add a number of days to a given date. [Table 4.6](#) lists the date and time functions available in SQL Server.

**Table 4.6: DATE AND TIME FUNCTIONS**

FUNCTION	DESCRIPTION
DATEADD( <i>interval</i> , <i>number</i> , <i>date</i> )	Returns a datetime that is the result of adding the specified <i>number</i> of <i>interval</i> units to <i>date</i> . Valid intervals include year, quarter, month, dayofyear, day, week, hour, minute, second, and millisecond.
DATEDIFF ( <i>interval</i> , <i>startDate</i> , <i>endDate</i> )	Returns the difference between <i>startDate</i> and <i>endDate</i> , with the difference calculated in <i>interval</i> units (year, quarter, and so on).
DATENAME( <i>interval</i> , <i>date</i> )	Returns a character string that represents the name of <i>interval</i> part of <i>date</i> .
DATEPART( <i>interval</i> , <i>date</i> )	Returns an integer that represents the <i>interval</i> part of <i>date</i> .
DAY( <i>date</i> )	Returns an integer that represents the day part of <i>date</i> .
GETDATE()	Returns a datetime containing the current system date.
GETUTCDATE()	Returns a datetime containing the current system date as UTC time (Universal Time Coordinate or Greenwich Mean Time). The UTC time is derived from the current local time and the system time-zone setting.
MONTH( <i>date</i> )	Returns an integer that represents the month part of <i>date</i> .
YEAR( <i>date</i> )	Returns an integer that represents the year part of <i>date</i> .

Let's consider examples that use some of the date and time functions.

You use the DATEADD( ) function to add a number of intervals to a date. The following example adds two days to the date 12/20/2003 and returns 2003-12-22 00:00:00.000:

```
SELECT DATEADD(day, 2, '12/20/2003');
```

You use the DATEDIFF( ) function to obtain the difference between two dates. The following example obtains the difference between 12/20/2003 and 12/22/2003 in days and returns 2 days:

```
SELECT DATEDIFF(day, '12/20/2003', '12/22/2003');
```

You use the DATENAME( ) method to obtain a character string that represents the interval part of a date. The following example gets the month name of 12/20/2003 and returns December:

## Using Functions

```
SELECT DATENAME(month, '12/20/2003');
```

You use the DATEPART( ) method to obtain an integer that represents the interval part of a date. The following example gets the month number of 12/20/2003 and returns 12:

```
SELECT DATEPART(month, '12/20/2003');
```

You use the DAY( ) function to obtain an integer that represents the day part of a date. The following example gets the day number of 12/20/2003 and returns 20:

```
SELECT DAY('12/20/2003');
```

You use the MONTH( ) function to obtain an integer that represents the month part of a date. The following example gets the month number of 12/20/2003 and returns 12:

```
SELECT MONTH('12/20/2003');
```

You use the YEAR( ) function to obtain an integer that represents the year part of a date. The following example gets the year number of 12/20/2003 and returns 2003:

```
SELECT YEAR('12/20/2003');
```

You use the GETDATE( ) function to obtain the current system date. The following example returns 2002-07-16 12:59:50.823:

```
SELECT GETDATE();
```

You use the GETUTCDATE( ) function to obtain the current system date as UTC time. The following example returns 2002-07-16 20:02:18.123:

```
SELECT GETUTCDATE();
```

## Using System Functions

The system functions allow you to manipulate and obtain information about values, objects, and settings in SQL Server. For example, you can convert a value in one type to another type. [Table 4.7](#) lists some of the system functions available in SQL Server.

**Table 4.7: SYSTEM FUNCTIONS**

FUNCTION	DESCRIPTION
<code>CONVERT(dataType expression [, style [(length )], ])</code>	<p>Converts the value in <i>expression</i> to the type specified by <i>dataType</i>. If you are converting to an <i>nchar</i>, <i>nvarchar</i>, <i>char</i>, <i>varchar</i>, <i>binary</i>, or <i>varbinary</i> type, you can also specify an optional <i>length</i>, which specifies the length of the new value. You can use the optional <i>style</i> when</p> <ul style="list-style-type: none"><li>• Converting <i>datetime</i> or <i>smalldatetime</i> data to character data; <i>style</i> is the format for the date and time.</li><li>• Converting <i>float</i>, <i>real</i>, <i>money</i>, or <i>smallmoney</i> data to character data; <i>style</i> is the string format for the number. You can look up the details for <i>style</i> option in the SQL Server Books Online documentation.</li></ul>
<code>COALESCE(expression1 [ , ... expressionN] )</code>	Returns the first non-null expression in the list of expressions.
<code>DATALENGTH(expression) .</code>	Returns the number of bytes used to represent <i>expression</i> .

<code>@@ERROR</code>	Returns the error number for the last T-SQL statement that was executed.
<code>@@IDENTITY</code>	Returns the last inserted identity value.
<code>ISDATE(expression)</code>	Returns 1 when <i>expression</i> is a valid date, otherwise 0 is returned.
<code>ISNULL(expression, replacementValue)</code>	If <i>expression</i> is null, then <i>replacementValue</i> is returned, otherwise <i>expression</i> is returned.
<code>ISNUMERIC(expression)</code> .	Returns 1 when <i>expression</i> is a valid number, otherwise 0 is returned.
<code>NEWID()</code>	Returns a unique value of the <code>uniqueidentifier</code> type.
<code>NULLIF(expression1, expression2)</code>	Returns a null if <i>expression1</i> equals <i>expression2</i> .
<code>@@ROWCOUNT</code>	Returns the number of rows affected by the last T-SQL statement that was executed.
<code>@@TRANCOUNT</code>	Returns the number of active transactions for the current connection to the database.

Let's consider examples that use some of the system functions.

You use the `CONVERT( )` function to convert a value from one type to another. The following example converts the number 123.456 to an `nvarchar` and returns 123.456:

```
SELECT CONVERT(nvarchar, 123.456);
```

You use the `COALESCE( )` function to obtain the first non-null expression in a list. The following example returns 123.456:

```
SELECT COALESCE(null, null, 123.456, null);
```

You use the `DATALENGTH( )` function to obtain the number of bytes used to represent an expression. The following example displays the number of bytes used to represent the value stored in the `CompanyName` column of the `Customers` table for the row where `CustomerID` equals ALFKI:

```
SELECT DATALENGTH(CompanyName), CompanyName
FROM Customers
WHERE CustomerID = 'ALFKI';
```

This example returns 38 and `Alfreds Futterkiste`, which contains 19 letters. Each letter is stored in 2 bytes, and the 19-letter string therefore takes up 38 bytes ( $2 * 19$ ).

You use the `ISDATE( )` function to determine if an expression is a valid date. `ISDATE( )` returns 1 when the expression is a valid date, otherwise it returns 0. The following example returns 1 and 0:

```
SELECT ISDATE('12/20/2004'), ISDATE(1234);
```

You use the `ISNUMERIC( )` function to determine if an expression is a valid number. The following example returns 1 and 0:

```
SELECT ISNUMERIC(1234), ISNUMERIC('abc');
```

You use the `ISNULL( )` function to replace a null value with another value. The following example returns 10 and 20:

```
SELECT ISNULL(null, 10), ISNULL(20, 10);
```

## Creating User-Defined Functions

You can create your own user-defined functions in SQL Server. For example, you might want to create your own function to compute the discounted price given the original price and factor to multiply that price by. You create a function using the `CREATE FUNCTION` statement. There are three types of userdefined functions:

- **Scalar functions** Scalar functions return a single value. The returned value can be of any data type except `text`, `ntext`, `image`, `cursor`, `table`, `timestamp`, and user-defined data types.
- **Inline table-valued functions** Inline table-valued functions return an object of the `table` type. You can think of a `table` as a regular database table, except it is stored in memory. An inline table-valued function can return the results retrieved by only a single `SELECT` statement.
- **Multistatement table-valued functions** Multistatement table-valued functions return an object of the `table` type. Unlike an inline table-valued function, a multistatement table-valued function can contain multiple T-SQL statements.

You'll see examples of these three types of functions in the following sections.

### Using Scalar Functions

Scalar functions return a single value. [Listing 4.2](#) shows the `DiscountPrice.sql` script that creates the `DiscountPrice()` function, which returns the original price of an item multiplied by a discount factor. These values are passed as parameters to the `DiscountPrice()` function. You can load this file into Query Analyzer and run it.

#### [Listing 4.2: DISCOUNTPRICE.SQL](#)

---

```
/*
DiscountPrice.sql creates a scalar function to
return the new price of an item given the original
price and a discount factor
*/
CREATE FUNCTION DiscountPrice(@OriginalPrice money, @Discount float)
RETURNS money
AS
BEGIN
    RETURN @OriginalPrice * @Discount
END
```

---

The parameters to the function are placed in brackets after the name of the function in the `CREATE FUNCTION` statement.

**Warning** Make sure you select the Northwind database from the drop-down list box on the Query Analyzer toolbar before running the script. That way, the function is created in the Northwind database.

You can also create functions using Enterprise Manager. You do this by clicking the right mouse button on the User Defined Functions node in the Databases folder and selecting New User Defined Function. You can then cut and paste the contents of `DiscountPrice.sql` into the Enterprise Manager properties dialog box, as shown in [Figure 4.4](#).



**Figure 4.4:** Using Enterprise Manager to define a function

You can view and modify a function by double-clicking the function name in Enterprise Manager. You can also delete a function using Enterprise Manager. The Object Browser of Query Analyzer allows you to view, modify, and delete functions as well.

**Tip** You can also delete a function using the *DROP FUNCTION* statement, and you can modify a function using the *ALTER FUNCTION* statement.

Once you've created the function, you can call it. When calling a scalar function, you use the following syntax:

*owner.functionName*

Where *owner* is the database user who owns the function, and *functionName* is the name of the function.

Let's say you created the *DiscountPrice()* function using the *dbo* user, then you call that function using *dbo.DiscountPrice()*. The following example returns 3.0000, which is  $10 * 0.3$ :

```
SELECT dbo.DiscountPrice(10, 0.3);
```

As with any other function, you can pass a column to *DiscountPrice()*. The following example returns 5.4000 and 18.0000; 5.4000 is  $18.0000 * 0.3$ :

```
SELECT dbo.DiscountPrice(UnitPrice, 0.3), UnitPrice
FROM Products
WHERE ProductID = 1;
```

You can of course also pass variables as parameters to a function. As before, this example returns 5.4000 and 18.0000:

```
DECLARE @MyDiscountFactor float
SET @MyDiscountFactor = 0.3
SELECT dbo.DiscountPrice(UnitPrice, @MyDiscountFactor), UnitPrice
FROM Products
WHERE ProductID = 1;
```

## Using Inline Table-Valued Functions

An inline table-valued function returns an object of the *table* type, which is populated using a single *SELECT* statement. Unlike a scalar function, an inline table-valued function doesn't contain a body of statements placed within *BEGIN* and *END* statements. Instead, only a single *SELECT* statement is placed within the function.

For example, [Listing 4.3](#) shows the *ProductsToBeReordered.sql* script that creates the *ProductsToBeReordered()* function. This function returns a *table* containing the rows from the *Products* table with a *UnitsInStock* column value less than

or equal to the reorder level parameter passed to the function.

### **Listing 4.3: PRODUCTSTOBEREORDERED.SQL**

---

```
/*
ProductsToBeReordered.sql creates an inline table-valued function to
return the rows from the Products table whose UnitsInStock column
is less than or equal to the reorder level passed as a parameter
to the function
*/
CREATE FUNCTION ProductsToBeReordered(@ReorderLevel int)
RETURNS table
AS
RETURN
(
    SELECT *
    FROM Products
    WHERE UnitsInStock <= @ReorderLevel
)
```

---

Unlike a scalar function, you don't have to add the owner when calling an inline table-valued function. You use a `SELECT` statement to read the table returned by the function as you would any other table. For example, the following `SELECT` statement displays all the rows and columns returned by the function call `ProductsToBeReordered(10)`:

```
SELECT *
FROM ProductsToBeReordered(10);
```

You can of course also display only selected columns and rows from the `table` returned by an inline table-valued function. For example:

```
SELECT ProductID, ProductName, UnitsInStock
FROM ProductsToBeReordered(10)
WHERE ProductID <= 50;
```

[Figure 4.5](#) shows the results of this `SELECT` statement.



**Figure 4.5:** Using an inline table-valued function

## Using Multistatement Table-Valued Functions

Multistatement table-valued functions return an object of the `table` type. Unlike an inline table-valued function, a multistatement table-valued function can contain multiple T-SQL statements, and allow you to build complex functions.

For example, [Listing 4.4](#) shows the `ProductsToBeReordered2.sql` script that creates the `ProductsToBeReordered2()` function. This function returns a table containing the `ProductID`, `ProductName`, and `UnitsInStock` columns from the `Products` table with a `UnitsInStock` column value less than or equal to the reorder level parameter. In addition, a new column named `Reorder` is added to the table, which contains the word `Yes` or `No`, depending on whether the product must be reordered.

### Listing 4.4: PRODUCTSTOBEREORDERED2.SQL

```
/*
ProductsToBeReordered2.sql creates an inline table-valued
function that returns the rows from the Products table
whose UnitsInStock column is less than or equal to the
reorder level passed as a parameter to the function
*/
CREATE FUNCTION ProductsToBeReordered2(@ReorderLevel int)
RETURNS @MyProducts table
(
    ProductID int,
    ProductName nvarchar(40),
    UnitsInStock smallint,
    Reorder nvarchar(3)
)
AS
```

BEGIN

```
-- retrieve rows from the Products table and
-- insert them into the MyProducts table,
-- setting the Reorder column to 'No'
INSERT INTO @MyProducts
    SELECT ProductID, ProductName, UnitsInStock, 'No'
    FROM Products;

-- update the MyProducts table, setting the
-- Reorder column to 'Yes' when the UnitsInStock
-- column is less than or equal to @ReorderLevel
UPDATE @MyProducts
SET Reorder = 'Yes'
WHERE UnitsInStock <= @ReorderLevel
```

RETURN

END

As with an inline table-valued function, you don't have to add the owner when calling an inline table-valued function. You use a `SELECT` statement to read the table returned by the function as you would any other regular database table. For example, the following `SELECT` statement displays all the rows and columns returned by the function call `ProductsToBeReordered2( 20 )`:

```
SELECT *
FROM ProductsToBeReordered2( 20 );
```

[Figure 4.6](#) shows the results of this `SELECT` statement.



The screenshot shows the SQL Query Analyzer interface. The query window contains the following code:

```
USE Northwind
SELECT *
FROM ProductsToBeReordered2( 20 );
```

The results grid displays 15 rows of data from the function's output:

	ProductID	ProductName	UnitsInStock	Reorder
1	1	Chai	39	No
2	2	Chang	17	Yes
3	3	Aniseed Syrup	13	Yes
4	4	Chef Anton's Cajun Seasoning	53	No
5	5	Chef Anton's Gumbo Mix	0	Yes
6	6	Grandma's Boysenberry Spread	120	No
7	7	Uncle Bob's Organic Dried Pears	15	Yes
8	8	Northwoods Cranberry Sauce	6	Yes
9	9	Mishi Kobe Niku	29	No
10	10	Ikura	31	No
11	11	Queso Cabrales	22	No
12	12	Queso Manchego La Pastora	86	No
13	13	Kobe	24	No
14	14	Tofu	35	No
15	15	Genen Shoyu	29	No

**Figure 4.6:** Using a multistatement table-valued function

In the [next section](#), you'll learn how to use stored procedures.

## Introducing Stored Procedures

SQL Server allows you to store procedures in a database. Stored procedures differ from user-defined functions in that procedures can return a much wider array of data types.

You'll typically create a stored procedure when you need to perform a task that intensively uses the database, or you want to centralize code in the database that any user can call rather than have each user write their own program to perform the same task. One example of intensive database use is a banking application by which you need to update accounts at the end of each day. One example of when you'd use centralized code is when you want to restrict user access to database tables: you might want users to be able to add a row to a table only through a procedure so that no mistakes are made.

In this section, you'll learn how to create a stored procedure in the Northwind database and run it using the Query Analyzer tool.

### Creating a Stored Procedure

The procedure you'll see in this section is named `AddProduct()`. This procedure adds a row to the `Products` table, setting the column values for the new row to those passed as parameters to the procedure.

The `ProductID` column for the new row is assigned a value automatically by the database through the use of an identity that was set up when the table was originally created. This identity value may be read using the `@@IDENTITY` function after the new row is added to the table. The `AddProduct()` procedure you'll see here returns that identity value to the calling statement.

You create a procedure using the `CREATE PROCEDURE` statement, and [Listing 4.5](#) shows the `AddProduct.sql` script that creates the `AddProduct()` procedure.

#### **Listing 4.5: ADDPRODUCT.SQL**

---

```
/*
AddProduct.sql creates a procedure that adds a row to the
Products table using values passed as parameters to the
procedure. The procedure returns the ProductID of the new row.
*/
CREATE PROCEDURE AddProduct
    @MyProductName nvarchar(40),
    @MySupplierID int,
    @MyCategoryID int,
    @MyQuantityPerUnit nvarchar(20),
    @MyUnitPrice money,
    @MyUnitsInStock smallint,
    @MyUnitsOnOrder smallint,
    @MyReorderLevel smallint,
    @MyDiscontinued bit
AS
DECLARE @ProductID int

-- insert a row into the Products table
INSERT INTO Products (
    ProductName, SupplierID, CategoryID, QuantityPerUnit,
    UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel,
    Discontinued
) VALUES (
    @MyProductName, @MySupplierID, @MyCategoryID, @MyQuantityPerUnit,
    @MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder, @MyReorderLevel,
    @MyDiscontinued
)
-- use the @@IDENTITY function to get the last inserted
```

```
-- identity value, which in this case is the ProductID of
-- the new row in the Products table
SET @ProductID = @@IDENTITY

-- return the ProductID
RETURN @ProductID
```

You can also create procedures using Enterprise Manager. You do this by clicking the right mouse, button on the Stored Procedures node in the Databases folder and selecting New Stored Procedure. You can then cut and paste the contents of `AddProduct.sql` into the Enterprise Manager properties dialog box, as shown in [Figure 4.7](#). You'll notice I've added some comments to the start of the file that indicate what the procedure does.



**Figure 4.7:** Using Enterprise Manager to define a procedure

You can view and modify a procedure by double-clicking the procedure name in Enterprise Manager. You can also delete a procedure using Enterprise Manager. The Object Browser of Query Analyzer allows you to view, modify, and delete procedures as well.

**Tip** You can also delete a procedure using the `DROP PROCEDURE` statement, and you can modify a procedure using the `ALTER PROCEDURE` statement.

In the [next section](#), you'll see how to run a stored procedure.

## Running a Stored Procedure

You run a procedure using the `EXECUTE` statement. For example, the following statements run the `AddProduct()` procedure:

```
DECLARE @MyProductID int
EXECUTE @MyProductID = AddProduct 'Widget', 1, 1, '1 Per box', 5.99, 10, 5, 5, 1
PRINT @MyProductID
```

With the initial set of rows in the `Products` table, the next identity value generated by SQL Server for the `ProductID` is 78, which is the value displayed by the previous example if you run it.

You can of course also pass variables as parameters to a procedure. The following example displays 79—the next `ProductID`:

```
DECLARE @MyProductID int
DECLARE @MyProductName nvarchar(40)
DECLARE @MySupplierID int
DECLARE @MyCategoryID int
```

## Introducing Stored Procedures

```
DECLARE @MyQuantityPerUnit nvarchar(20)
DECLARE @MyUnitPrice money
DECLARE @MyUnitsInStock smallint
DECLARE @MyUnitsOnOrder smallint
DECLARE @MyReorderLevel smallint
DECLARE @MyDiscontinued bit

SET @MyProductName = 'Wheel'
SET @MySupplierID = 2
SET @MyCategoryID = 1
SET @MyQuantityPerUnit = '4 per box'
SET @MyUnitPrice = 99.99
SET @MyUnitsInStock = 10
SET @MyUnitsOnOrder = 5
SET @MyReorderLevel = 5
SET @MyDiscontinued = 0

EXECUTE @MyProductID = AddProduct @MyProductName,
    @MySupplierID, @MyCategoryID, @MyQuantityPerUnit,
    @MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder,
    @MyReorderLevel, @MyDiscontinued

PRINT @MyProductID
```

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Introducing Triggers

A database *trigger* is a special kind of stored procedure that is run automatically by the database—or in trigger terms, *fired*—after a specified `INSERT`, `UPDATE`, or `DELETE` statement is run against a specified database table. Triggers are very useful for doing things such as auditing changes made to column values in a table.

A trigger can also fire instead of an `INSERT`, `UPDATE`, or `DELETE`. For example, instead of performing an `INSERT` to add a row to the `Products` table, a trigger could raise an error if a product with the same `ProductID` already existed in the table.

As mentioned, triggers are very useful for auditing changes made to column values. In this section, you'll see an example of a trigger that will audit changes made to the `Products` table.

Also, when an `UPDATE` statement modifies the `UnitPrice` column of a row in the `Products` table, a row will be added to the `ProductAudit` table. Finally, when a `DELETE` statement removes a row from the `Products` table, a row will be added to the `ProductAudit` table.

Before you see the triggers, you'll need to create the `ProductAudit` table. Listing 4.6 shows the `ProductAudit.sql` script that creates the `ProductAudit` table.

### **Listing 4.6: PRODUCTAUDIT.SQL**

---

```
/*
ProductAudit.sql creates a table that is used to
store the results of triggers that audit modifications
to the Products table
*/
USE Northwind

CREATE TABLE ProductAudit (
    ID int IDENTITY(1, 1) PRIMARY KEY,
    Action nvarchar(100) NOT NULL,
    PerformedBy nvarchar(15) NOT NULL DEFAULT User,
    TookPlace datetime NOT NULL DEFAULT GetDate()
)
```

---

The `IDENTITY` clause creates an identity for the `ID` primary key column of the `ProductAudit` table. An identity automatically generates values for a column. The identity for the `ID` column starts with the value 1, which is incremented by 1 after each `INSERT`. The `Action` column stores a string that records the action performed, for example, 'Product added with ProductID of 80'. The `PerformedBy` column stores the name of the user who performed the action; this is defaulted to `User`, which returns the current user. The `TookPlace` column stores the date and time when the action took place; this is defaulted using the `GetDate()` function, which returns the current date and time.

In the following sections, you'll learn how to create and use the following triggers:

- **InsertProductTrigger** Fires after an `INSERT` statement is performed on the `Products` table.
- **UpdateUnitPriceProductTrigger** Fires after an `UPDATE` statement is performed on the `Products` table.
- **DeleteProductTrigger** Fires after a `DELETE` statement is performed on the `Products` table.

First off, let's examine `InsertProductTrigger`.

### **Creating InsertProductTrigger**

You create a trigger using the `CREATE TRIGGER` statement. [Listing 4.7](#) shows the `InsertProductTrigger.sql` script that creates the trigger `InsertProductTrigger`, which audits the addition of new rows to the `Products` table.

### **Listing 4.7: INSERTPRODUCTTRIGGER.SQL**

---

```
/*
InsertProductTrigger.sql creates a trigger that fires
after an INSERT statement is performed on the
Products table
*/
CREATE TRIGGER InsertProductTrigger
ON Products
AFTER INSERT
AS
-- don't return the number of rows affected
SET NOCOUNT ON

-- declare an int variable to store the new
-- ProductID
DECLARE @NewProductID int

-- get the ProductID of the new row that
-- was added to the Products table
SELECT @NewProductID = ProductID
FROM inserted

-- add a row to the ProductAudit table
INSERT INTO ProductAudit (
    Action
) VALUES (
    'Product added with ProductID of ' +
    CONVERT(nvarchar, @NewProductID)
)
```

---

There are several things you should notice about this `CREATE TRIGGER` statement:

- The `AFTER INSERT` clause specifies that the trigger is to fire after an `INSERT` statement is performed.
- `SET NOCOUNT ON` prevents the trigger from returning the number of rows affected. This improves performance of the trigger.
- You can retrieve column values for the `INSERT` statement that caused the trigger to fire by performing a `SELECT` against the special `inserted` table. For example, you can retrieve all the columns of a newly added row using `SELECT * FROM inserted`. The trigger code retrieves the `ProductID` column of the new row from the `inserted` table.
- The `INSERT` statement that adds a row to the `ProductAudit` table supplies a value only for the `Action` column. This is because the `ID`, `PerformedBy`, and `TookPlace` column values are set automatically by SQL Server.

You can also create, edit, and delete triggers using Enterprise Manager. You do this by first clicking the Tables node in the Databases folder, then clicking the right mouse button on the table you want to modify, and then selecting All Tasks > Manage Triggers. [Figure 4.8](#) shows `InsertProductTrigger` in Enterprise Manager. You'll notice I've added some comments to the start of the code that indicates what the trigger does.



**Figure 4.8:** Using Enterprise Manager to view a trigger

The Object Browser of Query Analyzer also allows you to view, modify, and delete triggers.

**Tip** You can delete a trigger using the `DROP TRIGGER` statement, and you can modify a trigger using the `ALTER TRIGGER` statement.

## Testing *InsertProductTrigger*

To test `InsertProductTrigger`, all you have to do is to add a row to the `Products` table using an `INSERT` statement. For example:

```

INSERT INTO Products (
    ProductName, SupplierID, UnitPrice
) VALUES (
    'Widget', 1, 10
)

```

You can check that `InsertProductTrigger` fired by retrieving the rows from the `ProductAudit` table using the following `SELECT` statement:

```

SELECT *
FROM ProductAudit

```

The row added to the `ProductAudit` table by `InsertProductTrigger` as a result of performing the previous `INSERT` statement is shown in [Table 4.8](#).

**Table 4.8: ROW ADDED TO THE ProductAudit TABLE BY InsertProductTrigger**

ID	ACTION	PERFORMEDBY	TOOKPLACE
1	Product added with ProductID of 80	dbo	2002-07-18 13:55:12.620

## Creating and Testing *UpdateUnitPriceProductTrigger*

The `UpdateUnitPriceProductTrigger` trigger fires after an `UPDATE` statement is performed on the `UnitPrice` column of the `Products` table. If the reduction of the unit price of a product is greater than 25 percent, then a row is added to the `ProductAudit` table to audit the change. [Listing 4.8](#) shows the `UpdateUnitPriceProductTrigger.sql` script.

### **Listing 4.8: UPDATEUNITPRICEPRODUCTTRIGGER.SQL**

---

```
/*
UpdateUnitPriceProductTrigger.sql creates a trigger
that fires after an UPDATE statement is performed on the
the UnitPrice column of the Products table.
If the reduction of the unit price of a product is
greater than 25% then a row is added to the ProductAudit table
to audit the change.
*/
CREATE TRIGGER UpdateUnitPriceProductTrigger
ON Products
AFTER UPDATE
AS
-- don't return the number of rows affected
SET NOCOUNT ON
-- only run the code if the UnitPrice column
-- was modified
IF UPDATE(UnitPrice)
BEGIN
-- declare an int variable to store the
-- ProductID
DECLARE @MyProductID int
-- declare two money variables to store the
-- old unit price and the new unit price
DECLARE @OldUnitPrice money
DECLARE @NewUnitPrice money
-- declare a float variable to store the price
-- reduction percentage
DECLARE @PriceReductionPercentage float
-- get the ProductID of the row that
-- was modified from the inserted table
SELECT @MyProductID = ProductID
FROM inserted
-- get the old unit price from the deleted table
SELECT @OldUnitPrice = UnitPrice
FROM deleted
WHERE ProductID = @MyProductID
```

```
-- get the new unit price from the inserted table
SELECT @NewUnitPrice = UnitPrice
FROM inserted

-- calculate the price reduction percentage
SET @PriceReductionPercentage =
    ((@OldUnitPrice -@NewUnitPrice) / @OldUnitPrice) * 100

-- if the price reduction percentage is greater than 25%
-- then audit the change by adding a row to the PriceAudit table
IF (@PriceReductionPercentage > 25)
BEGIN

    -- add a row to the ProductAudit table
    INSERT INTO ProductAudit (
        Action
    ) VALUES (
        'UnitPrice of ProductID #' +
        CONVERT(nvarchar, @MyProductID) +
        ' was reduced by ' +
        CONVERT(nvarchar, @PriceReductionPercentage) +
        '%'
    )

END
END
```

---

There are a couple of things you should notice about this CREATE TRIGGER statement:

- The AFTER UPDATE clause specifies that the trigger is to fire after an UPDATE statement is performed.
- You can retrieve the old column values before the UPDATE was applied from the deleted table, and you can retrieve the new column values after the UPDATE was applied from the inserted table.

To test `UpdateUnitPriceProductTrigger`, all you have to do is to reduce the value of the `UnitPrice` column for a row in the `Products` table using an UPDATE statement. For example, the following UPDATE statement multiplies the `UnitPrice` by 0.70 for the row with a `ProductID` of 80 (this reduces the `UnitPrice` of that row by 30 percent):

```
UPDATE Products
SET UnitPrice = UnitPrice * 0.70
WHERE ProductID = 80
```

The row added to the `ProductAudit` table as a result of performing this UPDATE statement is shown in [Table 4.9](#). This row is added by `UpdateUnitPriceProductTrigger`.

**Table 4.9: ROW ADDED TO THE ProductAudit TABLE BY UpdateUnitPriceProductTrigger**

ID	ACTION	PERFORMEDBY	TOOKPLACE
2	UnitPrice of ProductID #80 was reduced by 30%	dbo	2002-07-18 17:26:37.590

## Creating and Testing `DeleteProductTrigger`

The `DeleteProductTrigger` trigger fires after a DELETE statement is performed on the `Products` table. This trigger adds a row to the `ProductAudit` table to audit the change. [Listing 4.9](#) shows the `DeleteProductTrigger.sql` script.

**Listing 4.9: DELETEPRODUCTTRIGGER.SQL**

```
/*
DeleteProductTrigger.sql creates a trigger that fires
after a DELETE statement is performed on the
Products table
*/
CREATE TRIGGER DeleteProductTrigger
ON Products
AFTER DELETE
AS

-- don't return the number of rows affected
SET NOCOUNT ON

-- declare an int variable to store the
-- ProductID
DECLARE @NewProductID int

-- get the ProductID of the row that
-- was removed from the Products table
SELECT @NewProductID = ProductID
FROM deleted

-- add a row to the ProductAudit table
INSERT INTO ProductAudit (
    Action
) VALUES (
    'Product #' +
    CONVERT(nvarchar, @NewProductID) +
    ' was removed'
)
```

To test `DeleteProductTrigger`, all you have to do is to remove a row from the `Products` table using a `DELETE` statement. For example, the following `DELETE` statement removes the row with the `ProductID` of 80:

```
DELETE FROM Products
WHERE ProductID = 80
```

The row added to the `ProductAudit` table as a result of performing this `DELETE` statement is shown in [Table 4.10](#). This row is added by `DeleteProductTrigger`.

**Table 4.10: ROW ADDED TO THE ProductAudit TABLE BY DeleteProductTrigger**

ID	ACTION	PERFORMEDBY	TOOKPLACE
3	Product #80 was removed	dbo	2002-07-18 17:35:53.510

## Summary

In this chapter, you learned about programming with Transact-SQL. T-SQL enables you to write programs that contain SQL statements, along with standard programming constructs such as variables, conditional logic, loops, procedures, and functions.

SQL Server provides a number of functions you can use to get values from the database. For example, you can use the COUNT( ) function to get the number of rows in a table. You saw how to use the following functions: aggregate, mathematical, string, date and time, and system.

You can create your own user-defined functions in SQL Server. For example, you might want to create your own function to compute the discounted price given the original price and factor to multiply that price by.

SQL Server allows you to store procedures in a database. Stored procedures differ from user-defined functions in that procedures can return a much wider array of data types. You'll typically create a stored procedure when you need to perform a task that intensively uses the database, or you want to centralize code in the database that any user can call rather than have each user write their own program to perform the same task.

In the [next chapter](#), you'll learn about the ADO.NET classes.

## Chapter 5: Overview of the ADO.NET Classes

### Overview

ADO.NET allows you to interact with a database directly using objects of the *managed provider* classes. These objects allow you to connect to the database and execute SQL statements while directly connected to the database. The example program you saw in [Chapter 1](#) showed how to connect to a database directly and read the rows from a table in a forward-only direction.

ADO.NET also allows you to work in a disconnected manner. When doing this, you store information from a database locally in the memory of the computer on which your program is running. You store that information using objects of the *data set* classes. Once you have that information in the memory, you can then read and manipulate that information. For example, you can display the columns for the rows, add new rows, modify rows, and delete rows. Periodically, you'll reconnect to the database to synchronize your changes you've made locally with the database. This disconnected model allows you to write applications that run on the Internet, as well as for devices that aren't always connected to the database-PDAs such as the Palm and the Pocket PC, for example.

This chapter provides descriptions of the ADO.NET classes, as well as a complete C# program that connects to a database, stores the rows locally, disconnects from the database, and then reads the contents of those local rows while disconnected from the database. This capability to store a local copy of rows retrieved from the database is one of the main strengths of ADO.NET. The example program illustrates the basic ideas of using the ADO.NET disconnected model to read rows from the database and store them locally in memory. In later chapters, you'll see how to modify data locally and then synchronize those changes with the database.

This chapter lays the foundation for [Part II](#), "Fundamental Database Programming with ADO.NET," where you'll see the details of the various ADO.NET classes.

Featured in this chapter:

- The Managed Provider and Generic Data Set Classes
- Performing a SQL SELECT Statement and Storing the Rows Locally

## The Managed Provider and Generic Data Set Classes

To provide both connected and disconnected database access, ADO.NET defines two sets of classes: managed provider and generic data.

You use objects of the managed provider classes to directly connect to a database and to synchronize your locally stored data with the database. You can use the managed provider classes to read rows from the database in a forward-only direction. You use a different set of managed provider classes depending on the database you use.

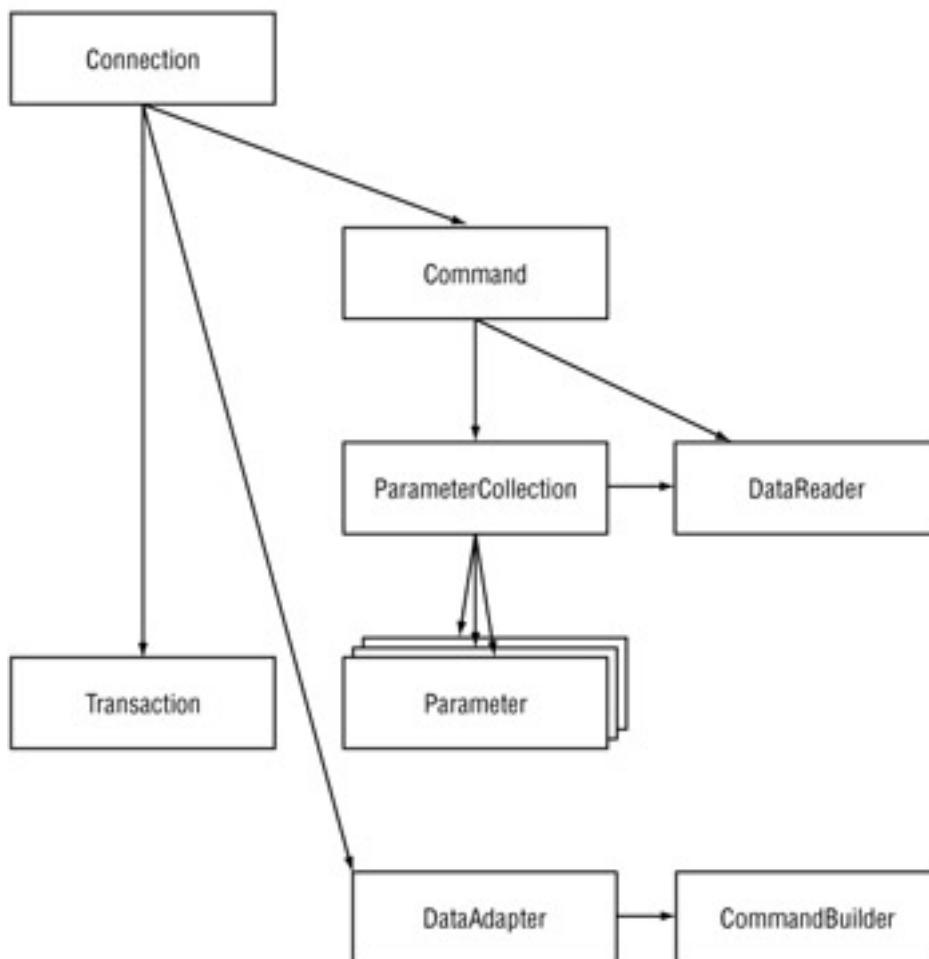
You use objects of the generic data classes to store a local copy of the information retrieved from the database. This copy is stored in the memory of the computer where the C# program is running. The main generic data class is the `System.Data.DataSet` class. The generic data classes, as their name suggests, are not specific to any database, and you always use the same classes regardless of the database you use. The generic data classes represent information retrieved from the database as XML.

### The Managed Provider Classes

The managed provider objects allow you to directly access a database, and you'll be introduced to the classes that allow you to create these objects in this section. You use the managed provider objects to connect to the database and read and write information to and from the database.

[Figure 5.1](#) illustrates some of the managed provider objects and how they relate to each other.

Managed Provider Objects



**Figure 5.1:** Some of the managed provider objects

There are currently three sets of managed provider classes, and each set is designed to work with different database standards:

- **SQL Server Managed Provider Classes** You use the SQL Server managed provider classes to connect to a SQL Server database.

- **OLE DB Managed Provider Classes** You use the OLE DB (Object Linking and Embedding for Databases) managed provider classes to connect to any database that supports OLE DB, such as Access or Oracle.
- **ODBC Managed Provider Classes** You use the ODBC (Open Database Connectivity) managed provider classes to connect to any database that supports ODBC. All the major databases support ODBC, but ODBC is typically slower than the previous two sets of classes when working with .NET. You should use the ODBC managed provider classes only when there aren't any alternative OLE DB managed provider classes.

These three sets of classes all implement the same basic functionality.

**Note** Whenever you see *Sq1* at the start of a managed provider class name, you know that class is used with a SQL Server database. For example, *SqlConnection* allows you to connect to a SQL Server database. Similarly, *OleDb* is for databases that support OLE DB. For example, *OleDbConnection* allows you to connect to a database using OLE DB. Finally, *Odbc* is for databases that support ODBC. For example, *OdbcConnection* allows you to connect to a database using ODBC. I refer to all of these classes as the *Connection* classes.

You'll see some of the various managed provider classes in the following sections.

## The *Connection* Classes

There are three Connection classes: *SqlConnection*, *OleDbConnection*, and *OdbcConnection*. You use an object of the *SqlConnection* class to connect to a SQL Server database. You use an object of the *OleDbConnection* class to connect to any database that supports OLE DB, such as Access or Oracle. You use an object of the *OdbcConnection* class to connect to any database that supports ODBC. Ultimately, all communication with a database is done through a *Connection* object.

## The *Command* Classes

There are three Command classes: *SqlCommand*, *OleDbCommand*, and *OdbcCommand*. You use a *Command* object to run a SQL statement, such as a SELECT, INSERT, UPDATE, or DELETE statement. You can also use a *Command* object to call a stored procedure or retrieve rows from a specific table. You run the command stored in a *Command* object using a *Connection* object.

## The *Parameter* Classes

There are three Parameter classes: *SqlParameter*, *OleDbParameter*, and *OdbcParameter*. You use a *Parameter* object to pass a parameter to a *Command* object. You can use a *Parameter* to pass a value to a SQL statement or a stored procedure call. You can store multiple *Parameter* objects in a *Command* object through a *ParameterCollection* object.

## The *ParameterCollection* Classes

There are three *ParameterCollection* classes: *SqlParameterCollection*, *OleDbParameterCollection*, and *OdbcParameterCollection*. You use a *ParameterCollection* object to store multiple *Parameter* objects for a *Command* object.

## The *DataReader* Classes

There are three *DataReader* classes: *SqlDataReader*, *OleDbDataReader*, and *OdbcDataReader*. You use a *DataReader* object to read rows retrieved from the database using a *Command* object.

*DataReader* objects can only be used to read rows in a forward direction. *DataReader* objects act as an alternative to a *DataSet* object. You cannot use a *DataReader* to modify rows in the database.

**Tip** Reading rows using a *DataReader* object is typically faster than reading from a *DataSet*.

## The *DataAdapter* Classes

There are three *DataAdapter* classes: *SqlDataAdapter*, *OleDbDataAdapter*, and *OdbcDataAdapter*. You use a

`DataAdapter` object to move rows between a `DataSet` object and a database. You use a `DataAdapter` object to synchronize your locally stored rows with the database. This synchronization is performed through a `Connection` object. For example, you can read rows from the database into a `DataSet` through a `DataAdapter`, modify those rows in your `DataSet`, and then push those changes to the database through a `Connection` object.

## The `CommandBuilder` Classes

There are three `CommandBuilder` classes: `SqlCommandBuilder`, `OleDbCommandBuilder`, and `OdbcCommandBuilder`. You use a `CommandBuilder` object to automatically generate single-table `INSERT`, `UPDATE`, and `DELETE` commands that synchronize any changes you make to a `DataSet` object with the database. This synchronization is performed through a `DataAdapter` object.

## The `Transaction` Classes

There are three `Transaction` classes: `SqlTransaction`, `OleDbTransaction`, and `OdbcTransaction`. You use a `Transaction` object to represent a *database transaction*. A database transaction is a group of statements that modify the rows in the database. These statements are considered a logical unit of work. For example, in the case of a banking transaction, you might want to withdraw money from one account and deposit it into another. You would then commit both of these changes as one unit, or if there's a problem, roll back both changes.

## Namespaces for the Managed Provider Classes

The managed provider classes for SQL Server (`SqlConnection` and so on) are declared in the `System.Data.SqlClient` namespace. The classes for OLE DB-compliant databases (`SqlDbConnection` and so on) are declared in the `System.Data.OleDb` namespace. The classes for ODBC-compliant databases (`odbcConnection` and so on) are declared in the `System.Data.Odbc` namespace.

**Note** At time of writing, you have to download the ODBC managed provider classes from Microsoft's Web site at <http://msdn.microsoft.com/downloads>. This download is separate from the .NET SDK. Look for "ODBC .NET Data Provider" in the MSDN table of contents.

In the following section, you'll learn about the generic data classes.

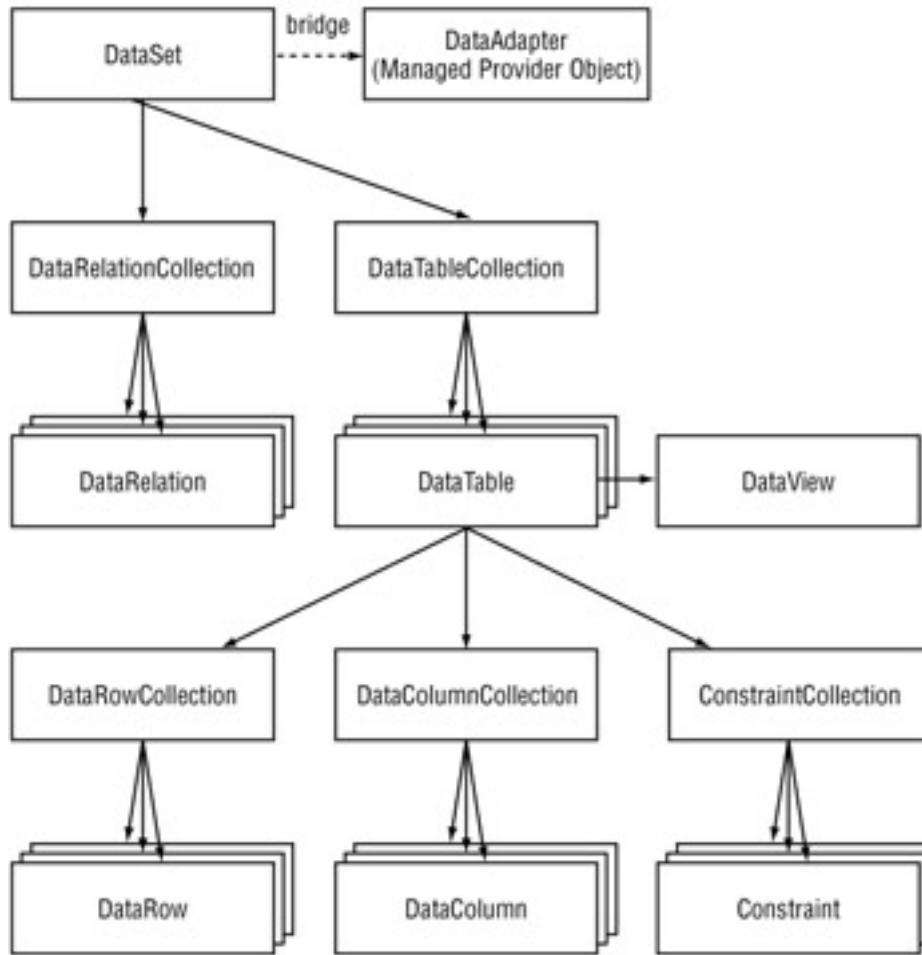
## The Generic Data Classes

As you learned in the [previous section](#), you can use the managed data provider objects to connect to the database through a `Connection` object, issue a SQL statement through a `Command` object, and read retrieved rows using a `DataReader` object; however, you can read rows only in a forward only direction and you must be connected to the database.

The generic data objects allow you to store a local copy of the information stored in the database. This allows you to work the information while disconnected from the database. You can read the rows in any order, and you can search, sort, and filter those rows in a flexible manner. You can even make changes to those rows and then synchronize those changes with the database.

[Figure 5.2](#) illustrates some of the generic data set objects and how they relate to each other. The bridge between the managed provider and generic data set objects is the `DataAdapter`, which you use to synchronize changes between your `DataSet` and the database.

### Generic Data Set Objects



**Figure 5.2:** Some of the generic data set objects

The following sections outline some of the generic data classes.

### The *DataSet* Class

You use an object of the `DataSet` class to represent a local copy of the information stored in the database. You can make changes to that local copy in your `DataSet` and then later synchronize those changes with the database through a managed provider `DataAdapter` object. A `DataSet` object can represent database structures such as tables, rows, and columns. You can even add constraints to your locally stored tables to enforce unique and foreign key constraints.

You can also use a `DataSet` object to represent XML data. In fact, all information stored in a `DataSet` is represented using XML, including information retrieved from the database.

### The *DataTable* Class

You use an object of the `DataTable` class to represent a table. You can store multiple `DataTable` objects in a `DataSet` through a `DataTableCollection` object. A `DataSet` object has a property named `Tables`, which you use to access the `DataTableCollection` containing the `DataTable` objects stored in that `DataSet`.

### The *DataRow* Class

You use an object of the `DataRow` class to represent a row. You can store multiple `DataRow` objects in a `DataTable` through a `DataRowCollection` object. A `DataTable` object has a property named `Rows`, which you use to access the `DataRowCollection` containing the `DataRow` objects stored in that `DataTable`.

### The *DataColumn* Class

You use an object of the `DataTable` class to represent a column. You can store multiple `DataTable` objects in a `DataTable` through a `DataTableCollection` object. A `DataTable` object has a property named `Columns`, which you use to access the `DataTableCollection` containing the `DataTable` objects stored in that `DataTable`.

## The `Constraint` Class

You use an object of the `Constraint` class to represent a database constraint that is to be enforced on one or more `DataTable` objects of a `DataTable`. You can store multiple `Constraint` objects in a `DataTable` through a `ConstraintCollection` object. A `DataTable` object has a property named `Constraints`, which you use to access the `ConstraintCollection` containing the `Constraint` objects stored in that `DataTable`.

## The `DataView` Class

You use an object of the `DataView` class to view only specific rows in a `DataTable` object using a filter, which specifies the criteria to restrict the rows.

## The `DataRelation` Class

You use an object of the `DataRelation` class to represent a relationship between two `DataTable` objects. You can use a `DataRelation` object to model parent-child relationships between two database tables. You can store multiple `DataRelation` objects in a `DataSet` through a `DataRelationCollection` object. A `DataSet` object has a property named `Relations`, which you use to access the `DataRelationCollection` containing the `DataRelation` objects stored in that `DataSet`.

## The `UniqueConstraint` Class

You use an object of the `UniqueConstraint` class to represent a database constraint that enforces that the value stored in a `DataTable` object is unique. The `UniqueConstraint` class is derived from the `Constraint` class. You can store multiple `UniqueConstraint` objects in a `DataTable` through a `ConstraintCollection` object.

## The `ForeignKeyConstraint` Class

You use an object of the `ForeignKeyConstraint` class to specify the action performed when the column values in the parent table are updated or deleted.

The `ForeignKeyConstraint` class is derived from the `Constraint` class. You can either have the child rows deleted (cascading action), set the child columns to null, or set the child columns to a default value. You can store multiple `ForeignKeyConstraint` objects in a `DataTable` through a `ConstraintCollection` object.

## Namespaces for the Generic Data Classes

The `DataSet`, `DataTable`, `DataRow`, `DataTable`, `Constraint`, and `DataView` classes are all declared in the `System.Data` namespace. This namespace contains other classes that you can use in your programs. You can view the full set of classes declared in the `System.Data` namespace using the .NET documentation. [Chapter 1](#) explains how you access this documentation.

In the [next section](#), you'll see a simple example that illustrates how to issue a SQL `SELECT` statement that retrieves rows from the `Customers` table, and then stores the returned rows in a `DataSet` object. This program will give you a basic understanding on how to use some of the managed provider and generic data classes previously outlined. In [Part II](#), you'll see the details of the various classes used in this example.

## Performing a SQL **SELECT** Statement and Storing the Rows Locally

In the example featured in this section, you'll see how to connect to the SQL Server Northwind database and perform a SQL **SELECT** statement to retrieve the `CustomerID`, `CompanyName`, `ContactName`, and `Address` columns for the first 10 rows from the `Customers` table. These rows are stored in a `DataSet` object.

**Note** Since I'll be using a SQL Server database, I'll use the SQL Server managed provider classes in the example.

### Outlining the Procedure

You can use the following steps to retrieve the rows into a `DataSet` object:

1. Formulate a string containing the details of the database connection.
2. Create a `SqlConnection` object to connect to the database, passing the connection string to the constructor.
3. Formulate a string containing a `SELECT` statement to retrieve the columns for the rows from the `Customers` table.
4. Create a `SqlCommand` object to hold the `SELECT` statement.
5. Set the `CommandText` property of the `SqlCommand` object to the `SELECT` string.
6. Create a `SqlDataAdapter` object.
7. Set the `SelectCommand` property of the `SqlAdapter` object to the `SqlCommand` object.
8. Create a `DataSet` object to store the results of the `SELECT` statement.
9. Open the database connection using the `Open()` method of the `SqlConnection` object.
10. Call the `Fill()` method of the `SqlDataAdapter` object to retrieve the rows from the table, storing the rows locally in a `DataTable` of the `DataSet` object.
11. Close the database connection, using the `Close()` method of the `SqlConnection` object created in step 1.
12. Get the `DataTable` object from the `DataSet` object.
13. Display the columns for each row in the `DataTable`, using a `DataRow` object to access each row in the `DataTable`.

In the following sections, you'll learn the details of these steps and see example code.

### Step 1: Formulate a String Containing the Details of the Database Connection

When connecting to a SQL Server database, your string must specify the following:

- The name of the computer on which SQL Server is running. You set this in the `server` part of the string. If SQL Server is running on your local computer, you can use `localhost` as the server name. For example: `server=localhost`.
- The name of the database. You set this in the `database` part of the string. For example: `database=Northwind`.
- The name of the user to connect to the database as. You set this in the `uid` part of the string. For example: `uid=sa`.
- The password for the database user. You set this in the `pwd` part of the string. For example: `pwd=sa`.

**Note** Typically, your organization's DBA (database administrator) will provide you with the appropriate values for the connection string. The DBA is responsible for administering the database.

The following example creates a string named `connectionString` and sets it to an appropriate string to connect to the Northwind database running on the local computer, using the `sa` user (with a password of `sa`) to connect to that database:

```
string connectionString =
    "server=localhost;database=Northwind;uid=sa;pwd=sa";
```

Your connection string will differ based on how you connect to your Northwind database.

## Step 2: Create a `SqlConnection` Object to Connect to the Database

Create a `SqlConnection` object to connect to the database, passing the connection string created in the previous step to the constructor. You use an object of the `SqlConnection` class to connect to a SQL Server database.

The following example creates a `SqlConnection` object named `mySqlConnection`, passing `connectionString` (created in step 1) to the constructor:

```
SqlConnection mySqlConnection =
    new SqlConnection(connectionString);
```

## Step 3: Formulate a String Containing the `SELECT` Statement

Formulate a string containing the `SELECT` statement to retrieve the `CustomerID`, `CompanyName`, `ContactName`, and `Address` columns for the first 10 rows from the `Customers` table. For example:

```
string selectString =
    "SELECT TOP 10 CustomerID, CompanyName, ContactName, Address " +
    "FROM Customers " +
    "ORDER BY CustomerID";
```

**Note** You use the `TOP` keyword in combination with an `ORDER BY` clause to retrieve the top N rows from a `SELECT` statement. You can learn more about the `TOP` keyword in [Chapter 3](#).

## Step 4: Create a `SqlCommand` Object to Hold the `SELECT` Statement

You can call the `CreateCommand()` method of `mySqlConnection` to create a new `SqlCommand` object for that connection. The `CreateCommand()` method returns a new `SqlCommand` object for the `SqlConnection` object.

In the following example, a new `SqlCommand` object named `mySqlCommand` is set to the `SqlCommand` object returned by calling the `CreateCommand()` method of `mySqlConnection`:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
```

## Step 5: Set the `CommandText` Property of the `SqlCommand` Object to the `SELECT` String

Set the `CommandText` property of your `SqlCommand` object to the `SELECT` string created in step 4. The `CommandText` property contains the SQL statement you want to perform. In the following example, the `CommandText` property of `mySqlCommand` is set to `selectString`:

```
mySqlCommand.CommandText = selectString;
```

## Step 6: Create a `SqlDataAdapter` Object

You use a `SqlDataAdapter` object to move information between your `DataSet` object and the database. You'll see how to create a `DataSet` object in step 8. The following example creates a `SqlDataAdapter` object named `mySqlDataAdapter`:

```
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
```

## Step 7: Set the `SelectCommand` Property of the `SqlAdapter` Object to the `SqlCommand` Object

The `SelectCommand` property contains the `SELECT` statement you want to run. In the following example, the `SelectCommand` property of `mySqlDataAdapter` is set to `mySqlCommand`:

```
mySqlDataAdapter.SelectCommand = mySqlCommand;
```

This enables you to perform the `SELECT` statement defined in `mySqlCommand`. Step 10 actually performs the `SELECT` statement to retrieve rows from the database.

## Step 8: Create a *DataSet* Object to Store the Results of the *SELECT* Statement

You use a `DataSet` object to store a local copy of information retrieved from the database. The following example creates a `DataSet` object named `myDataSet`:

```
DataSet myDataSet = new DataSet();
```

## Step 9: Open the Database Connection Using the *Open()* Method of the *SqlConnection* Object

The following example calls the `Open()` method for `mySqlConnection`:

```
mySqlConnection.Open();
```

Once you've opened the database connection, you can access the database.

## Step 10: Call the *Fill()* Method of the *SqlDataAdapter* Object to Retrieve the Rows From the Table

Call the `Fill()` method of your `SqlDataAdapter` object to retrieve the rows from the database, storing these rows locally in a `DataTable` of your `DataSet` object.

The `Fill()` method is overloaded, and the version you'll see in the example accepts two parameters:

- A `DataSet` object
- A string containing the name of the `DataTable` object to create in the specified `DataSet`

The `Fill()` method then creates a `DataTable` in the `DataSet` with the specified name and runs the `SELECT` statement. The `DataTable` created in your `DataSet` is then populated with the rows retrieved by the `SELECT` statement.

The following example calls the `Fill()` method of `mySqlDataAdapter`, passing `myDataSet` and "Customers" to the `Fill()` method:

```
mySqlDataAdapter.Fill(myDataSet, "Customers");
```

The `Fill()` method creates a `DataTable` object named `Customers` in `myDataSet` and populates it with the rows retrieved by the `SELECT` statement. You can access these rows, even when disconnected from the database.

## Step 11: Close the Database Connection

Close the database connection using the `Close()` method of the `SqlConnection` object created in the first step. For example:

```
mySqlConnection.Close();
```

**Note** Of course, you don't have to immediately close the database connection before reading locally stored rows from your `DataSet`. I close the connection at this point in the example to show that you can indeed read the locally stored rows- even when disconnected from the database.

## Step 12: Get the *DataTable* Object From the *DataSet* Object

Get the `DataTable` object created in step 10 from the `DataSet` object.

You get a `DataTable` from your `DataSet` using the `Tables` property, which returns a `DataTableCollection` object. To get an individual `DataTable` from your `DataSet`, you pass the name of your `DataTable` in brackets ("Customers", for example) to the `Tables` property. The `Tables` property will then return your requested `DataTable`, which you can store in a new `DataTable` object that you declare. In the following example, `myDataSet.Tables["Customers"]` returns the `Customers` `DataTable` created in `myDataSet` in step 10, and stores the returned `DataTable` in `myDataTable`:

```
DataTable myDataTable = myDataSet.Tables["Customers"];
```

**Note** You can also specify the `DataTable` you want to get by passing a numeric value to the `Tables` property. For example, `myDataSet.Tables[0]` also returns the `Customers` `DataTable`.

## Step 13: Display the Columns for Each Row in the `DataTable`

Display the columns for each row in the `DataTable`, using a `DataRow` object to access each row in the `DataTable`. The `DataTable` class defines a property named `Rows` that returns a `DataRowCollection` object containing the `DataRow` objects stored in that `DataTable`. You can use the `Rows` property in a `foreach` loop to iterate over the `DataRow` objects. For example:

```
foreach (DataRow myDataRow in myDataTable.Rows)
{
    // ... access the myDataRow object
}
```

Each `DataRow` object stores  `DataColumn` objects that contain the values retrieved from the columns of the database table. You can access these column values by passing the name of the column in brackets to the `DataRow` object. For example, `myDataRow["CustomerID"]` returns the value of the `CustomerID` column.

In the following example, a `foreach` loop iterates over the `DataRow` objects in `myDataTable`, and the column values are displayed for each row:

```
foreach (DataRow myDataRow in myDataTable.Rows)
{
    Console.WriteLine("CustomerID = " + myDataRow["CustomerID"]);
    Console.WriteLine("CompanyName = " + myDataRow["CompanyName"]);
    Console.WriteLine("ContactName = " + myDataRow["ContactName"]);
    Console.WriteLine("Address = " + myDataRow["Address"]);
}
```

As you can see, the name of each column is passed in brackets to each `DataRow` object, which then returns the column value.

**Note** You can also specify the column you want to get by passing a numeric value in brackets. For example, `myDataRow[0]` also returns the `CustomerID` column value.

## Putting It All Together

[Listing 5.1](#) shows a complete program that uses these steps. This program is named `SelectIntoDataSet.cs` and is located in the `ch05` directory.

### **Listing 5.1: SELECTINTODATASET.CS**

---

```
/*
SelectIntoDataSet.cs illustrates how to perform a SELECT statement
and store the returned rows in a DataSet object
*/
using System;
using System.Data;
using System.Data.SqlClient;
```

```
class SelectIntoDataSet
{
    public static void Main()
    {
        // step 1: formulate a string containing the details of the
        // database connection
        string connectionString =
            "server=localhost;database=Northwind;uid=sa;pwd=sa";

        // step 2: create a SqlConnection object to connect to the
        // database, passing the connection string to the constructor
        SqlConnection mySqlConnection =
            new SqlConnection(connectionString);

        // step 3: formulate a SELECT statement to retrieve the
        // CustomerID, CompanyName, ContactName, and Address
        // columns for the first ten rows from the Customers table
        string selectString =
            "SELECT TOP 10 CustomerID, CompanyName, ContactName, Address " +
            "FROM Customers " +
            "ORDER BY CustomerID";

        // step 4: create a SqlCommand object to hold the SELECT statement
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        // step 5: set the CommandText property of the SqlCommand object to
        // the SELECT string
        mySqlCommand.CommandText = selectString;

        // step 6: create a SqlDataAdapter object
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();

        // step 7: set the SelectCommand property of the SqlDataAdapter object
        // to the SqlCommand object
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        // step 8: create a DataSet object to store the results of
        // the SELECT statement
        DataSet myDataSet = new DataSet();

        // step 9: open the database connection using the
        // Open() method of the SqlConnection object
        mySqlConnection.Open();

        // step 10: use the Fill() method of the SqlDataAdapter object to
        // retrieve the rows from the table, storing the rows locally
        // in a DataTable of the DataSet object
        Console.WriteLine("Retrieving rows from the Customers table");
        mySqlDataAdapter.Fill(myDataSet, "Customers");

        // step 11: close the database connection using the Close() method
        // of the SqlConnection object created in Step 1
        mySqlConnection.Close();

        // step 12: get the DataTable object from the DataSet object
        DataTable myDataTable = myDataSet.Tables["Customers"];

        // step 13: display the columns for each row in the DataTable,
        // using a DataRow object to access each row in the DataTable
        foreach (DataRow myDataRow in myDataTable.Rows)
        {
```

```

        Console.WriteLine("CustomerID = " + myDataRow["CustomerID"]);
        Console.WriteLine("CompanyName = " + myDataRow["CompanyName"]);
        Console.WriteLine("ContactName = " + myDataRow["ContactName"]);
        Console.WriteLine("Address = " + myDataRow["Address"]);
    }
}
}

```

---

The output from this program is as follows:

```

Retrieving rows from the Customers table
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
ContactName = Maria Anders
Address = Obere Str. 57
CustomerID = ANATR
CompanyName = Ana Trujillo Emparedados y helados
ContactName = Ana Trujillo
Address = Avda. de la Constitución 2222
CustomerID = ANTON
CompanyName = Antonio Moreno Taquería
ContactName = Antonio Moreno
Address = Mataderos 2312
CustomerID = AROUT
CompanyName = Around the Horn
ContactName = Thomas Hardy
Address = 120 Hanover Sq.
CustomerID = BERGS
CompanyName = Berglunds snabbköp
ContactName = Christina Berglund
Address = Berguvsvägen 8
CustomerID = BLAUS
CompanyName = Blauer See Delikatessen
ContactName = Hanna Moos
Address = Forsterstr. 57
CustomerID = BLONP
CompanyName = Blondesdsl père et fils
ContactName = Frédérique Citeaux
Address = 24, place Kléber
CustomerID = BOLID
CompanyName = Bólido Comidas preparadas
ContactName = Martín Sommer
Address = C/ Araquil, 67
CustomerID = BONAP
CompanyName = Bon app'
ContactName = Laurence Lebihan
Address = 12, rue des Bouchers
CustomerID = BOTTM
CompanyName = Bottom-Dollar Markets
ContactName = Elizabeth Lincoln
Address = 23 Tsawassen Blvd.

```

## Summary

In this chapter, you got an overview of the ADO.NET classes, and you examined a complete program that connected to a database, stored the rows locally, disconnected from the database, and then read the contents of those local rows while disconnected from the database.

ADO.NET allows you to interact with a database directly using objects of the *managed provider* classes. These objects allow you to connect to the database and execute SQL statements while directly connected to the database. You use different sets of managed provider classes, depending on the database you are using.

ADO.NET also allows you to work in a disconnected manner. When doing this, you store information from a database locally in the memory of the computer on which your program is running. You store that information using objects of the *data set* classes.

Some of the SQL Server managed provider classes include `SqlConnection`, `SqlCommand`, `SqlDataReader`, `SqlDataAdapter`, and `SqlTransaction`. You use an object of the `SqlConnection` class to connect to a SQL Server database. You use an object of the `SqlCommand` class to represent a SQL statement or stored procedure call that you then execute. You use an object of the `SqlDataReader` class to read rows retrieved from a SQL Server database. You use an object of the `SqlDataAdapter` class to move rows between a `DataSet` object and a SQL Server database. You use an object of the `SqlTransaction` class to represent a database transaction in a SQL Server database.

You use an object of the `DataSet` class to represent a local copy of the information stored in a database. You can also use a `DataSet` object to represent XML data. Some of the objects you can store in a `DataSet` include `DataTable`, `DataRow`,  `DataColumn`, `DataRelation`, and `DataView` objects.

You use an object of the `DataTable` class to represent a table. You use an object of the `DataRow` class to represent a row. You use an object of the  `DataColumn` class to represent a column. You use an object of the `DataRelation` class to represent a relationship between two `DataTable` objects. You use a `DataRelation` object to model parent-child relationships between two database tables. You use an object of the `DataView` class to view only specific rows in a `DataTable` object using a filter.

In [Chapter 6](#), you'll learn how to use Visual Studio .NET to create Windows programs.

## Chapter 6: Introducing Windows Applications and ADO.NET

### Overview

In the previous chapters, you ran programs using the Windows Command Prompt tool. In this chapter, you'll be introduced to Windows applications. Windows provides graphical items, such as menus, text boxes, and radio buttons, so that you can build a visual interface that is easy to use. You can create Windows applications that use ADO.NET, and you'll see how to do that, using Visual Studio .NET (VS .NET), in this chapter.

Windows applications are simple to learn and use because people have become accustomed to interacting with machines in a visual manner. The ubiquitous Microsoft Word and Excel are just two examples of how successful Windows applications can be because of the way they combine power and ease of use.

The idea of using graphical user interfaces (GUIs) and a mouse to interact with a computer is not unique to Windows. In fact, these concepts were originally developed back in the early 1970s by engineers at Xerox Corporation's Palo Alto Research Center (PARC) in California, and one of the first computers to use a GUI and a mouse was the Alto. Unfortunately, the Alto was expensive, and it wasn't until Apple Computer launched the Macintosh in 1984 that the GUI became popular. Later, Microsoft developed the Windows operating system that built on the ideas made popular by Apple.

Featured in this chapter:

- Developing a simple Windows application
- Using Windows controls
- Accessing a database with a DataGrid control
- Creating a Windows form with the Data Form Wizard

## Developing a Simple Windows Application

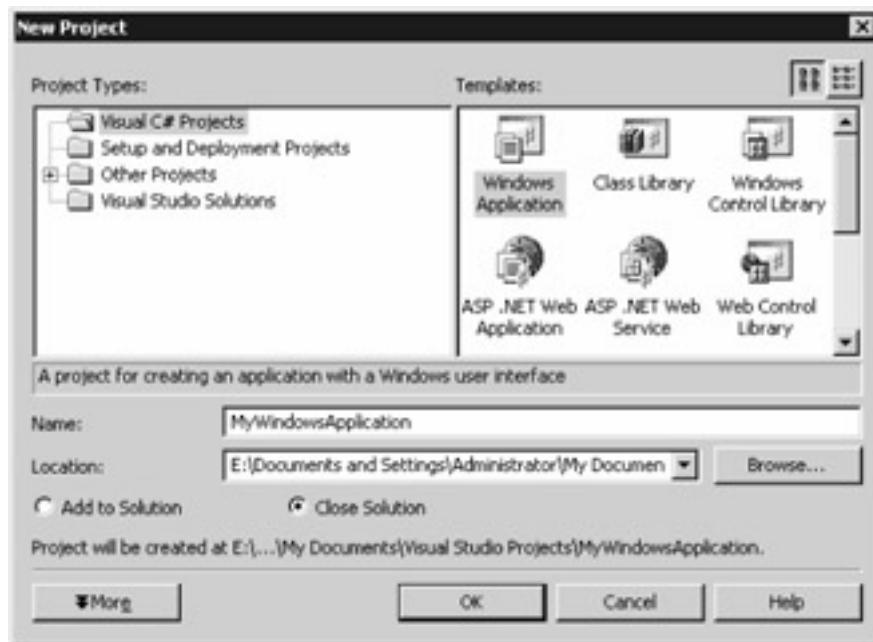
In this section, you'll see how to create a simple Windows application using VS .NET. This application will consist of a single form that contains a label and a button. When you click the button, the text for the label will change to a quote from Shakespeare's play, *Macbeth*. You'll also see how to compile and run the example application.

### Creating the Windows Application

Start VS .NET by selecting Start > Programs > Microsoft Visual Studio .NET > Microsoft Visual Studio .NET. To create a new Windows application, click the New Project button on the Start page, or select File > New > Project.

**Tip** You can also create a new project by pressing Ctrl+Shift+N on your keyboard.

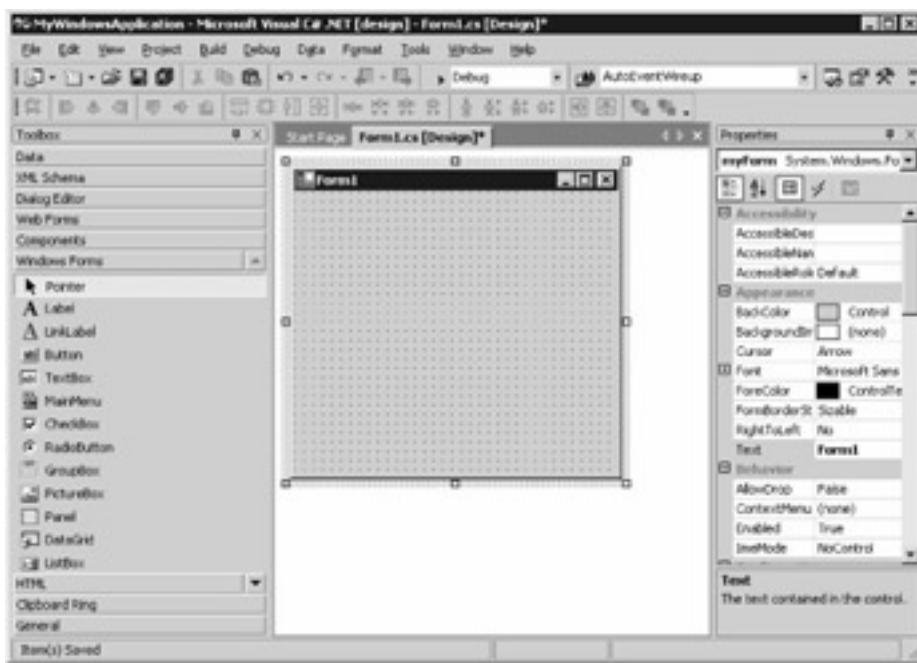
You'll see the New Project dialog box, which you use to select the type of project you want to create. Because you're going to create a C# Windows application, select the Visual C# Projects folder from the Project Types list, and select Windows Application from the Templates area of the New Project dialog box. VS .NET will assign a default name to your project; this default name will be WindowsApplication1, or something similar. You can specify your own name for your project by changing the text in the Name field; go ahead and enter **MyWindowsApplication** in the Name field, as shown in [Figure 6.1](#).



**Figure 6.1:** Creating a C# Windows application in Visual Studio .NET

**Note** The Location field specifies the directory where the files for your new project are stored. VS .NET will set a default directory, but you can change this by entering your own directory. This default directory is the *Documents and Settings* directory on your hard drive.

Click the OK button to continue. VS .NET will create a new subdirectory named **MyWindowsApplication** in the directory specified in the Location field. Once VS .NET creates the directory, along with some initial files for your project, VS .NET will display a blank form, as shown in [Figure 6.2](#). You can think of the form as the canvas on which you can place standard Windows controls, such as labels, text boxes, and buttons. You'll be adding controls to your form shortly.



**Figure 6.2:** A blank form

In the [next section](#), you'll learn about the Toolbox, which you use to add controls to your form.

## Working with the Toolbox

You add controls to your form by selecting the control from the Toolbox and dragging the control to your form. You can also click and drag, or double-click on the control to drop a new one of that type onto your form. As you can see in [Figure 6.2](#) shown earlier, the Toolbox is to the left of the blank form.

**Note** If you don't see the Toolbox, you can display it by selecting View > Toolbox, or by pressing Ctrl+Alt+X on your keyboard.

You can see that the available items in the Toolbox are categorized into groups with names such as Data and XML Schema. The Toolbox will show only categories that are relevant to the type of application you are developing. The following list describes the contents of some of these categories:

- **Data** The Data category contains classes that allow you to access and store information from a database. The Data category includes the following classes: `SqlConnection`, `SqlCommand`, `DataSet`, and `DataTable`, among others.
- **XML Schema** The XML Schema category contains classes that allow you to access XML data.
- **Dialog Editor** The Dialog Editor category contains controls that you can place on Windows dialog boxes.
- **Web Forms** The Web Forms category contains controls that are for web forms. You can design web forms using VS .NET and deploy them to Microsoft's Internet Information Server (IIS). These web forms may then be run over the Internet.
- **Components** The Components category contains classes such as `FileSystemWatcher`, which allows you to monitor changes in a computer's file system. Other classes include `EventLog`, `DirectoryEntry`, `DirectorySearcher`, `MessageQueue`, `PerformanceCounter`, `Process`, `ServiceController`, and `Timer`. These allow you to perform various system operations.
- **Windows Forms** The Windows Forms category contains controls that you can add to a Windows form. These include labels, buttons, and text boxes, among others. You'll use some of these controls in this chapter.
- **HTML** The HTML category contains controls that you can add to a web form. These include labels, buttons, tables, and images, among others.

In the [next section](#), you'll learn about the Properties window.

## Working with the Properties Window

The Properties window contains aspects of a control that you can set. For example, you can set the background color of your form using the `BackColor` property. Some other properties of the form control include `ForeColor` (the foreground color) and `BackgroundImage` (an image displayed in the background). Different types of controls have different types of properties.

As you can see from [Figure 6.2](#) shown earlier, the Properties window is to the right of the blank form.

**Note** If you don't see the Properties window, you can display it by selecting View > Properties Window, or by pressing F4 on your keyboard.

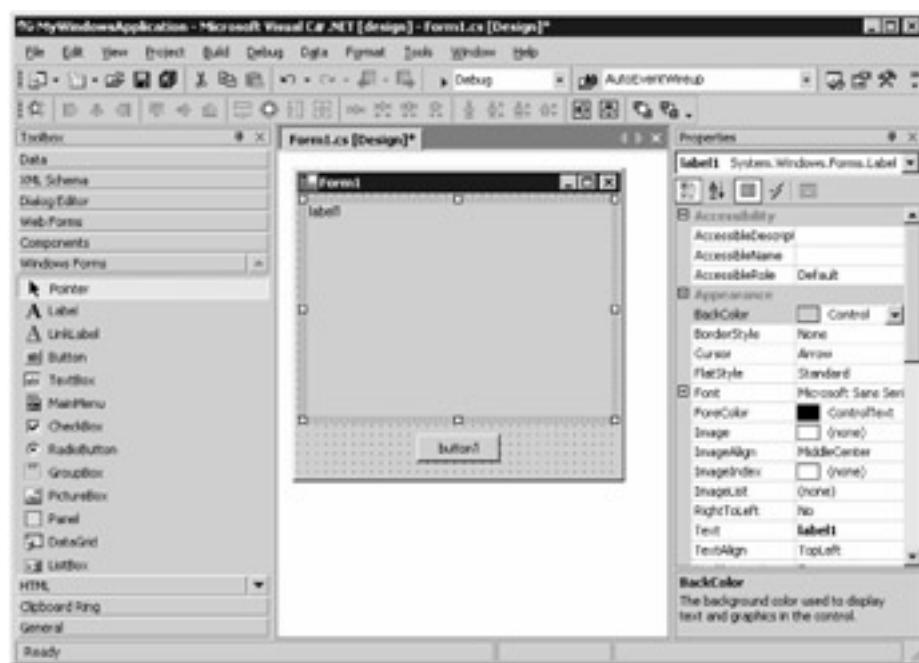
You set the property by clicking the area to the right of the property name. Go ahead and click to the right of the `BackColor` property to view some of the colors to which you can set this property.

In the [next section](#), you'll learn how to add a label and button control to your form. You'll also set a couple of the properties for those controls.

## Adding a Label and a Button Control

To add a label and a button control to your form select the appropriate control from the Toolbox and drag it to your form. For example, to add a label to your form, you select the label control from the Toolbox. Once you've dragged a label to your form, you can resize it by using the mouse or setting the `Size` property in the Properties window. You can also click on the label in the Toolbox and drag it on your form.

Make your label big enough so that it stretches across the length of your form. Next, add a button control below your label, as shown in [Figure 6.3](#).



**Figure 6.3:** The form with a label and button control

Next, you'll change some of the properties for your label and button. You do this using the Properties window. Set the `Name` property of your label to `myLabel`. Set the `Name` and `Text` properties for your button to `myButton` and `Press Me!`, respectively. Also, set the `Text` property of your form to `My Form`.

**Note** You use the `Name` property when referencing a Windows control in C# code.

Next, you'll add a line of code to the `myButton_Click()` method. This method is executed when `myButton` is clicked in your running form. The statement you'll add to `myButton_Click()` will set the `Text` property of `myLabel` to a string. This string will contain a line from Shakespeare's play, *Macbeth*. To add the code, double-click `myButton` and enter the following code in the `myButton_Click()` method:

```
myLabel.Text =
    "Is this a dagger which I see before me,\n" +
```

```
"The handle toward my hand? Come, let me clutch thee.\n" +
"I have thee not, and yet I see thee still.\n" +
"Art thou not, fatal vision, sensible\n" +
"To feeling as to sight? or art thou but\n" +
"A dagger of the mind, a false creation,\n" +
"Proceeding from the heat-oppressed brain?";
```

**Note** If you're a Shakespeare fan, you'll recognize this line from the scene before Macbeth kills King Duncan.

You've now finished your form. Build your project by selecting Build > Build Solution, or by pressing Ctrl+Shift+B on your keyboard.

To run your form, select Debug > Start without Debugging, or press Ctrl+F5 on your keyboard.

**Tip** You can take a shortcut when building and running your form: If you simply start your form without first building it, VS .NET will check to see if you made any changes to your form since you last ran it. If you did make a change, VS .NET will first rebuild your project and then run it.

[Figure 6.4](#) shows the running form after the Press Me! button is clicked.



**Figure 6.4:** The running form

Now that you've created and run the form, let's take a look at the code generated by VS .NET for it. The C# code for your form is contained in the file `Form1.cs` file. You'll examine this code in the [next section](#).

## Examining the Code behind the Form

The `Form1.cs` file contains the code for your form. This code is often referred to as the *code behind* your form because you can think of it as being behind the visual design for your form. You can view the code for your form by selecting View > Code, or by pressing the F7 key on your keyboard. [Listing 6.1](#) shows the contents of the `Form1.cs` file.

### **Listing 6.1: Form1.cs**

---

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

```
namespace MyWindowsApplication
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Label myLabel;
        private System.Windows.Forms.Button myButton;
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public Form1()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            //
            // TODO: Add any constructor code after InitializeComponent call
            //
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose(bool disposing)
        {
            if(disposing)
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.myLabel = new System.Windows.Forms.Label();
            this.myButton = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // myLabel
            //
            this.myLabel.Location = new System.Drawing.Point(8, 8);
            this.myLabel.Name = "myLabel";
            this.myLabel.Size = new System.Drawing.Size(288, 184);
            this.myLabel.TabIndex = 0;
            this.myLabel.Text = "label1";
            //
        }
    }
}
```

```

// myButton
//
this.myButton.Location = new System.Drawing.Point(120, 200);
this.myButton.Name = "myButton";
this.myButton.Size = new System.Drawing.Size(72, 24);
this.myButton.TabIndex = 1;
this.myButton.Text = "Press Me!";
this.myButton.Click += new System.EventHandler(this.myButton_Click);
//
// Form1
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(304, 237);
this.Controls.AddRange(new System.Windows.Forms.Control[] {
    this.myButton,
    this.myLabel});
this.Name = "Form1";
this.Text = "My Form";
this.ResumeLayout(false);

}
#endregion

/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
    Application.Run(new Form1());
}

private void myButton_Click(object sender, System.EventArgs e)
{
    myLabel.Text =
        "Is this a dagger which I see before me,\n" +
        "The handle toward my hand? Come, let me clutch thee.\n" +
        "I have thee not, and yet I see thee still.\n" +
        "Art thou not, fatal vision, sensible\n" +
        "To feeling as to sight? or art thou but\n" +
        "A dagger of the mind, a false creation,\n" +
        "Proceeding from the heat-oppressed brain?";

}
}
}

```

---

As you can see, the `Form1` class is derived from the `System.Windows.Forms.Form` class. The `Form` class represents a Windows form.

**Note** The `System.Windows.Forms` namespace contains the various classes for creating Windows applications. Most of the classes in this namespace are derived from the `System.Windows.Forms.Control` class; this class provides the basic functionality for the controls you can place on a form.

The `Form1` class declares two private objects named `myLabel` and `myButton`, which are the label and button controls you added to your form earlier. Because the `myLabel` and `myButton` objects are private, this means that they are accessible only in the `Form1` class.

**Access modifiers** enable you to specify the degree to which a class member is available outside the class. You can also use an access modifier to specify the degree to which the class itself is available.

[Table 6.1](#) shows the access modifiers in decreasing order of availability: `public` is the most accessible, and `private` the least.

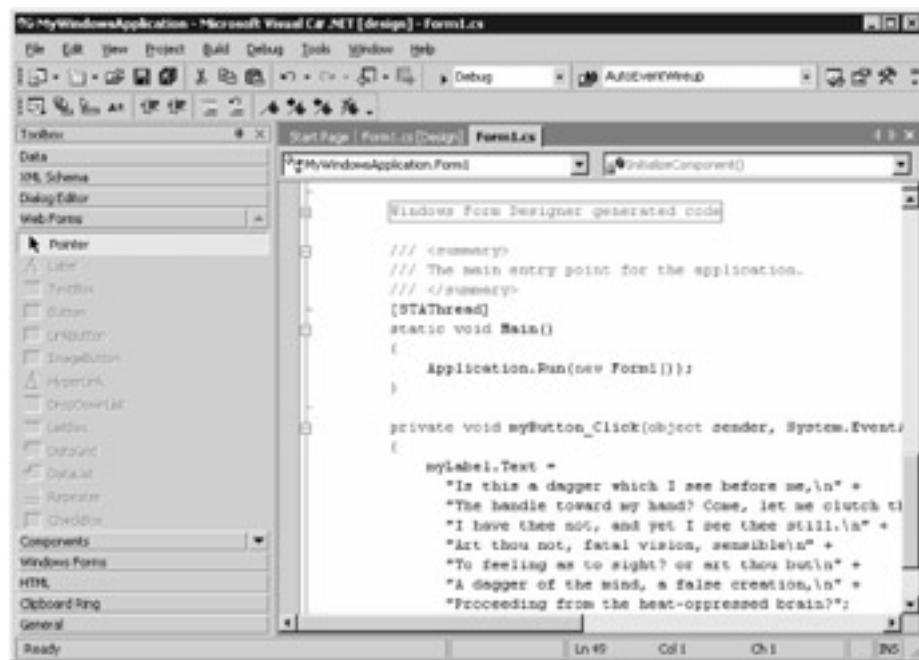
**Table 6.1: ACCESS MODIFIERS**

ACCESS MODIFIER	ACCESSIBILITY
<code>public</code>	Member accessible without restriction.
<code>protected</code>	Member accessible only within the class, a derived class, or class in the same program (or assembly).
<code>internal</code>	Member accessible only within the class or class in the same program (or assembly).
<code>protected</code>	Member accessible only within the class or derived classes.
<code>private</code>	Member accessible only within the class. This is the default.

The `Form1` class constructor calls the `InitializeComponent()` method. This method adds `myLabel` and `myButton` to the form and sets the properties for those objects. These properties include the `Location` (the position in the form), `Name`, `Size`, `TabIndex` (the order in which the control is accessed using the Tab key), and `Text`. For example, the following code sets the properties of `myLabel`:

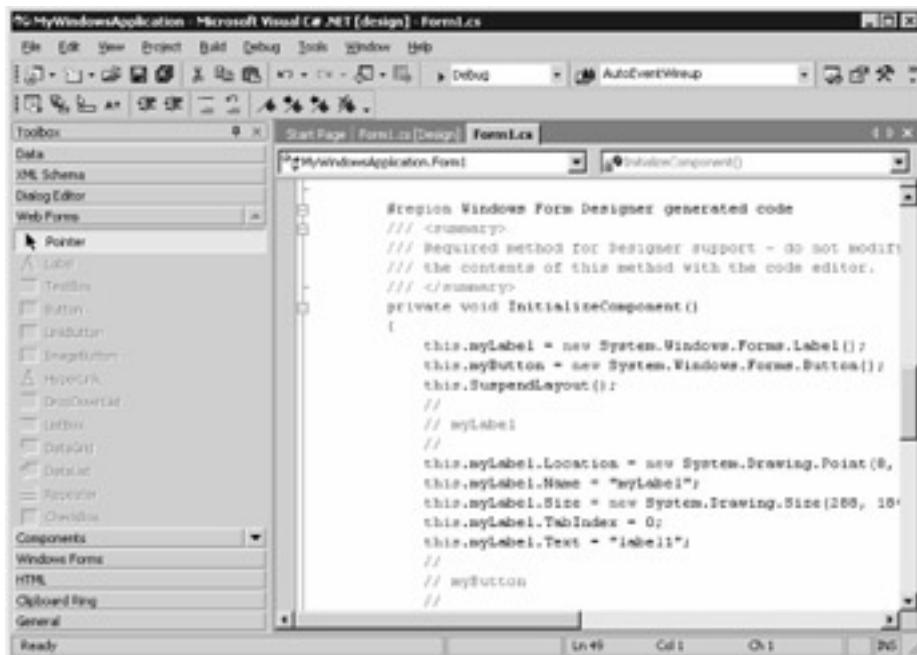
```
this.myLabel.Location = new System.Drawing.Point(8, 8);
this.myLabel.Name = "myLabel";
this.myLabel.Size = new System.Drawing.Size(288, 184);
this.myLabel.TabIndex = 0;
this.myLabel.Text = "label1";
```

You'll notice that the `InitializeComponent()` method is within `#region` and `#endregion` *preprocessor directives*. These directives enclose an area of code that may be hidden in VS .NET's code editor, leaving only the text that immediately follows `#region` visible. [Figure 6.5](#) shows how the hidden code appears in VS .NET.



**Figure 6.5:** Hiding code in VS .NET using the `#region` directive

To view hidden code, all you have to do is to click the plus icon to the left of the code. [Figure 6.6](#) shows the code within the `#region` and `#endregion` directives.



**Figure 6.6:** Viewing hidden code in VS .NET

The `Main()` method runs the form by calling the `Application.Run()` method. The `Application` class is static and provides a number of methods you can use in your Windows programs. Because this class is static, you don't create an instance of this class, and its members are always available within your form. When the `Run()` method is called, your form waits for events from the mouse and keyboard. One example of an event is the clicking of the button in your form.

The `myButton_Click()` method is the method you modified earlier that sets the `Text` property of `myLabel` to a string containing the quote from *Macbeth*. When `myButton` is clicked, the `myButton_Click()` method is called and the text in `myLabel` is changed; you saw this when you ran your form earlier.

In the [next section](#), you'll learn about the VS .NET Solution Explorer.

## Working with the Solution Explorer

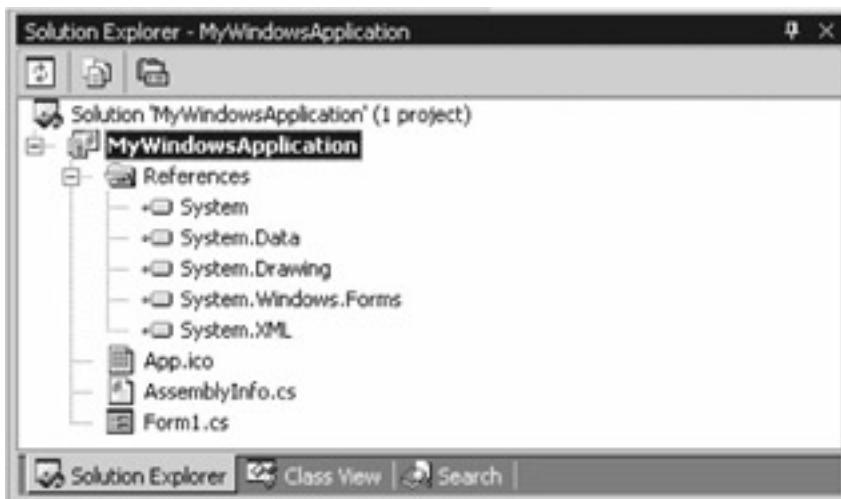
You can use the VS .NET Solution Explorer to view the items in your project, such as the namespace for your project. Of course, a project may contain more than one namespace. To view the Solution Explorer, you select `View > Solution Explorer`.

**Tip** You can also view the Solution Explorer by pressing `Ctrl+Alt+L` on your keyboard.

You can use Solution Explorer to view the following items in a project's namespace:

- **References** References include other namespaces and classes to which your form's code refers. You can employ the `using` statement to reference other namespaces and classes.
- **Icon File** An icon file has the extension `.ico`. You use an icon file to set the image displayed in Windows Explorer for your application.
- **Assembly File** An assembly file contains the metadata for your application's assembly. An assembly is collection of code for your application.
- **Code Files** A code file is a program source file, such as the code for a form. You saw an example of this in the earlier "[Examining the Code behind the Form](#)" section.

[Figure 6.7](#) shows the Solution Explorer for this example.



**Figure 6.7:** The Solution Explorer

As you can see from [Figure 6.7](#), you can expand or collapse the items shown in the Solution Explorer by clicking the plus or minus icon, respectively. You can also display the properties for an item in Solution Explorer: When you have the Properties window displayed, selecting an item in Solution Explorer will also display the properties for that item. For example, in [Figure 6.7](#), the properties for the `MyWindowsApplication` project are displayed; you can see that the project file is `MyWindowsApplication.csproj`.

In the [next section](#), you'll learn about the VS .NET Class View.

## Working with the Class View

You use the VS .NET Class View to examine the classes, methods, and objects in your project. To see the Class View, you select **View > Class View**.

**Tip** You can also see the Class View by pressing **Ctrl+Shift+C** on your keyboard.

[Figure 6.8](#) shows the Class View for the example.



**Figure 6.8:** The Class View

As you can see from [Figure 6.8](#), you can view the classes, methods, and objects for the example. You can also view the properties for a selected item in the Properties window. For example, [Figure 6.8](#) also shows the properties for the `Form1` class.

Next, you'll be introduced to the other types of Windows controls.

## Using Windows Controls

[Table 6.2](#) lists the commonly used Windows form controls that you can pick from the Windows Forms section of the Toolbox. You can place any of these controls on your Windows form.

**Table 6.2: COMMONLY USED WINDOWS FORM CONTROLS**

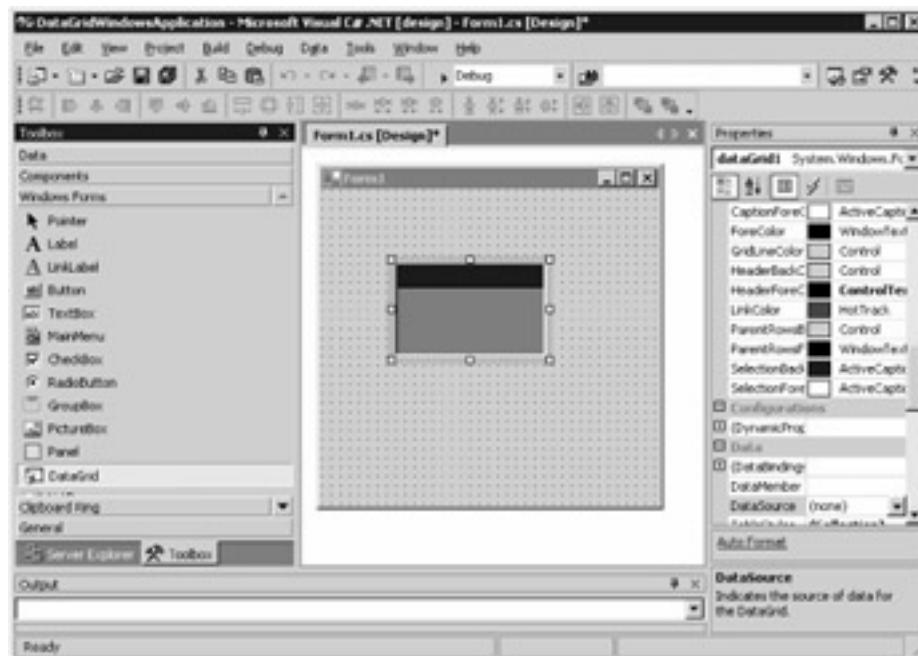
CONTROL	DESCRIPTION
Label	Displays text. You set the text that you want to display using the <code>Text</code> property.
LinkLabel	Similar to a label, except it displays hyperlinks. You set the hyperlink that you want to display using the <code>Text</code> property. You set the navigation via the <code>LinkClicked</code> event.
Button	A clickable button. The <code>Text</code> property determines the text shown on the button.
TextBox	A box containing text that the user of your form may edit at runtime. The <code>Text</code> property contains the text contained in the <code>TextBox</code> .
MainMenu	A menu you can add to a form.
CheckBox	A check box contains a Boolean <code>true/false</code> value that is set to <code>true</code> by the user if they check the box. The <code>Checked</code> property indicates the Boolean value.
RadioButton	A radio button contains a Boolean <code>true/false</code> value that is set to <code>true</code> by the user if they click the button. The <code>Checked</code> property indicates the Boolean value.
GroupBox	A group box allows you to group related controls together. For example, you can group related radio buttons together. Most importantly, it allows you to treat multiple controls as a group.
PictureBox	A picture box displays an image that you set using the <code>Image</code> property.
Panel	A container for other controls such as radio buttons or group boxes.
DataGridView	A grid containing data retrieved from a data source, such as a database. You set the data source using the <code>DataSource</code> property.
ListBox	A list of options. You set the list of options using the <code>Add( )</code> method of the <code>Items</code> collection property.
CheckedListBox	Similar to a list box except that a check mark is placed to the left of each item in the list. The check mark allows the user to select the items via a check box, as opposed to multiselecting with the Shift and/or Ctrl keys.
ComboBox	Combines an editable field with a list box.

In the [next section](#), you'll learn how to use a `DataGridView` control to access the rows in a database table.

## Using a *DataGrid* Control to Access a Database

In this section, you'll learn how to use a *DataGrid* control to access the rows in a database table. Follow these steps to create a *DataGridView* using VS .NET:

1. First, select File > New Project. In the New Project dialog box, select Windows Application, and enter **DataGridWindowsApplication** in the Name field.
2. Click OK to continue. Your new project will contain a blank form.
3. Add a *DataGrid* control to the form by selecting View > Toolbox, selecting a *DataGrid*, and dragging it to your form. [Figure 6.9](#) shows a form with a *DataGrid*. Make your *DataGrid* almost as big as your form by dragging the corners of your *DataGrid* out to the end of your form.



**Figure 6.9:** Form with a *DataGrid*

Next, you'll add a *SqlConnection* object and a *SqlDataAdapter* object to your form.

**Note** You use a *SqlConnection* object to connect to a SQL Server database, and a *SqlDataAdapter* object to move rows between SQL Server and a *DataSet* object. You'll learn the details on how to pull rows from the database into a *DataSet* in [Chapter 10](#), and how to push changes made in a *DataSet* to the database in [Chapter 11](#).

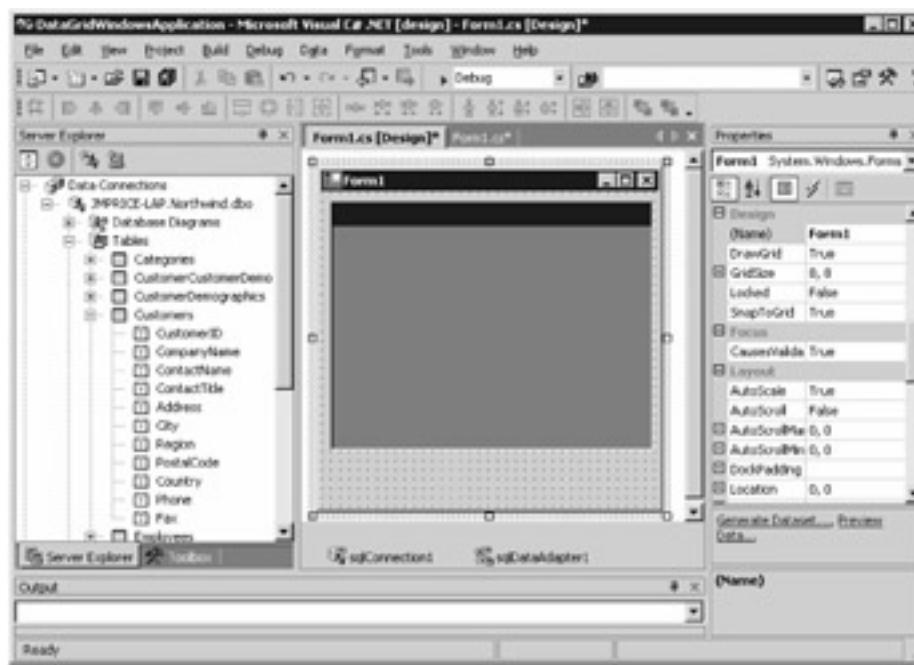
You can drag a table from a SQL Server database onto your form and have the *SqlConnection* and *SqlDataAdapter* objects created in one step. You use Server Explorer for this. With databases that do not show up in Server Explorer, your choices are limited. You can use the controls in Data category of the Toolbox to drag each item to your form, and then set properties for each data object with the Properties window.

**Note** To open Server Explorer, select View Server Explorer, or press Cntl+Alt+S.

To add a *SqlConnection* and *SqlDataAdapter* object to your form, perform the following steps:

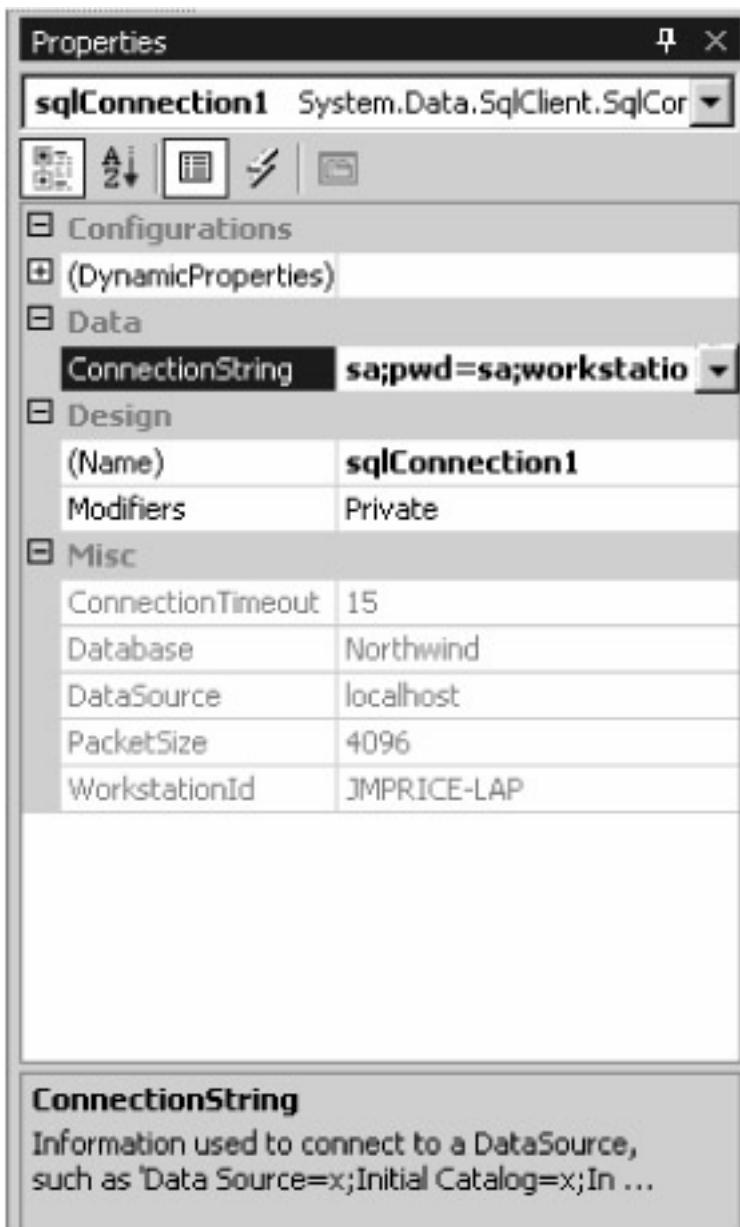
1. Open Server Explorer.
2. Open the connection to your SQL Server Northwind database (or create a new connection if necessary by right-clicking on the Data Connections node and selecting Add Connection, and entering the *sa* username and password for your Northwind database; you might need to get the password from your database administrator).
3. Drill down to the *Customers* table in the Northwind database and drag it to your form. This creates a *SqlConnection* object

named `sqlConnection1` and a `SqlDataAdapter` object named `sqlDataAdapter1`, as shown in [Figure 6.10](#).



**Figure 6.10:** Form with `SqlConnection` and `SqlDataAdapter` objects

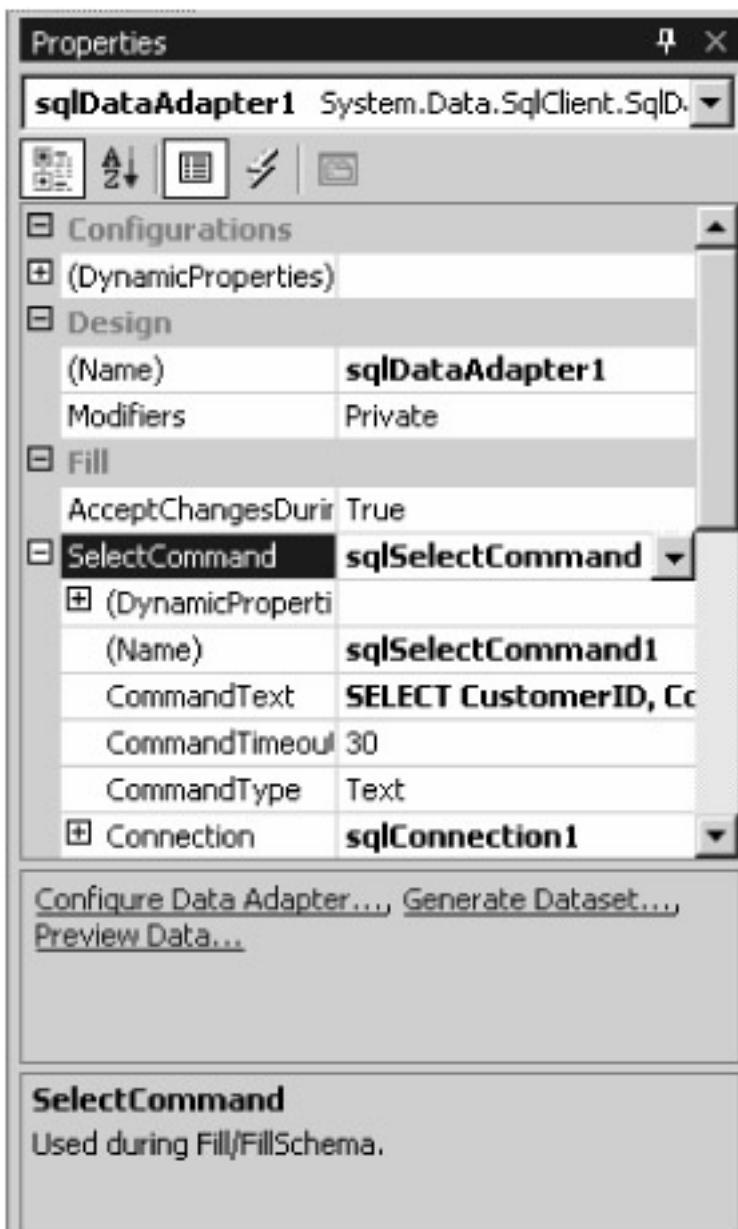
4. Click your `sqlConnection1` object to display the properties for this object in the Properties window.
5. To enable `sqlConnection1` to access the database, you need to set the password for the connection. To do this, add a substring containing `pwd` to the `ConnectionString` property of `sqlConnection1`. Go ahead and add `pwd=sa` (you might need to get the password for the `sa` user from your database administrator) to the `ConnectionString` property, as shown in [Figure 6.11](#).



**Figure 6.11:** Setting the `ConnectionString` property for the `sqlConnection1` object

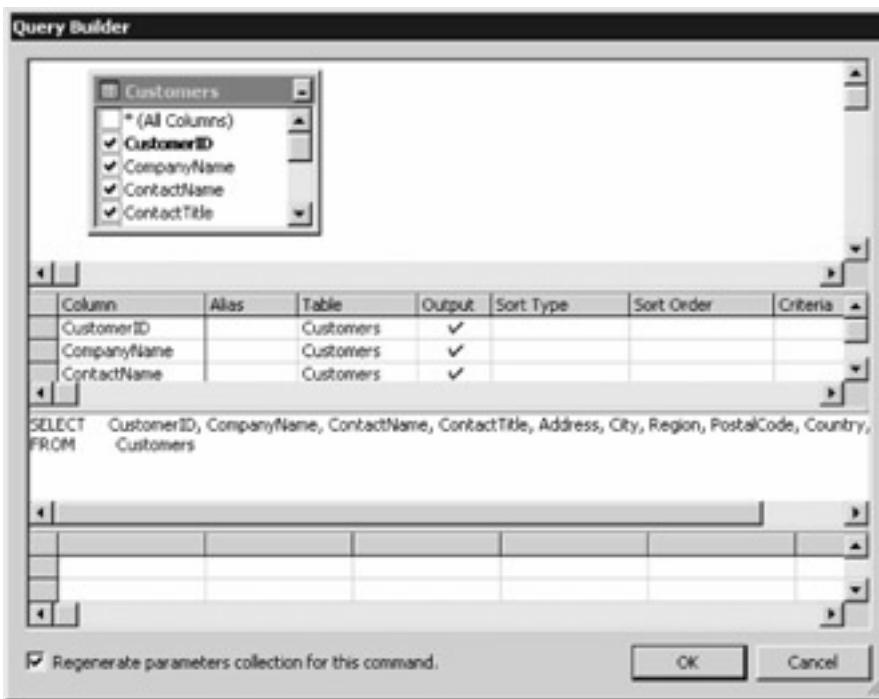
Next, you'll modify the SQL SELECT statement used to retrieve the rows from the `Customers` table:

1. Click your `sqlDataAdapter1` object to display the properties for this object.
2. Click the addition icon to the left of the `SelectCommand` property to display the dynamic properties; one of the dynamic properties is the `CommandText` property, which contains the `SELECT` statement (see [Figure 6.12](#)).



**Figure 6.12:** SelectCommand property for the `sqlDataAdapter1` object

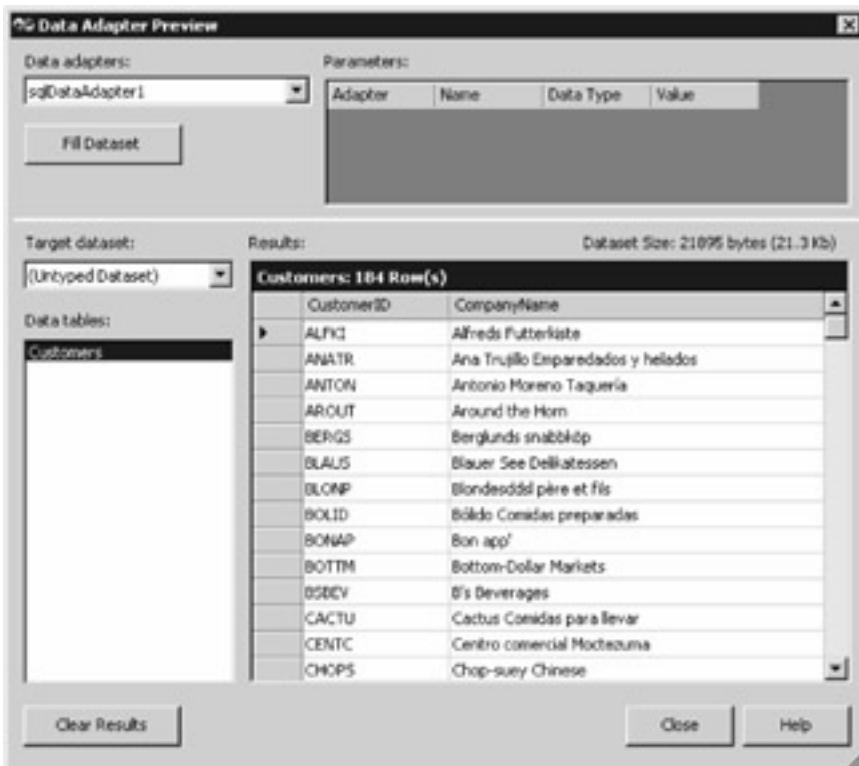
3. Click CommandText, and then click the button with the ellipsis to display the Query Builder, as shown in [Figure 6.13](#).

**Figure 6.13:** The Query Builder

4. You use the Query Builder to define SQL statements. You can type in the SQL statement, or you can build it up visually. Make sure all the columns are selected from the `Customers` table using the `Customers` box at the top left of the Query Builder.
5. Click OK to continue.

To check the rows returned by this `SELECT` statement, perform the following steps:

1. Click the Preview Data link near the bottom of the Properties window. This displays the Data Adapter Preview dialog box.
2. In the Data Adapter Preview dialog box, click the Fill Dataset button to run the `SELECT` statement, as shown in [Figure 6.14](#).

**Figure 6.14:** Previewing the rows retrieved by the `SELECT` statement

3. Click the Close button to close the Data Adapter Preview dialog box.

Next, you need to create a `DataSet` object. You use a `DataSet` object to store local copy of the information stored in the database. A `DataSet` object can represent database structures such as tables, rows, and columns, among others. In this example, you'll use a `DataSet` object to store the rows from the `Customers` table:

1. Click an area of your form outside the `DataGrid`.
2. Click the Generate Dataset link near the bottom of the Properties window. This displays the Generate Dataset dialog box.
3. Select the New radio button and make sure the field to the right of this radio button contains `DataSet1`, as shown in [Figure 6.15](#).



**Figure 6.15:** Entering the `DataSet` details in the Generate Dataset dialog box

4. Click the OK button to continue. This adds a new `DataSet` object named `dataset11` to your form.

Next, you'll need to set the `DataSource` property of your `DataGrid` to your `DataSet` object. This sets the source of the data for your `DataGrid`, allowing the rows from your `DataSet` to be displayed in your `DataGrid`. To set the `DataSource` property, you perform the following steps:

1. Click your `DataGrid` object and set the `DataSource` property to `dataset11.Customers`.
2. Now, you'll add a button that will fill `sqlDataAdapter1` with the rows retrieved by your `SELECT` statement. Select Button from the Toolbox and drag it onto your form to a position just below your `DataGrid`.
3. Set the `Text` property for your button to Run `SELECT` in the Properties window.

To populate `sqlDataAdapter1` with the rows retrieved by the `SELECT` statement, you need to call the `Fill()` method for this object. You'll call this method when the button is clicked. To add the required code, perform the following steps:

1. Double-click the button you added earlier. This opens the code window and positions the cursor in the `button1_Click()` method.
2. Enter the following code in this method:

```
dataset11.Clear();
sqlDataAdapter1.Fill(dataset11, "Customers");
```

**Note** You could also call the `Fill()` method in the `Form1_Load` event. This event occurs when the form is initially loaded.

Next, add another button that will allow you to save any changes you make to the rows in the `DataGrid`:

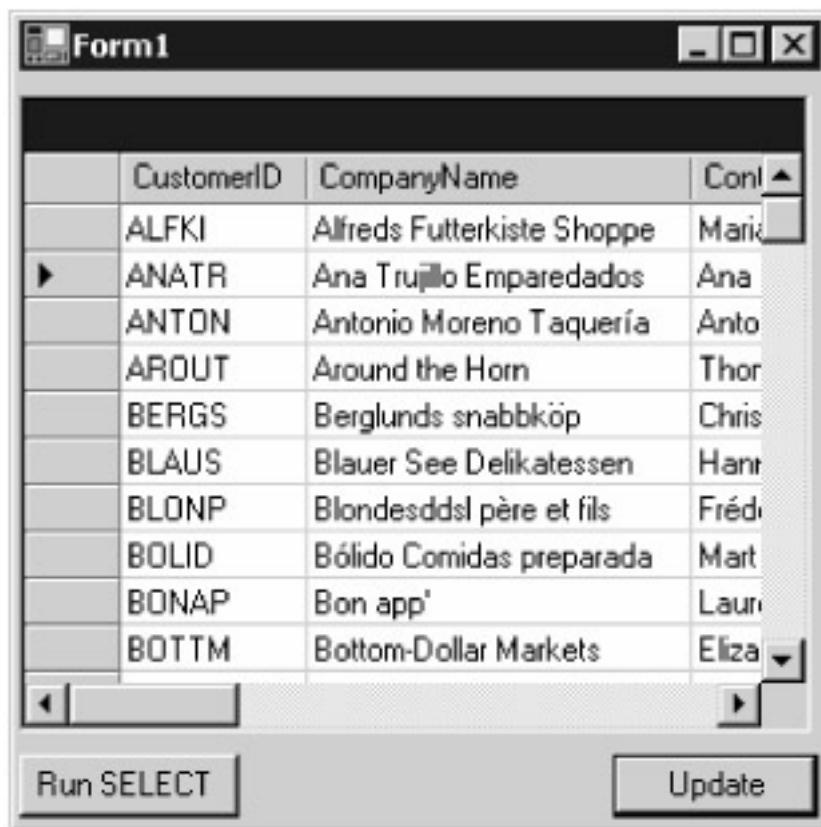
1. Go ahead and add another button and set the `Text` property of this button to `Update`.
2. Double-click this button and add the following statement to the `button2_Click()` method: `sqlDataAdapter1.Update(dataSet1, "Customers")`;

This statement updates a row with the new column values you enter in your `DataGrid`.

You've now finished your form. Build the project by selecting `Build > Build Solution`.

Finally, you're ready to run your form! Perform the following steps:

1. Select `Debug > Start without Debugging` to start your form.
2. Click the Run `SELECT` button on your form to run your `SELECT` statement. This retrieves the rows from the `Customers` table and displays them in the `DataGrid` of your form.
3. Modify the `CompanyName` column of the first row to `Alfreds Futterkiste Shoppe` and click the `Update` button; this commits the change you made to the row in the `Customers` table (see [Figure 6.16](#)).



**Figure 6.16:** The running form

4. Reset the `CompanyName` for the first row back to the original by removing `Shoppe` from the end and clicking the `Update` button again.

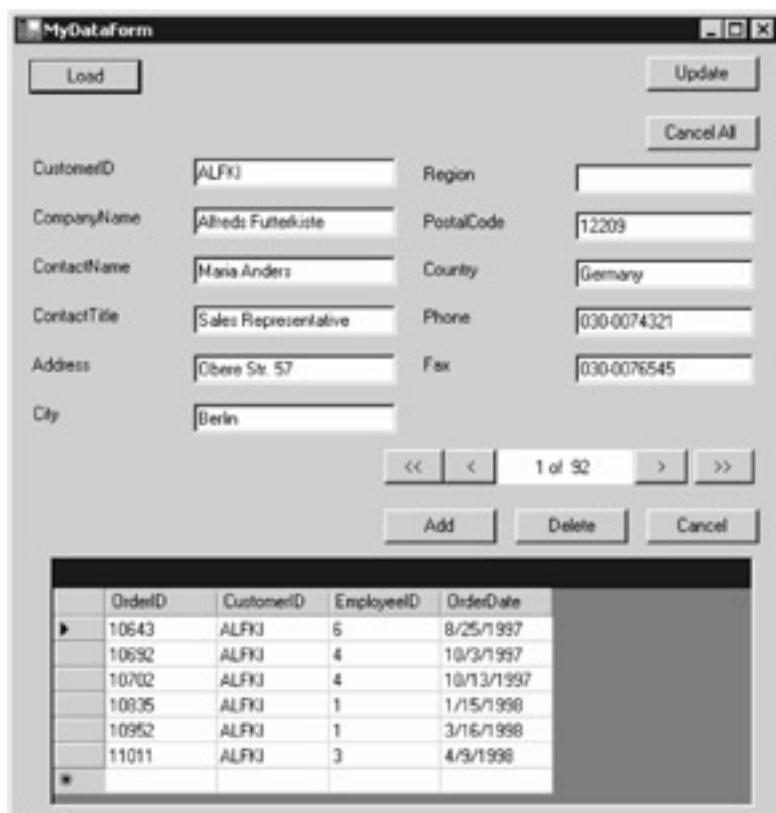
In the [next section](#), you learn how to use the VS .NET Data Form Wizard to create a more advanced Windows application that accesses the SQL Server Northwind database.

## Using the Data Form Wizard to Create a Windows Form

In this section, you'll use the VS .NET Data Form Wizard to create a Windows application that accesses both the `Customers` and `Orders` tables. The `Orders` table contains rows that represent orders placed by the customers.

The rows in the `Orders` table are related to the rows in the `Customers` table through a foreign key: The `Orders` table contains a column named `CustomerID` that is a foreign key to the `CustomerID` column of the `Customers` table (`CustomerID` is the primary key for the `Customers` table). The use of the foreign key defines a parent-child relationship between the `Customers` and `Orders` tables.

The form you'll create will display a row from the `Customers` table, along with any related rows from the `Orders` table. To give you a clear idea of the final goal, Figure 6.17 shows the completed form up and running. Notice that the top part of the form shows the details for the row from the `Customers` table where the `CustomerID` is `ALFKI`; the bottom part of the form contains a `DataGrid` control that displays the rows from the `Orders` table for that customer. When you move to the next row in the `Customers` table, the rows from the `Orders` table for that customer are automatically displayed in the `DataGrid`.



**Figure 6.17:** The running form

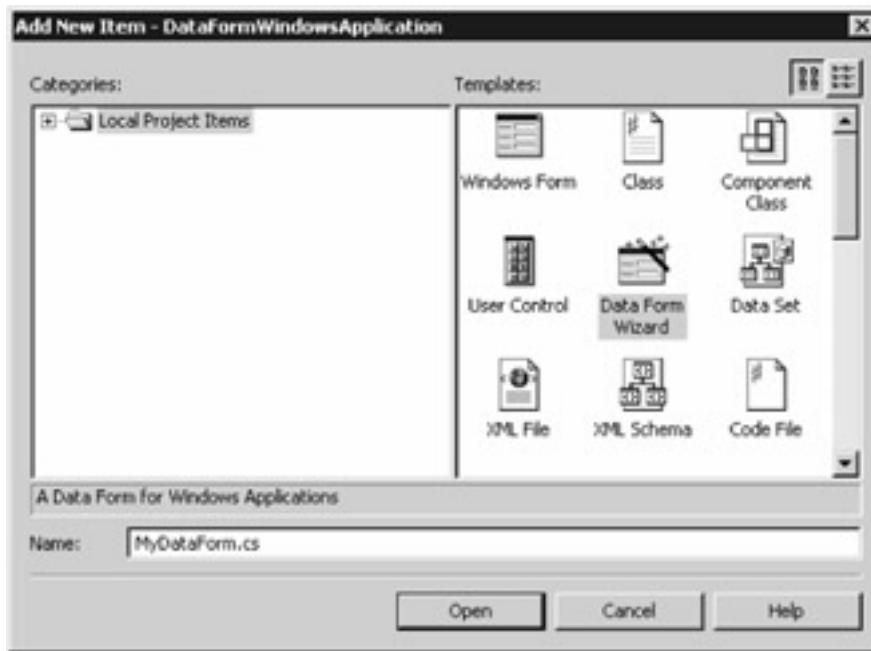
Perform these steps to begin build the form:

1. Select File > New Project.
2. In the New Project dialog box, select Empty Project, and enter **DataFormWindowsApplication** in the Name field. Because you'll be adding a new form to your new application shortly, there's no need to have VS .NET generate the usual blank form for you; that's why you're creating an empty project.
3. Click OK to continue. VS .NET will create a new empty project for you.

Next, you'll use the Data Form Wizard to create a form that accesses the `Customers` and `Orders` tables in the Northwind database:

1. Select Project > Add New Item.
2. Select Data Form Wizard from the Templates section on the right, enter the Name of the form as **MyDataForm.cs**, and click

Open (see [Figure 6.18](#)). You'll then see the welcome page for the Data Form Wizard.



**Figure 6.18:** Adding a data form using the Data Form Wizard

3. Click the Next button to proceed.
4. Now you enter the **DataSet** object you want to use in your form. You can pick an existing **DataSet**, or you can create a new one. Because this is a new project, you'll be creating a new **DataSet**. Enter **myDataSet** as the name for your **DataSet**, as shown in [Figure 6.19](#).



**Figure 6.19:** Entering the name of the new **DataSet**

5. Click the Next button to go to the next step.
6. You must now choose a data connection to access the database. You can pick an existing connection, or you can create a new one. Select your connection, as shown in [Figure 6.20](#)-of course, your connection name will be different.



**Figure 6.20:** Choosing the data connection

7. Click the Next button to continue.
8. You now log in to the database by specifying the password for the database user. You used the `sa` user when creating the database connection earlier, and you therefore need to reenter the password for that user, as shown in [Figure 6.21](#).



**Figure 6.21:** Logging in to the SQL Server Northwind database

9. Click the OK button to proceed.

You now select the database tables or views you want to use in your form. The area on the bottom left of the dialog box shows the tables and views you can access using your form. The area on the bottom right shows the tables and views you've added. You add a table or view to your form by selecting it from the area on the left and clicking the right-arrow button.

**Tip** You can also double-click on the table or view to add them to your form.

When you do this, the table or view moves to the right, indicating that you've selected them for use in your form. If you decide you don't want to use a table or view, you can unselect them using the left-arrow button. You can also double-click the table or view to unselect them. Select the `Customers` and `Orders` tables, as shown in [Figure 6.22](#). Click the Next button to proceed.



**Figure 6.22:** Selecting the `Customers` and `Orders` tables for use in the form

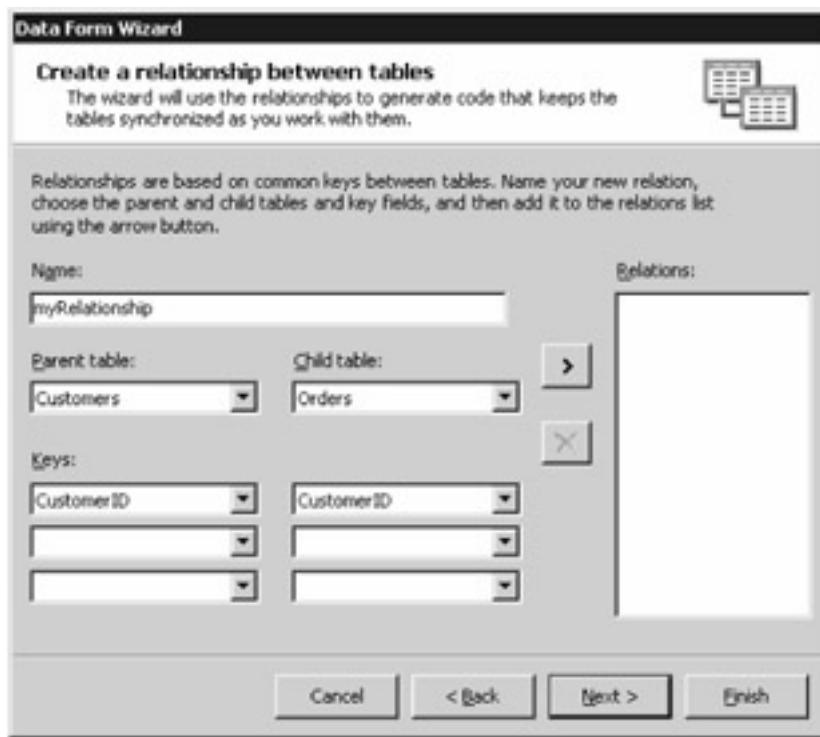
Because you selected two tables—`Customers` and `Orders`—your next step is to define a relationship between those tables. This relationship is used in your form to synchronize navigation between the rows in the `Customers` table with the rows in the `Orders` table. When you move to a new row in the `Customers` table, the rows from the `Orders` table will be displayed in your form. Set the following in the dialog box (as shown in [Figure 6.23](#)):

1. Enter **myRelationship** in the Name field.
2. Select `Customers` as the parent table.
3. Select `Orders` as the child table.
4. Select `CustomerID` as the key for each table.

**Warning** To add the relationship to your form, click the right-arrow button. If you don't do this, your relationship won't be added to your form.

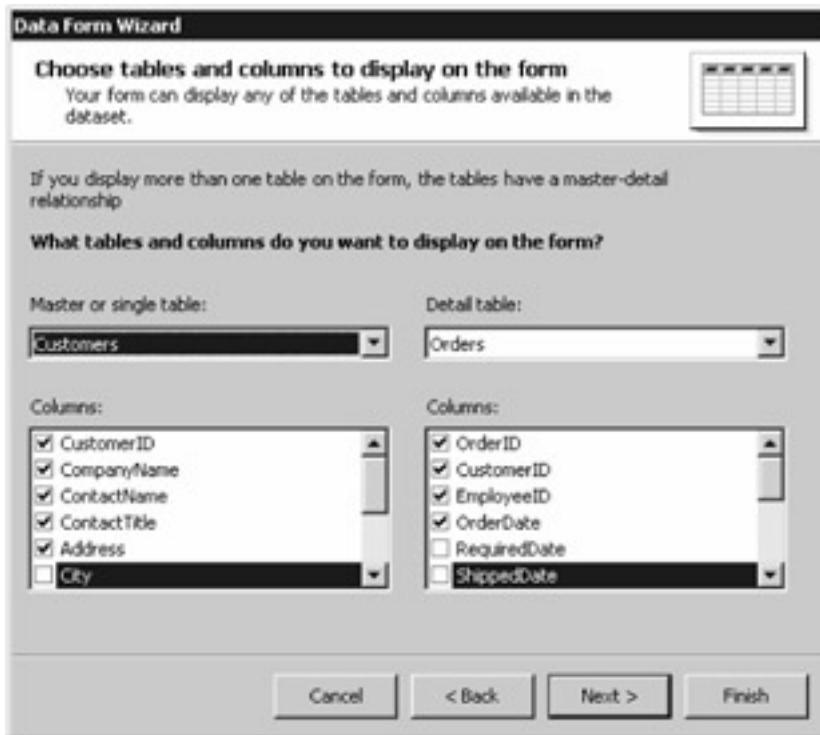
5. Click the Next button to continue.
6. Select the columns from the tables you want to display in your form. Because you added the `Customers` and `Orders` tables to your form, you'll be selecting the columns to display from these two tables. By default, all the columns from the tables are selected. You won't be displaying all the columns from the `Customers` or the `Orders` table. Unselect the `City` column for the `Customers` table. (Later, you'll see how to add this column to your form manually.)
7. Deselect the following columns for the `Orders` table:

RequiredDate	ShipAddress
ShippedDate	ShipCity
ShipVia	ShipRegion
Freight	ShipPostalCode
ShipName	ShipCountry

**Figure 6.23:** Creating a relationship between two tables

**Note** Remember: You're unselecting these columns, so you uncheck the columns for the *Orders* table.

[Figure 6.24](#) shows the completed dialog box with the selected columns to display from each table.

**Figure 6.24:** Selecting the columns to display from each table

8. Click the Next button to proceed.
9. Select the display style for the rows (also known as *records*) in the parent table that are displayed in your form. You can display the rows in a grid, or you can display each column using a separate control. You'll use a separate control for the columns, so select the Single Record in individual controls radio button. The other check boxes in the dialog box allow you pick the controls you want to add to your form. These controls affect the rows in the master table, and you can add the following controls to your form:

**Note** In this example, the parent table is the *Customers* table, and the child table is the *Orders* table. The rows for the child table are displayed in a *DataGrid* control.

- **Cancel All** The Cancel All button allows you to undo any changes you've made to the current row.
- **Add** The Add button allows you to add a new row.
- **Delete** The Delete button allows you to delete the current row.
- **Cancel** The Cancel button allows you to cancel a change made to the current row.
- **Navigation Controls** The Navigation controls consist of four buttons that allow you to move to first row, the previous row, the next row, and the last row. An indicator is also displayed to show the current row.

[Figure 6.25](#) shows the completed dialog box.



**Figure 6.25:** Choosing the display style

10. You've now completed all the steps in the Data Form Wizard. Click the Finish button to create your form. VS .NET will now display the new form, as shown in [Figure 6.26](#).



**Figure 6.26:** The completed form

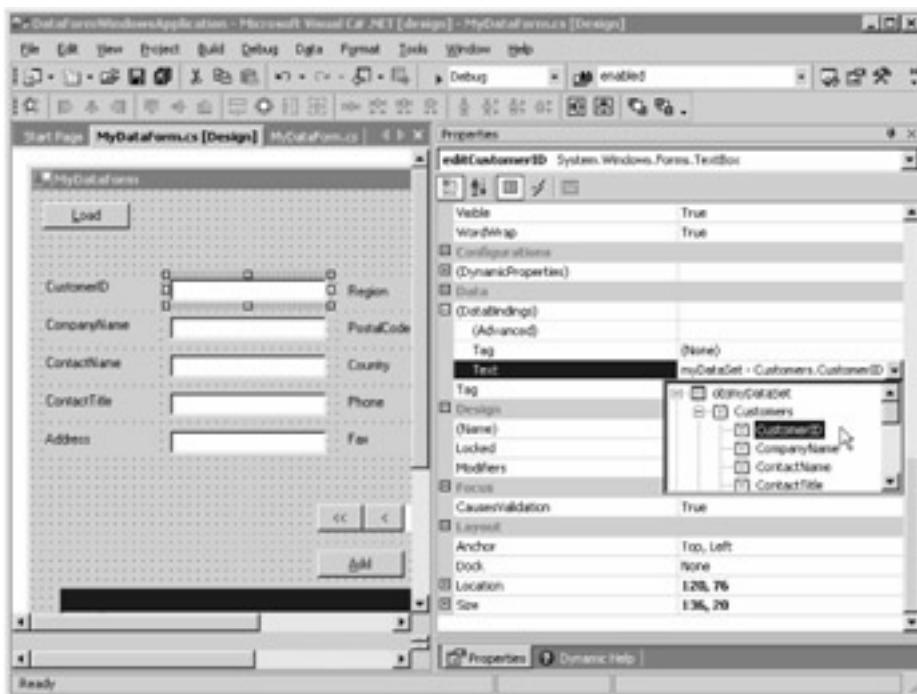
The managed provider objects in your form use the OLE DB classes contained in the `System.Data.OleDb` namespace—even though a SQL Server database is used. These objects work with any OLE DB-compliant database. The code would be more efficient if the managed provider classes in the `System.Data.SqlClient` namespace were used instead; these classes are specifically for use with a SQL Server database. This is the price of having the VS .NET wizard generate the code for you.

In the [next section](#), you'll learn how the text-box controls in your form access the columns in the `Customers` table.

## Data Binding

Each text-box control in the upper part of your form is bound to a column in the `Customers` table using a process known as ***data binding***. When a control is bound to a column in a `DataSet` object, the value for that column is displayed in the control through the `Text` property in the `DataBindings` group. The `Text` property in the `DataBindings` group sets the text displayed in a control. To examine or set the data binding for a control, you select the control in the form designer and expand the `DataBindings` properties in the Properties window. You'll see these properties in the Data area of the Properties window.

Next, you'll see how the text box for the `CustomerID` is set. Select the text box to the right of the `CustomerID` label in your form; this text box is named `editCustomerID`. Make sure the `DataBindings` properties are expanded in the Properties window. Finally, click the drop-down list for the `Text` property to view the current column to which the text box is bound. As you can see from [Figure 6.27](#), `editCustomerID` is bound to the `CustomerID` column of the `Customers` table. This means that when you run the form and load data from the database, the `CustomerID` column value will be displayed in the `editCustomerID` text box.



**Figure 6.27:** The editCustomerID text box is bound to the CustomerID column

In the [next section](#), you'll add a label and a text-box control to display the City column in your form.

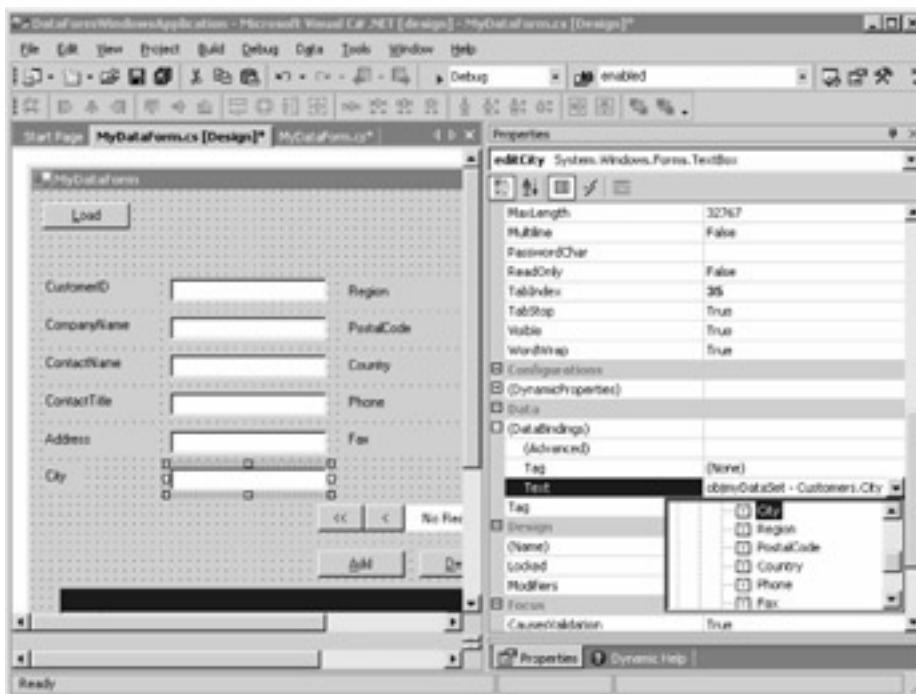
## Adding Controls to the Form

When you ran the Data Form Wizard earlier to create your form, you'll recall that I told you to unselect the City column of the Customers table so that it didn't appear on your form. I asked you to do this so that you can now see how to manually add a control and bind it to the City column. That way, you can see how to build your own forms that access the database.

Follow these steps to add a label and a text box to your form:

1. Add a label below the Address label in your form. Set the Name property for your new label to lblCity.
2. Set the Text property for your label to City.
3. Next, add a text box below the editAddress text box.
4. Set the Name property for your new text box to editCity.
5. Remove the current text from the Text property so that no default text is shown in the control.

Next, you need to bind editCity to the City column of the Customers table. To do this, you open the DataBindings properties and set the text property by selecting City from the Customers table, as shown in [Figure 6.28](#).



**Figure 6.28:** Binding the City column to the editCity text box

In the [next section](#), you'll add a `Main()` method to the code of your form.

## Adding the `Main()` Method

As you know, all programs must have a `Main()` method. The `Main()` method is executed when you run your program. In this section, you'll add a `Main()` method to your form. To do this, select `View > Code`, and add the following `Main()` method inside your `MyDataForm` class (a good place to add `Main()` would be at the start of your `MyDataForm` class after the open curled bracket `{}`):

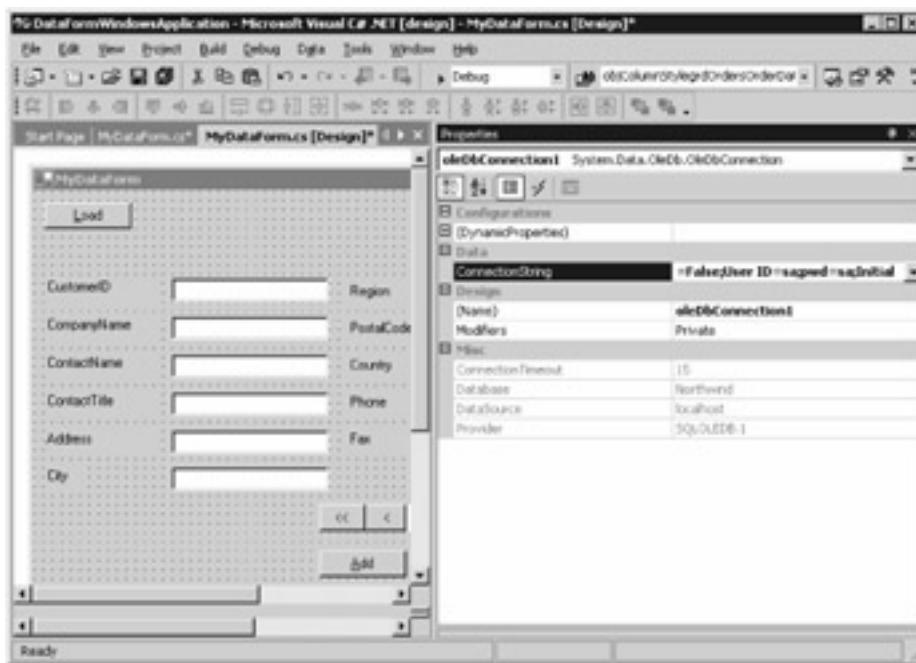
```
public class MyDataForm : System.Windows.Forms.Form
{
    public static void Main()
    {
        Application.Run(new MyDataForm());
    }
    ...
}
```

This code creates a new object of the `MyDataForm` class, causing your form to be displayed on the screen.

## Setting the Password

Before you can run your form, you need to set the password for the database user in the `ConnectionString` property of the data connection object. This object was automatically created by VS .NET when you ran the Data Form Wizard, and the object has the default name `oleDbConnection1`.

To modify the `ConnectionString` property for `oleDbConnection1`, select `oleDbConnection1` from the drop-down list in the Properties window. Go ahead and add the text `pwd=sa` in the `ConnectionString` property, as shown in [Figure 6.29](#).

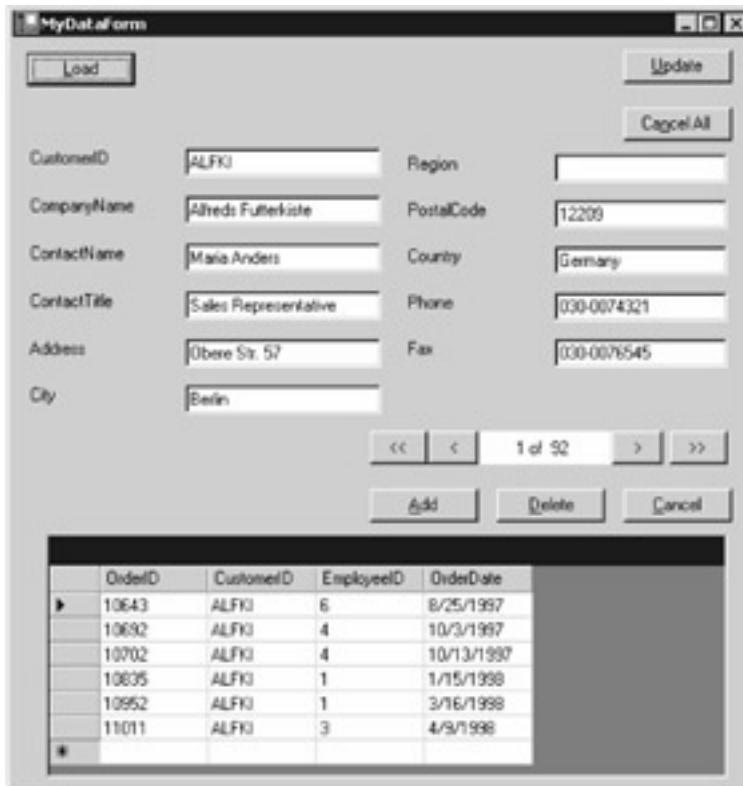


**Figure 6.29:** Setting the ConnectionString property

You're now ready to run your form.

## Running the Form

To run your form, select Debug > Start without Debugging. [Figure 6.30](#) shows the running form. You click the Load button to display the rows from the Customers and Orders tables in your form.



**Figure 6.30:** The running form

**Note** You'll see a Windows console appear in the background. Don't worry about it.

Notice that the top part of the form shows the details for the row from the `Customers` table where the `CustomerID` is `ALFKI`; the bottom part of the form contains a `DataGridView` control that displays the rows from the `Orders` table for that customer. When you move to

the next row in the `Customers` table, the rows from the `Orders` table for that customer are automatically displayed in the `DataGridView`.

Feel free to try out the other buttons on your form to add, modify, and delete rows in the `Customers` table. You can also use the `DataGridView` control to add, modify, and delete rows from the `Orders` table for the current customer.

**Team LiB**

PREVIOUS NEXT

**Team LiB**[◀ PREVIOUS](#)[NEXT ▶](#)

## Summary

In this chapter, you learned how to create Windows programs using Visual Studio .NET. Windows provides graphical items such as menus, text boxes, radio buttons, and text boxes that allow you to build a visual interface that users of your programs will find easy to use.

You saw how to use a `DataGridView` control to access the rows in a database table. You also learned how to use the Visual Studio .NET Data Form Wizard to create a Windows application that accesses both the `Customers` and `Orders` tables at the same time.

In Part II, "Fundamental Database Programming with ADO.NET," you'll examine the details of the various ADO.NET classes and you'll see how to work in depth with ADO.NET.

**Team LiB**[◀ PREVIOUS](#)[NEXT ▶](#)

**Team LiB**

## Part 2: Fundamental Database Programming with ADO.NET

### Chapter List

[Chapter 7:](#) Connecting to a Database

[Chapter 8:](#) Executing Database Commands

[Chapter 9:](#) Using *DataReader* Objects to Read Results

[Chapter 10:](#) Using *Dataset* Objects to Store Data

[Chapter 11:](#) Using *DataSet* Objects to Modify Data

[Chapter 12:](#) Navigating and Modifying Related Data

[Chapter 13:](#) Using *DataView* Objects

**Team LiB**

## Chapter 7: Connecting to a Database

### Overview

In this chapter, you'll learn the details on connecting to a database using objects of a `Connection` class. There are three `Connection` classes: `SqlConnection`, `OleDbConnection`, and `OdbcConnection`. You use an object of the `SqlConnection` class to connect to a SQL Server database, an object of the `OleDbConnection` class to connect to any database that supports OLE DB, such as Oracle or Access, and an object of the `OdbcConnection` class to connect to any database that supports ODBC. Ultimately, all communication with a database is done through a `Connection` object.

Featured in this chapter:

- Using a `SqlConnection` object to connect to a SQL Server database
- Connection pooling
- Getting the state of a `Connection`
- [Using `Connection` events](#)
- [Creating a `Connection` object using Visual Studio .NET](#)

## Understanding the *SqlConnection* Class

You use an object of the `SqlConnection` class to connect to a SQL Server database, and this object handles the communication between the database and your C# program.

**Note** Although the `SqlConnection` class is specific to SQL Server, many of the properties, methods, and events in this class are the same as those for the `OleDbConnection` and `OdbcConnection` classes. If a property or method is specific to `SqlConnection`, it says so in the Description column of the tables shown in this section. You can look up the exact properties, methods, and events for a specific class using the .NET online reference. You saw how to do that in [Chapter 1](#), "Introduction to Database Programming with ADO.NET."

[Table 7.1](#) shows some of the `SqlConnection` properties.

**Table 7.1: `SqlConnection` PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
<code>ConnectionString</code>	<code>string</code>	Gets or sets the string used to open a database.
<code>ConnectionTimeout</code>	<code>int</code>	Gets the number of seconds to wait while trying to establish a connection to a database. The default is 15 seconds.
<code>Database</code>	<code>string</code>	Gets the name of the current database (or the database to be used once the connection to the database is made).
<code>DataSource</code>	<code>string</code>	Gets the name of the database server.
<code>PacketSize</code>	<code>int</code>	Gets the size (in bytes) of network packets used to communicate with SQL Server. This property applies only to the <code>SqlConnection</code> class. The default is 8,192 bytes.
<code>ServerVersion</code>	<code>string</code>	Gets a string containing the version of SQL Server.
<code>State</code>	<code>ConnectionState</code>	Gets the current state of the connection: <code>Broken</code> , <code>Closed</code> , <code>Connecting</code> , <code>Executing</code> , <code>Fetching</code> , or <code>Open</code> . These states are covered later in the "Getting the State of a Connection" section.
<code>WorkstationId</code>	<code>string</code>	Gets a string that identifies the client computer that is connected to SQL Server. This property applies only to the <code>SqlConnection</code> class.

[Table 7.2](#) shows some of the `SqlConnection` methods.

**Table 7.2: `SqlConnection` METHODS**

METHOD	RETURN TYPE	DESCRIPTION
<code>BeginTransaction()</code>	<code>SqlTransaction</code>	Overloaded. Begins a database transaction.
<code>ChangeDatabase()</code>	<code>void</code>	Changes the current database for an open connection.
<code>Close()</code>	<code>void</code>	Closes the connection to the database.
<code>CreateCommand()</code>	<code>SqlCommand</code>	Creates and returns a command object.
<code>Open()</code>	<code>void</code>	Opens a database connection with the property settings specified by the <code>ConnectionString</code> property.

You can use events to allow one object to notify another object that something has occurred. For example, when you click a mouse button in a Windows application, an event occurs, or is *fired*. [Table 7.3](#) shows some of the `SqlConnection` events. You'll learn how to use these

events later in the "[Using Connection Events](#)" section.

**Table 7.3: `sqlConnection EVENTS`**

EVENT	EVENT HANDLER	DESCRIPTION
StateChange	StateChangeEvent Handler	Fires when the state of the connection is changed.
InfoMessage	SqlInfoMessageEventHandler	Fires when the database returns a warning or information message.

You'll learn how to use some of these properties, methods, and events in the following sections.

**Team LiB**

[PREVIOUS](#)

[NEXT](#)

## Using a `SqlConnection` Object to Connect to a SQL Server Database

You create a `SqlConnection` object using the `SqlConnection()` constructor. This constructor is *overloaded*, meaning that there are multiple versions of the constructor that you can call. The `SqlConnection()` constructors are as follows:

```
SqlConnection()
SqlConnection(string connectionString)
```

where `connectionString` contains the details for the database connection. You'll learn the details of the `connectionString` in this section.

Assuming you've imported the `System.Data.SqlClient` namespace, you can create a new `SqlConnection` object using the following statement:

```
SqlConnection mySqlConnection = new SqlConnection();
```

You can then set the details for the database connection using the `ConnectionString` property of `mySqlConnection`. For example:

```
mySqlConnection.ConnectionString =
    "server=localhost;database=Northwind;uid=sa;pwd=sa";
```

where

**server** specifies the name of the computer on which SQL Server is running.

**database** specifies the name of the database.

**uid** specifies the name of the database user.

**pwd** specifies the password for the user.

**Warning** For security reasons, do not include the username password in your production code. Instead ask the user to enter their name and password or use integrated security, which you'll learn about shortly.

One thing you need to bear in mind is that you can set the `ConnectionString` property only when your `Connection` object is closed.

You can also pass a connection string directly to the `SqlConnection()` constructor. For example:

```
string connectionString =
    "server=localhost;database=Northwind;uid=sa;pwd=sa";
SqlConnection mySqlConnection =
    new SqlConnection(connectionString);
```

Or more simply:

```
SqlConnection mySqlConnection =
    new SqlConnection(
        "server=localhost;database=Northwind;uid=sa;pwd=sa"
    );
```

You can set an optional connection timeout, which specifies the number of seconds that the `Open()` method will wait for a connection to the database. You do this by specifying a `connection timeout` in your connection string. For example, the following string specifies a `connection timeout` of 10 seconds:

```
string connectionString =
    "server=localhost;database=Northwind;uid=sa;pwd=sa;" +
    "connection timeout=10";
```

**Note** The default connection timeout is 15 seconds. A connection timeout of 0 seconds means the connection attempt will wait indefinitely. Avoid setting your connection timeout to 0.

Before starting a Windows session, you typically log in to Windows with a username and password. If you're using Windows integrated security, you can pass your username and password to SQL Server and use those credentials to connect to the database. This saves you from providing a separate username and password to SQL Server. You can use integrated security in your program by specifying `integrated security=SSPI` in your connection string. For example:

```
string connectionString =
    "server=localhost;database=Northwind;integrated security=SSPI";
```

Notice that you don't provide the username and password. Instead, the username and password you used when logging into Windows is passed to SQL Server. SQL Server will then check its list of users to see if you have permission to access the database. (For further details on integrated security, consult the SQL Server Books Online documentation.)

You've now seen how to create a `Connection` object using program statements. You'll see how to create a `Connection` object visually using Visual Studio .NET later in the "[Creating a Connection Object using Visual Studio .NET](#)" section. Next, you'll see how to open and close a connection.

## Opening and Closing a Database Connection

Once you've created your `Connection` object and set its `ConnectionString` property to the appropriate details for your database connection, you can open the connection to the database. You do this by calling the `Open()` method of your `Connection` object. The following example calls the `Open()` method of `mySqlConnection`:

```
mySqlConnection.Open();
```

Once you've finished with your database connection, you call the `Close()` method of your `Connection` object. For example:

```
mySqlConnection.Close();
```

[Listing 7.1](#) illustrates how to connect to the SQL Server Northwind database using a `SqlConnection` object and display some of the properties of that object.

### **Listing 7.1: MYSQLCONNECTION.CS**

---

```
/*
  MySqlConnection.cs illustrates how to use a
  SqlConnection object to connect to a SQL Server database
*/

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

class MySqlConnection
{
    public static void Main()
    {
        // formulate a string containing the details of the
        // database connection
        string connectionString =
            "server=localhost;database=Northwind;uid=sa;pwd=sa";

        // create a SqlConnection object to connect to the
        // database, passing the connection string to the constructor
        SqlConnection mySqlConnection =
            new SqlConnection(connectionString);
```

```

// open the database connection using the
// Open() method of the SqlConnection object
mySqlConnection.Open();

// display the properties of the SqlConnection object
Console.WriteLine("mySqlConnection.ConnectionString = "+ 
    mySqlConnection.ConnectionString);
Console.WriteLine("mySqlConnection.ConnectionTimeout = "+ 
    mySqlConnection.ConnectionTimeout);
Console.WriteLine("mySqlConnection.Database = "+ 
    mySqlConnection.Database);
Console.WriteLine("mySqlConnection.DataSource = "+ 
    mySqlConnection.DataSource);
Console.WriteLine("mySqlConnection.PacketSize = "+ 
    mySqlConnection.PacketSize);
Console.WriteLine("mySqlConnection.ServerVersion = "+ 
    mySqlConnection.ServerVersion);
Console.WriteLine("mySqlConnection.State = "+ 
    mySqlConnection.State);
Console.WriteLine("mySqlConnection.WorkstationId = "+ 
    mySqlConnection.WorkstationId);

// close the database connection using the Close() method
// of the SqlConnection object
mySqlConnection.Close();
}
}

```

---

The output from this program is as follows:

```

mySqlConnection.ConnectionString = server=localhost;database=Northwind;uid=sa;
mySqlConnection.ConnectionTimeout = 15
mySqlConnection.Database = Northwind
mySqlConnection.DataSource = localhost
mySqlConnection.PacketSize = 8192
mySqlConnection.ServerVersion = 08.00.0194
mySqlConnection.State = Open
mySqlConnection.WorkstationId = JMPRICE-DT1

```

**Note** Your results will differ from those here. For example, your connection string and workstation ID will be different.

## Connection Pooling

Opening and closing a database connection is a relatively time-consuming process. For this reason, ADO.NET automatically stores database connections in a pool. **Connection pooling** offers a great performance improvement because you don't have to wait for a brand new connection to the database to be established when there's a suitable connection already available. When you close a connection, that connection isn't actually closed; instead, your connection is marked as unused and stored in the pool, ready to be used again.

If you then supply the same details in the connection string (same database, username, password, and so on), then the connection from the pool is retrieved and returned to you. You then use that same connection to access the database.

When using a SqlConnection object, you can indicate the maximum number of connections allowed in the pool by specifying `max pool size` in your connection string (the default is 100). You can also indicate the minimum number of connections in the pool by specifying `min pool size` (the default is 0). For example, the following SqlConnection specifies a `max pool size` of 10 and a `min pool size` of 5:

```

SqlConnection mySqlConnection =
    new SqlConnection(

```

```
"server=localhost;database=Northwind;uid=sa;pwd=sa;" +
"max pool size=10;min pool size=5"
);
```

In this example, a pool with five initial `SqlConnection` objects is created. A maximum of 10 `SqlConnection` objects can be stored in the pool. If you attempt to open a new `SqlConnection` object and the pool is already full with currently used objects, your request waits until a `SqlConnection` object is closed, at which point that object is returned for you to use. If your request waits longer than the number of seconds in the `ConnectionTimeout` property, then an exception is thrown.

[Listing 7.2](#) illustrates the time-saving when opening a previously pooled connection.

## **Listing 7.2: CONNECTIONPOOLING.CS**

---

```
/*
 ConnectionPooling.cs illustrates connection pooling
 */

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

class ConnectionPooling
{
    public static void Main()
    {
        // create a SqlConnection object to connect to the database,
        // setting max pool size to 10 and min pool size to 5
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa;" +
                "max pool size=10;min pool size=5"
            );

        // open the SqlConnection object 10 times
        for (int count = 1; count <= 10; count++)
        {
            Console.WriteLine("count = " + count);

            // create a DateTime object and set it to the
            // current date and time
            DateTime start = DateTime.Now;

            // open the database connection using the
            // Open() method of the SqlConnection object
            mySqlConnection.Open();

            // subtract the current date and time from the start,
            // storing the difference in a TimeSpan
            TimeSpan timeTaken = DateTime.Now - start;

            // display the number of milliseconds taken to open
            // the connection
            Console.WriteLine("Milliseconds = " + timeTaken.Milliseconds);

            // display the connection state
            Console.WriteLine("mySqlConnection.State = " +
                mySqlConnection.State);

            // close the database connection using the Close() method
        }
    }
}
```

```

    // of the SqlConnection object
    mySqlConnection.Close();
}
}
}

```

---

The output from this program is as follows:

```

count = 1
Milliseconds = 101
mySqlConnection.State = Open
count = 2
Milliseconds = 0
mySqlConnection.State = Open
count = 3
Milliseconds = 0
mySqlConnection.State = Open
count = 4
Milliseconds = 0
mySqlConnection.State = Open
count = 5
Milliseconds = 0
mySqlConnection.State = Open
count = 6
Milliseconds = 0
mySqlConnection.State = Open
count = 7
Milliseconds = 0
mySqlConnection.State = Open
count = 8
Milliseconds = 0
mySqlConnection.State = Open
count = 9
Milliseconds = 0
mySqlConnection.State = Open
count = 10
Milliseconds = 0
mySqlConnection.State = Open

```

**Note** Your results might differ from those here.

As you can see, the time to open the first connection is relatively long compared with the subsequent ones. This is because the first connection makes the actual connection to the database. When it is closed, it's stored in the connection pool. When the connection is then opened again, it's retrieved from the pool, and this retrieval is very fast.

## Getting the State of a *Connection* Object

The state of a connection enables you to know the progress of your connection request to the database; two examples of states are open and closed. You use the `Connection` object's `State` property to get the current state of the connection to the database. The `State` property returns a constant from the `ConnectionState` enumeration.

**Note** An enumeration is a list of numeric constants, each of which has a name.

[Table 7.4](#) lists the constants defined in the `ConnectionState` enumeration.

### Table 7.4: `ConnectionState` CONSTANTS

CONSTANT NAME	DESCRIPTION
Broken	The Connection is broken. This can happen after you've opened the Connection object. You can close the Connection and reopen it.
Closed	The Connection is closed.
Connecting	The Connection is establishing access to the database.
Executing	The Connection is running a command.
Fetching	The Connection is retrieving information from the database.
Open	The Connection is open.

**Note** In version 1 of ADO.NET, only the *Open* and *Closed* states are used. The other states will be used in later versions.

An example of using the `State` property would be to check if your `Connection` object is currently open before calling its `Open()` method. You might need to do that if you have a complex application and you're using a `Connection` object created somewhere else in the application: you might not know the current state of that `Connection` object and you don't want to call the `Open()` method on an already open `Connection` because that will raise an exception.

The following example uses the `State` property to check if `mySqlConnection` is closed before opening it:

```
if (mySqlConnection.State == ConnectionState.Closed)
{
    mySqlConnection.Open();
}
```

As you'll learn in the [next section](#), you can use the `StateChange` event to monitor changes in a `Connection` object's state.

## Using *Connection* Events

The `Connection` classes have two useful events: `StateChange` and `InfoMessage`. You'll see how to use these events next.

### The `StateChange` Event

The `StateChange` event fires when the state of your connection is changed, and you can use this event to monitor changes in the state of your `Connection` object.

The method that handles an event is known as an *event handler*. You call this method when a particular event is fired. All event handler methods must return `void` and accept two parameters. The first parameter is an `object` (of the class `System.Object`), and it represents the object that raises the event.

**Note** The `System.Object` class acts as the base class for all classes. In other words, all classes are ultimately derived from the `System.Object` class.

The second parameter is an object of a class that is derived from the `System.EventArgs` class. The `EventArgs` class is the base class for event data and represents the details of the event. In the case of the `StateChange` event, this second object is of the `StateChangeEventArgs` class.

The following example defines a method named `StateChangeHandler` to handle the `StateChange` event. You'll notice that the second parameter to this method is a `StateChangeEventArgs` object. You get the original state of the connection using this object's `OriginalState` property, and you get the current state using the `CurrentState` property.

```
public static void StateChangeHandler(
    object mySender, StateChangeEventArgs myEvent
)
```

```
{
    Console.WriteLine(
        "mySqlConnection State has changed from " +
        myEvent.OriginalState + "to " +
        myEvent.CurrentState
    );
}
```

To monitor an event, you must register your event handler method with that event. For example, the following statement registers the `StateChangeHandler()` method with the `StateChange` event of the `mySqlConnection` object:

```
mySqlConnection.StateChange +=  
    new StateChangeEventHandler(StateChangeHandler);
```

Whenever the `StateChange` event fires, the `StateChangeHandler()` method will be called, which displays the original and current state of `mySqlConnection`.

[Listing 7.3](#) illustrates the use of the `StateChange` event.

### **Listing 7.3: STATECHANGE.CS**

---

```
/*  
 * StateChange.cs illustrates how to use the StateChange event  
 */  
  
using System;  
using System.Data;  
using System.Data.SqlClient;  
  
class StateChange  
{  
    // define the StateChangeHandler() method to handle the  
    // StateChange event  
    public static void StateChangeHandler(  
        object mySender, StateChangeEventArgs myEvent  
    )  
    {  
        Console.WriteLine(  
            "mySqlConnection State has changed from " +  
            myEvent.OriginalState + "to " +  
            myEvent.CurrentState  
        );  
    }  
  
    public static void Main()  
    {  
        // create a SqlConnection object  
        SqlConnection mySqlConnection =  
            new SqlConnection("server=localhost;database=Northwind;uid=sa;pwd=sa");  
  
        // monitor the StateChange event using the StateChangeHandler() method  
        mySqlConnection.StateChange +=  
            new StateChangeEventHandler(StateChangeHandler);  
  
        // open mySqlConnection, causing the State to change from Closed  
        // to Open  
        Console.WriteLine("Calling mySqlConnection.Open()");  
        mySqlConnection.Open();  
  
        // close mySqlConnection, causing the State to change from Open  
        // to Closed
```

```

        Console.WriteLine("Calling mySqlConnection.Close()");
        mySqlConnection.Close();
    }
}

```

---

The output from this program is as follows:

```

Calling mySqlConnection.Open()
mySqlConnection State has changed from Closed to Open
Calling mySqlConnection.Close()
mySqlConnection State has changed from Open to Closed

```

## The *InfoMessage* Event

The *InfoMessage* event fires when the database returns a warning or information message produced by the database. You use the *InfoMessage* event to monitor these messages. To get the message, you read the contents of the *Errors* collection from the *SqlInfoMessageEventArgs* object.

You can produce information and error messages using the SQL Server `PRINT` or `RAISERROR` statements, which are described in [Chapter 4](#), "Introduction to Transact-SQL Programming."

The following *InfoMessageHandler()* method is used to handle the *InfoMessage* event. Notice the use of the *Errors* collection to display the message:

```

public static void InfoMessageHandler(
    object mySender, SqlInfoMessageEventArgs myEvent
)
{
    Console.WriteLine(
        "The following message was produced:\n" +
        myEvent.Errors[0]
);
}

```

**Note** If you're using the OLE DB managed providers, you replace *SqlInfoMessageEventArgs* with *OleDbInfoMessageEventArgs*. If you're using the ODBC managed providers, you replace *SqlInfoMessageEventArgs* with *OdbcInfoMessageEventArgs*.

[Listing 7.4](#) illustrates the use of the *InfoMessage* event. You'll notice this program uses the *ExecuteNonQuery()* method of the *SqlCommand* object to send `PRINT` and `RAISERROR` statements to the database for execution. You'll learn the details of the *SqlCommand* object and the *ExecuteNonQuery()* method in [Chapter 8](#), "Executing Database Commands."

### **Listing 7.4: INFOMESSAGE.CS**

---

```

/*
    InfoMessage.cs illustrates how to use the InfoMessage event
*/

using System;
using System.Data;
using System.Data.SqlClient;
class InfoMessage
{
    // define the InfoMessageHandler() method to handle the
    // InfoMessage event
    public static void InfoMessageHandler(
        object mySender, SqlInfoMessageEventArgs myEvent
    )

```

```

)
{
    Console.WriteLine(
        "The following message was produced:\n" +
        myEvent.Errors[0]
    );
}

public static void Main()
{
    // create a SqlConnection object
    SqlConnection mySqlConnection =
        new SqlConnection("server=localhost;database=Northwind;uid=sa;pwd=sa");

    // monitor the InfoMessage event using the InfoMessageHandler() method
    mySqlConnection.InfoMessage +=
        new SqlInfoMessageEventHandler(InfoMessageHandler);

    // open mySqlConnection
    mySqlConnection.Open();

    // create a SqlCommand object
    SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

    // run a PRINT statement
    mySqlCommand.CommandText =
        "PRINT 'This is the message from the PRINT statement'";
    mySqlCommand.ExecuteNonQuery();

    // run a RAISERROR statement
    mySqlCommand.CommandText =
        "RAISERROR('This is the message from the RAISERROR statement', 10, 1)";
    mySqlCommand.ExecuteNonQuery();

    // close mySqlConnection
    mySqlConnection.Close();
}
}

```

---

The output from this program is as follows:

The following message was produced:

System.Data.SqlClient.SqlError: This is the message from the PRINT statement

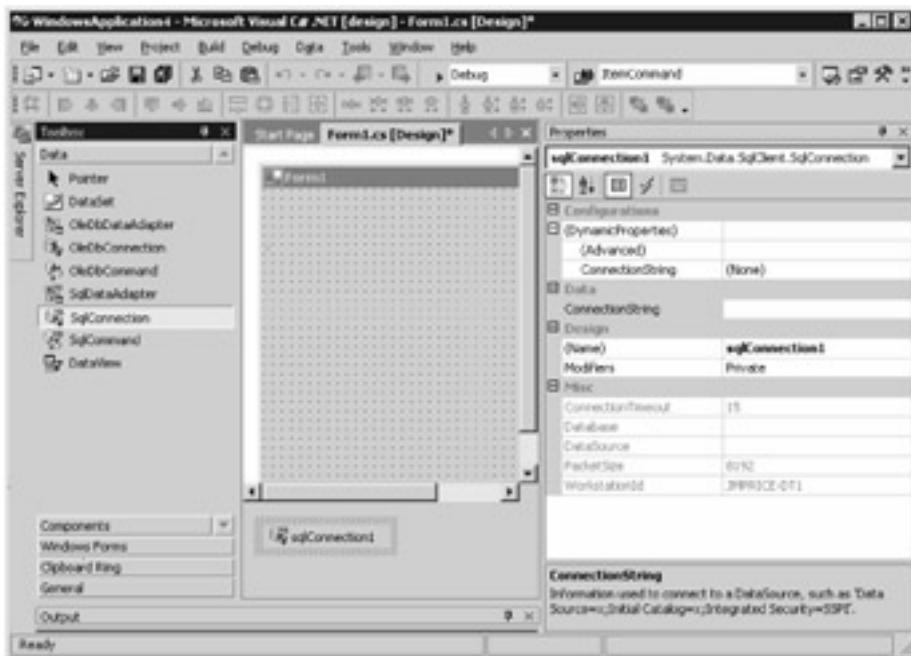
The following message was produced:

System.Data.SqlClient.SqlError: This is the message from the RAISERROR statement

## Creating a *Connection* Object Using Visual Studio .NET

To create a `SqlConnection` object using Visual Studio .NET, you drag a `SqlConnection` object from the Data tab of the Toolbox to your form. You'll recall that a `SqlConnection` object allows you to connect to a SQL Server database. You can also drag an `OleDbConnection` object from the Toolbox to your form to connect to a database that supports OLE DB.

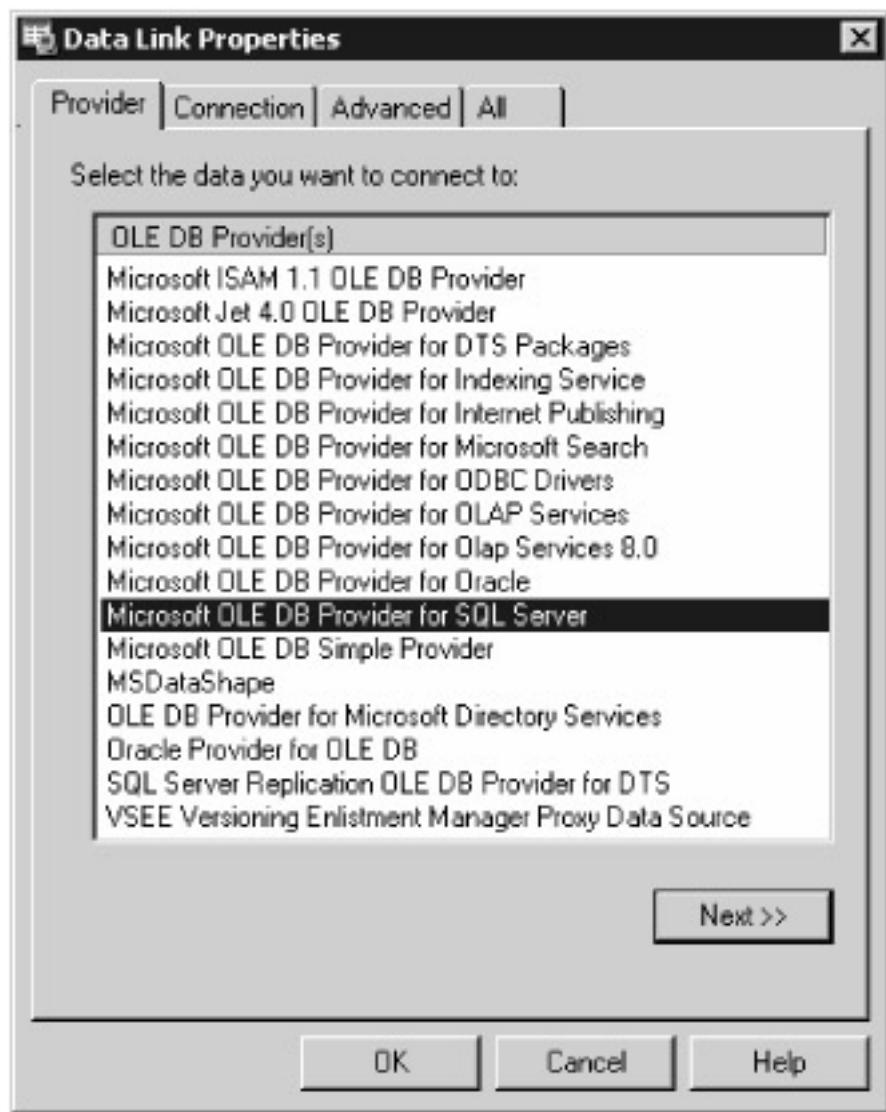
[Figure 7.1](#) shows a form with a `SqlConnection` object. This object is assigned the default name of `sqlConnection1`.



**Figure 7.1:** Creating a `SqlConnection` object with Visual Studio .NET

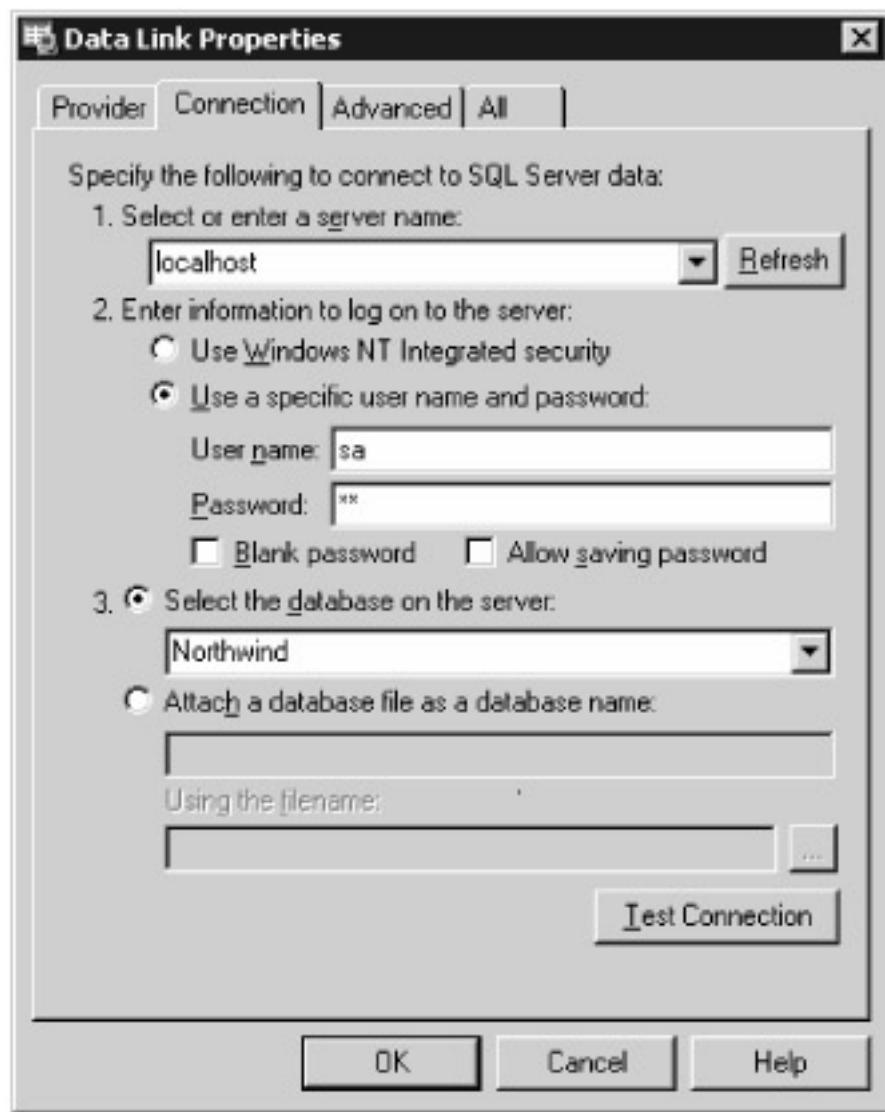
Once you've created a `SqlConnection` object, that object appears in the "tray" below the form. The tray is used to store nonvisual components like `SqlConnection` objects. Other objects that appear in the tray are `SqlCommand` objects. These objects are considered nonvisual because you don't see them when you run your form. You can of course still work with them visually when designing your form.

To the right of the form, you'll notice the Properties window, which you use to set the properties for your `SqlConnection` object. To set the `ConnectionString` property that specifies the details of the database connection, you can either type in the string directly or click the drop-down list and build the `ConnectionString` visually. To do this, you select `New Connection` from the drop-down list, which displays the Data Link Properties dialog box. This dialog box contains four tabs, the first of which is the Provider tab, which allows you to select the type of provider you want to connect to, as shown in [Figure 7.2](#).



**Figure 7.2:** Selecting the provider

Click the Next button to continue to the Connection tab (you can also click the Connection tab directly), where you enter the details for your database connection, as shown in [Figure 7.3](#).



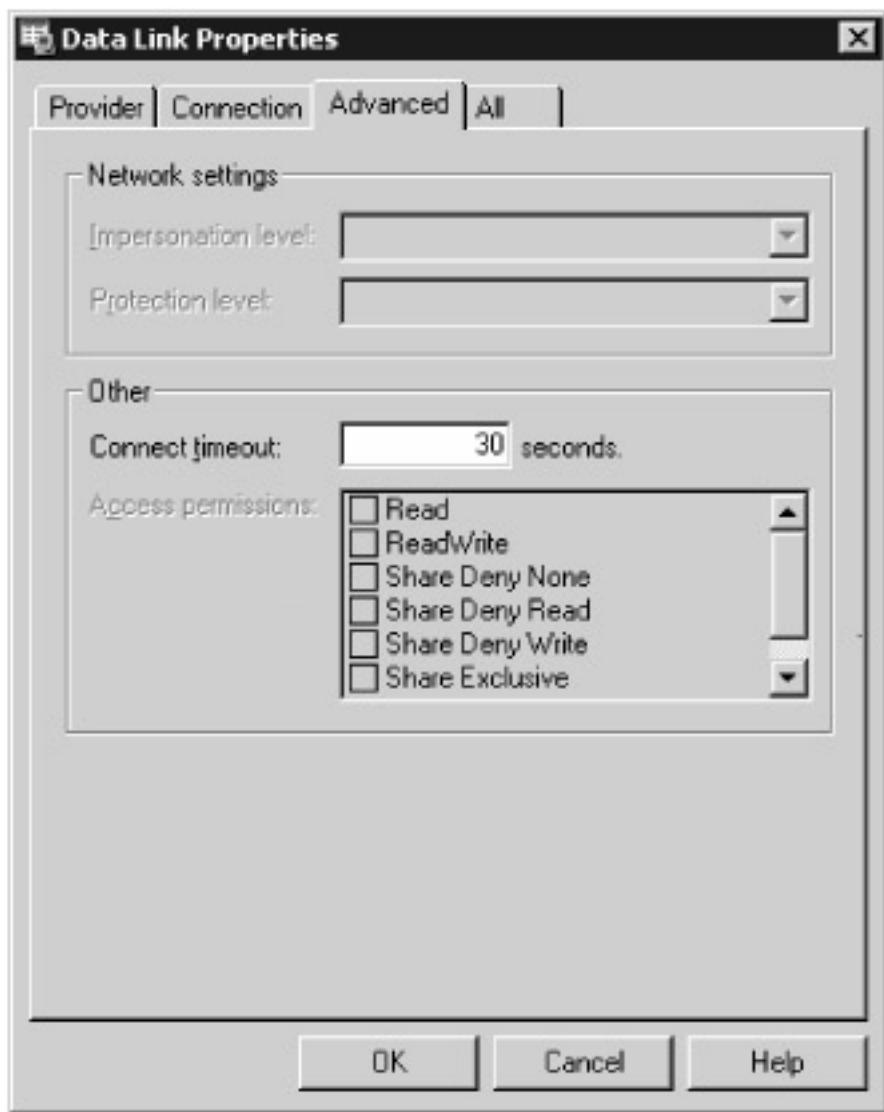
**Figure 7.3:** Entering the connection details

**Warning**

For security reasons, do not enable the Allow Saving Password check box. If you did, your password would be stored in the actual code, and anyone could get your password from the code. Leave Allow Saving Password in its default non-enabled state; that way, the user will be prompted to enter the password. For testing purposes, however, leaving your password in is sometimes acceptable—just remember not to release your password in production code.

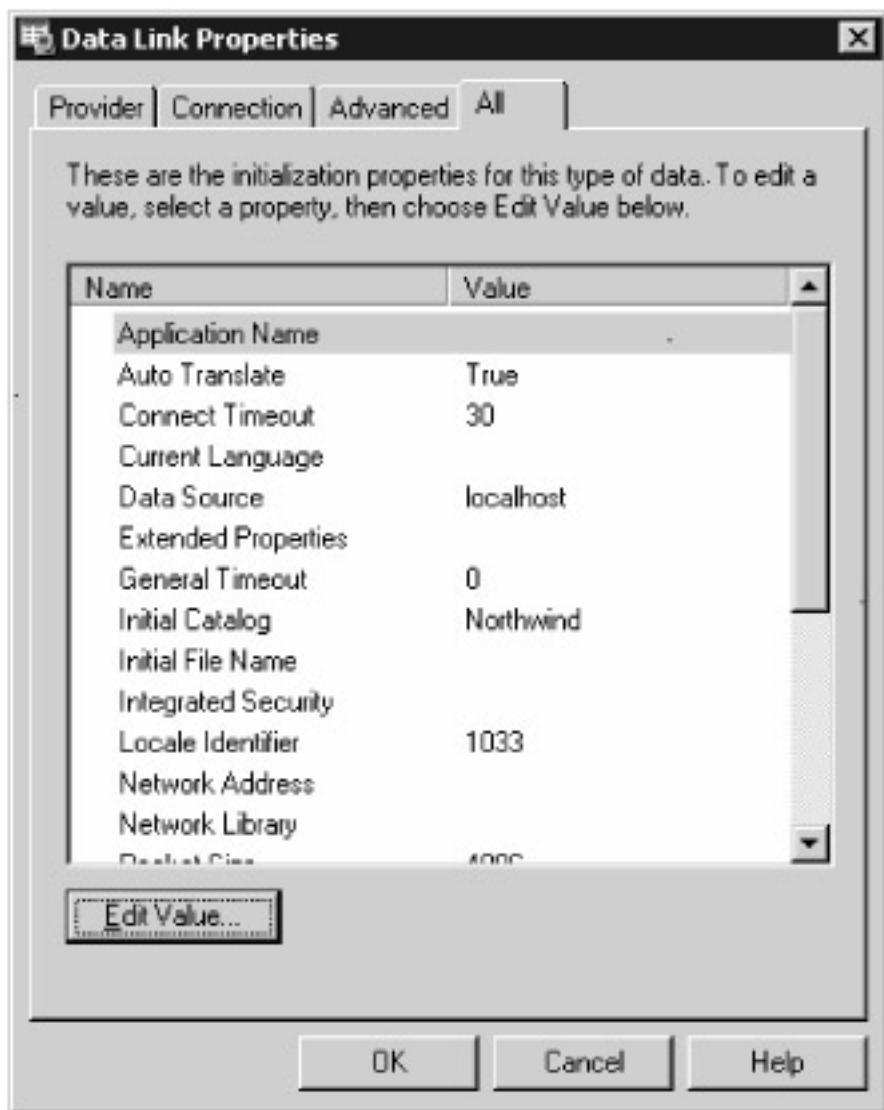
Once you've entered your connection details, you can press the Test Connection button to ensure your details are correct.

At this point, you've entered all the mandatory details, and you can choose to save your details by clicking OK, or you can click Advanced to enter additional details such as the connection timeout, as shown in [Figure 7.4](#).



**Figure 7.4:** Entering the advanced connection details

You can also click the All tab to view and edit all the values for the connection, as shown in [Figure 7.5](#). To edit a value, you click Edit Value.



**Figure 7.5:** Viewing all the connection details

Click the OK button to save your connection details.

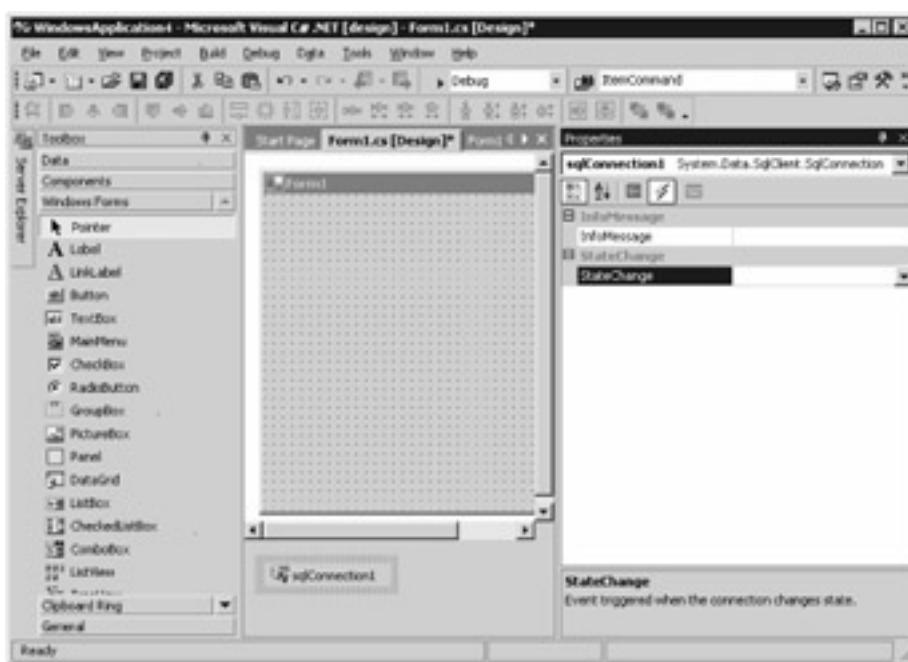
On my computer, the `ConnectionString` property for my `SqlConnection` object that connects to the SQL Server Northwind database is set to

```
data source=localhost;initial catalog=Northwind;persist security info=False;
user id=sa;pwd=sa;workstation id=JMPRICE-DT1;packet size=4096
```

**Note** The `persist security info` Boolean value controls whether security-sensitive information such as your password is returned in a connection that has been previously opened. You'll typically want to leave this in the default setting of `False`.

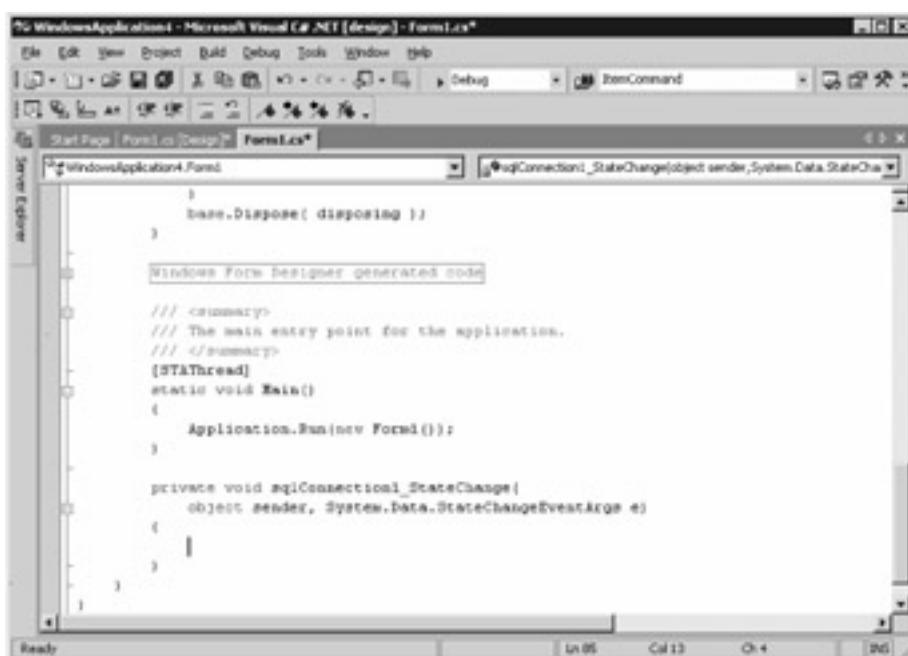
## Coding an Event in VS .NET

You can add code for an event in VS .NET. For example, let's say you wanted to add code for the `State-Change` event of the `sqlConnection1` object created earlier. To do this, you first select `sqlConnection1` in the tray, then you click the Events (lighting) button in the Properties window. [Figure 7.6](#) shows the `sqlConnection1` object's events.



**Figure 7.6:** sqlConnection1 object's events

You then double-click the name of the event in the Properties window you want to code. In this example, you double-click the StateChange event. VS .NET then displays the code and creates a skeleton of the event handler method for you, as shown in [Figure 7.7](#). The cursor shows where you add your code.

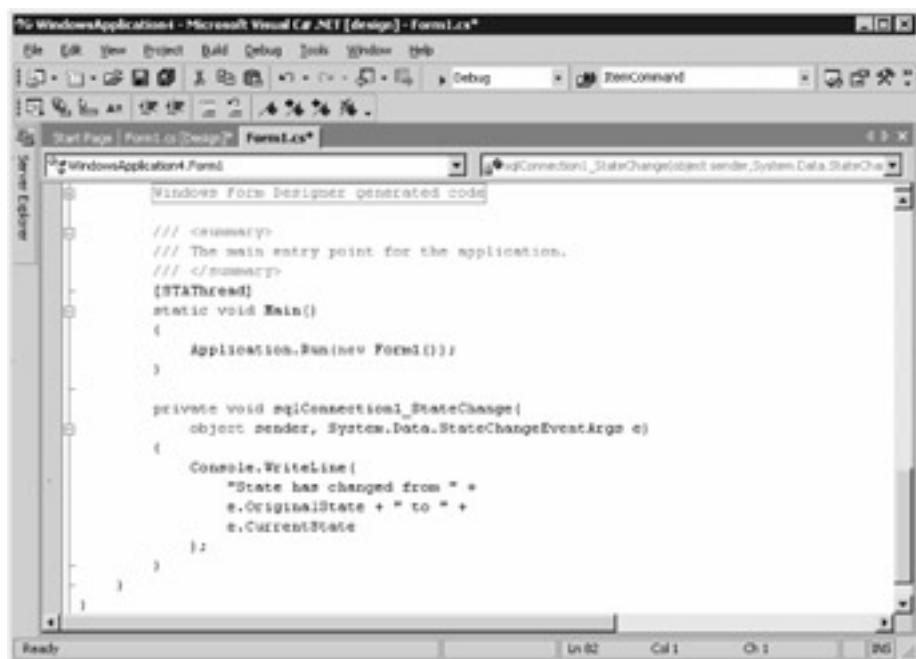


**Figure 7.7:** The beginning StateChange event handler method

All you have to do is add your code to the event handler method. For example, you can set your method to

```
private void sqlConnection1_StateChange(
    object sender, System.Data.StateChangeEventArgs e)
{
    Console.WriteLine(
        "State has changed from " +
        e.OriginalState + "to " +
        e.CurrentState
    );
}
```

[Figure 7.8](#) shows the completed event handler method.



The screenshot shows the Microsoft Visual Studio .NET IDE in design mode for a Windows application. The title bar reads "WindowsApplication4 - Microsoft Visual Studio .NET [design] - Form1.cs". The menu bar includes File, Edit, View, Project, Build, Debug, Tools, Window, Help. The toolbar has icons for New, Open, Save, Print, and others. The status bar at the bottom shows "Ready". The code editor window contains the following C# code:

```
Windows Form Designer generated code

/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
    Application.Run(new Form1());
}

private void sqlConnection1_StateChange(
    object sender, System.Data.StateChangeEventArgs e)
{
    Console.WriteLine(
        "State has changed from " +
        e.OriginalState + " to " +
        e.CurrentState
    );
}
```

**Figure 7.8:** The completed StateChange event handler method

Once you've created a `SqlConnection` object, you can then use it with other ADO.NET objects, such as a `SqlCommand` object. You'll see how to do that with VS .NET in [Chapter 8](#).

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

## Summary

In this chapter, you learned how to connect to a database. There are three `Connection` classes: `SqlConnection`, `OleDbConnection`, and `OdbcConnection`. You use an object of the `SqlConnection` class to connect to a SQL Server database, an object of the `OleDbConnection` class to connect to any database that supports OLE DB, such as Oracle or Access, and an object of the `OdbcConnection` class to connect to any database that supports ODBC. Ultimately, all communication with a database is done through a `Connection` object.

ADO.NET automatically stores database connections in a pool. When you close a connection, that connection isn't actually closed. Instead, your connection is marked as unused and stored in the pool ready to be used again. If you then supply the same details in the connection string (same database, username, password, and so on), then the connection from the pool is retrieved and returned to you. You then use that same connection to access the database. Connection pooling offers a great performance improvement, because you don't have to wait for a new connection to the database to be established when there's a suitable connection already available in the pool.

You use a `Connection` object's `State` property to get the current state of the connection to the database. The `State` property returns a constant from the `ConnectionState` enumeration.

You use a `Connection` object's `StateChange` event to monitor changes in the state of your `Connection` object. You use a `Connection` object's `InfoMessage` event to monitor warning or information messages produced by the database. You can produce such messages using the SQL Server `PRINT` or `RAISERROR` statements.

To create a `SqlConnection` object using Visual Studio .NET, you drag a `SqlConnection` object from the Data tab of the Toolbox to your form. You can also drag an `OleDbConnection` object from the Toolbox to your form.

In the [next chapter](#), you'll learn how to execute database commands.

## Chapter 8: Executing Database Commands

### Overview

Database commands are executed through `Command` objects, and are part of the managed providers. There are three `Command` classes: `SqlCommand`, `OleDbCommand`, and `OdbcCommand`. You use a `Command` object to execute a SQL `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. You can also use a `Command` object to call a stored procedure, or retrieve all the rows and columns from a specific table; this is known as a `TableDirect` command. A `Command` object communicates with the database using a `Connection` object, which was described in [Chapter 7](#), "Connecting to a Database."

Featured in this chapter:

- The `SqlCommand` class
- Using a `SqlCommand` object to execute commands against a SQL Server Database
- Executing `SELECT` statements and `TableDirect` commands
- Executing a command that retrieves data as XML
- Executing commands that modify information in the database
- Introducing transactions
- [Supplying parameters to commands](#)
- [Executing SQL Server stored procedures](#)
- Creating a `Command` object using Visual Studio .NET

## The *SqlCommand* Class

You use an object of the *SqlCommand* class to execute a command against a SQL Server database, an object of the  *OleDbCommand* class to execute a command against any database that supports OLE DB, such as Oracle or Access, and an object of the *OdbcCommand* class to execute a command against any database that supports ODBC. [Table 8.1](#) shows some of the *SqlCommand* properties. [Table 8.2](#) shows some of the *SqlCommand* methods. You'll learn how to use some of these properties and methods in this chapter.

**Table 8.1:** *sqlCommand* PROPERTIES

PROPERTY	TYPE	DESCRIPTION
CommandText	string	Gets or sets the SQL statement, stored procedure call, or table to retrieve from.
CommandTimeout	int	Gets or sets the number of seconds to wait before ending an attempt to execute the command. The default is 30 seconds.
CommandType	CommandType	Gets or sets a value that indicates how the <i>CommandText</i> property is to be interpreted. Valid values are <i> CommandType.Text</i> , <i> CommandType.StoredProcedure</i> , and <i> CommandType.TableDirect</i> . <i>Text</i> indicates the command is a SQL statement. <i>StoredProcedure</i> indicates the command is a stored procedure call. <i>TableDirect</i> indicates the name of a table, for which all rows and columns are to be retrieved. The default is <i>Text</i> .
Connection	string	Gets the name of the database connection.
DesignTimeVisible	bool	Gets or sets a Boolean value that indicates whether the <i> Command</i> object is visible in a Windows Forms Designer control. The default is <i>false</i> .
Parameters	SqlParameterCollection	Gets the parameters (if any) to supply to the command. When using a <i> SqlConnection</i> , the parameters are stored in a <i> SqlParameterCollection</i> object.
Transaction	SqlTransaction	Gets or sets the database transaction for the command.
UpdatedRowSource	UpdateRowSource	Gets or sets how the command results are to be applied to a <i> DataRow</i> object when the <i> Update</i> ( ) method of a <i> DataAdapter</i> object is called.

**Table 8.2:** *sqlCommand* METHODS

METHOD	RETURN TYPE	DESCRIPTION
Cancel()	void	Cancels the execution of the command.
CreateParameter()	SqlParameter	Creates a new parameter for the command.

ExecuteNonQuery()	int	Used to execute SQL statements that don't return a result set. These statements include <code>INSERT</code> , <code>UPDATE</code> , and <code>DELETE</code> statements, Data Definition Language statements, or stored procedure calls that don't return a result set. The <code>int</code> value returned is the number of database rows affected by the command, if any.
ExecuteReader()	SqlDataReader	Used to execute SQL <code>SELECT</code> statements, <code>TableDirect</code> commands, or stored procedures that return a result set. Returns the result set in a <code>DataReader</code> object.
ExecuteScalar()	object	Used to execute SQL <code>SELECT</code> statements that return a single value (any other values are ignored). Returns the result of the command as an <code>object</code> .
ExecuteXmlReader()	XmlReader	Used to execute SQL <code>SELECT</code> statements that return XML data. Returns the result set in an <code>xmlReader</code> object. Applies only to the <code>SqlCommand</code> class.
Prepare()	void	Creates a prepared version of the command. Sometimes results in faster execution of the command.
ResetCommandTimeout()	void	Resets the <code>CommandTimeout</code> property to its default value.

**Note** Although the `SqlCommand` class is specific to SQL Server, many of the properties and methods in this class are the same as those for the `OleDbCommand` and `OdbcCommand` classes. If a property or method is specific to `SqlCommand`, it says so in the Description column of the tables shown in this section.

**Tip** You're actually better off using the T-SQL `EXECUTE` command rather than `CommandType.StoredProcedure` to execute a stored procedure. This is because you can read values that are returned from a stored procedure through a `RETURN` statement, which you can't do when setting the `CommandType` to `StoredProcedure`. See the section "[Executing SQL Server Stored Procedures](#)" later in this chapter.

**Team LiB**
[◀ PREVIOUS](#) [NEXT ▶](#)

## Creating a *SqlCommand* Object

There are two ways you can create a *SqlCommand* object:

- Use one of the *SqlCommand* constructors.
- Call the *CreateCommand( )* method of a *SqlConnection* object.

You'll see how to use both these ways to create *SqlCommand* objects next.

**Note** You can use the same ways shown in the following sections to create an *OleDbCommand* or *OdbcCommand* object.

### Creating a *SqlCommand* Object Using a Constructor

The *SqlCommand* constructors are as follows:

```
SqlCommand()
SqlCommand(string commandText)
SqlCommand(string commandText, SqlConnection mySqlConnection)
SqlCommand(string commandText, SqlConnection mySqlConnection, SqlTransaction
mySqlTransaction)
```

where

**commandText** contains your SQL statement, stored procedure call, or table to retrieve from.

**mySqlConnection** is your *SqlConnection* object.

**mySqlTransaction** is your *SqlTransaction* object.

Before you use a *SqlCommand* object you first need a *SqlConnection* object, which is used to communicate with a SQL Server database:

```
mySqlConnection.ConnectionString =
"server=localhost;database=Northwind;uid=sa;pwd=sa";
```

Next, you can create a new *SqlCommand* object using the following statement:

```
SqlCommand mySqlCommand = new SqlCommand();
```

You then set the *Connection* property of *mySqlCommand* to *mySqlConnection*:

```
mySqlCommand.Connection = mySqlConnection;
```

The *mySqlCommand* object will then use *mySqlConnection* to communicate with the database.

Now, the  *CommandType* property of a *Connection* object determines the type of command to be executed. You can use any of the values in the *System.Data.CommandType* enumeration to specify the  *CommandType* property. [Table 8.3](#) shows the  *CommandType* enumeration values.

**Table 8.3: CommandType ENUMERATION VALUES**

VALUE	DESCRIPTION
Text	Indicates the command is a SQL statement. <i>Text</i> is the default.
StoredProcedure	Indicates the command is a stored procedure call.

**TableDirect**

Indicates the name of a table, for which all rows and columns are to be retrieved. Note: SqlCommand objects don't support TableDirect. You have to use an object of one of the other Command classes instead.

You'll see how to use all three of these command types in this chapter. For now, I'll focus on the default `Text` command type, which indicates the command is a SQL statement.

You set the command to be executed using the `CommandText` property of your `Command` object. The following example sets the `CommandText` property of `mySqlCommand` to a `SELECT` statement:

```
mySqlCommand.CommandText =
    "SELECT TOP 10 CustomerID, CompanyName, ContactName, Address " +
    "FROM Customers " +
    "ORDER BY CustomerID";
```

You can also pass the command and the `Connection` object to the constructor in one step when creating a `Command` object. For example:

```
SqlCommand mySqlCommand = new SqlCommand(
    "SELECT TOP 5 CustomerID, CompanyName, ContactName, Address " +
    "FROM Customers " +
    "ORDER BY CustomerID",
    mySqlConnection);
```

In the [next section](#), you'll learn how to create a `SqlCommand` object using the `CreateCommand()` method of a `SqlConnection` object.

## **Creating a *SqlCommand* Object Using the *CreateCommand()* Method**

Rather than creating a `SqlCommand` object using the constructors, you can use the `CreateCommand()` method of a `SqlConnection` object. The `CreateCommand()` method returns a new `SqlCommand` object. For example:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
```

The `mySqlCommand` object will use `mySqlConnection` to communicate with the database.

## Executing ***SELECT*** Statements and ***TableDirect*** Commands

A TableDirect command is actually a **SELECT** statement that returns all the rows and columns for a specified table. A Command object has three methods you can use to execute a **SELECT** statement or TableDirect command. [Table 8.4](#) shows these methods, which you'll learn how to use in the following sections.

**Table 8.4: METHODS THAT RETRIEVE INFORMATION FROM THE DATABASE**

METHOD	RETURN TYPE	DESCRIPTION
ExecuteReader()	SqlDataReader	Used to execute SQL <b>SELECT</b> statements, TableDirect commands or stored procedure calls that return a result set. Returns the result set in a <b>DataReader</b> object.
ExecuteScalar()	object	Used to execute SQL <b>SELECT</b> statements that return a single value (any other values are ignored). Returns the single value as an <b>object</b> .
ExecuteXmlReader()	XmlReader	Used to execute SQL <b>SELECT</b> statements that return XML data. Returns the result set in an <b>XmlReader</b> object. Applies only to the <b>SqlCommand</b> class.

### Executing a ***SELECT*** Statement Using the ***ExecuteReader()*** Method

Let's take a look at an example that executes a **SELECT** statement using the **ExecuteReader()** method. This method returns the result set in a **DataReader** object, which you can then use to read the rows returned by the database. For example, the following code creates the required objects and executes a **SELECT** statement that retrieves the top five rows from the **Customers** table:

```
SqlConnection mySqlConnection =
    new SqlConnection(
        "server=localhost;database=Northwind;uid=sa;pwd=sa"
    );
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 5 CustomerID, CompanyName, ContactName, Address " +
    "FROM Customers " +
    "ORDER BY CustomerID";
mySqlConnection.Open();
SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();
```

**Tip** You'll notice that I didn't call the **Open()** method of the **SqlConnection** object until just before calling the **ExecuteReader()** method of the **SqlCommand** object. This is intentional. By opening the connection at the very last moment, you minimize time spent connected to the database and therefore conserve database resources.

The result set returned by **mySqlCommand** is stored in **mySqlDataReader**. You then read the rows from **mySqlDataReader** using the **Read()** method. This method returns the Boolean **true** value when there is another row to read, otherwise it returns **false**. You can read an individual column value in a row from **mySqlDataReader** by passing the name of the column in square brackets. For example, to read the **CustomerID** column, you use **mySqlDataReader["CustomerID"]**.

**Note** You can also specify the column you want to get by passing a numeric value in brackets. For example, **mySqlDataReader[0]** also returns the **CustomerID** column value. 0 corresponds to the first column in the table, which in this example is the **CustomerID** column.

You can use the **Read()** method in a **while** loop to read each row in turn, as shown in the following example:

```
while (mySqlDataReader.Read())
```

```
{
    Console.WriteLine("mySqlDataReader[\" CustomerID\"] = " +
        mySqlDataReader["CustomerID"]);
    Console.WriteLine("mySqlDataReader[\" CompanyName\"] = " +
        mySqlDataReader["CompanyName"]);
    Console.WriteLine("mySqlDataReader[\" ContactName\"] = " +
        mySqlDataReader["ContactName"]);
    Console.WriteLine("mySqlDataReader[\" Address\"] = " +
        mySqlDataReader["Address"]);
}
```

[Listing 8.1](#) illustrates a complete program that uses the code examples shown in this section.

### **Listing 8.1: EXECUTESELECT.CS**

---

```
/*
 ExecuteSelect.cs illustrates how to execute a SELECT
 statement using a SqlCommand object
 */

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

class ExecuteSelect
{
    public static void Main()
    {
        // create a SqlConnection object to connect to the database
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        // create a SqlCommand object
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        // set the CommandText property of the SqlCommand object to
        // the SELECT statement
        mySqlCommand.CommandText =
            "SELECT TOP 5 CustomerID, CompanyName, ContactName, Address " +
            "FROM Customers " +
            "ORDER BY CustomerID";

        // open the database connection using the
        // Open() method of the SqlConnection object
        mySqlConnection.Open();

        // create a SqlDataReader object and call the ExecuteReader()
        // method of the SqlCommand object to run the SQL SELECT statement
        SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();

        // read the rows from the SqlDataReader object using
        // the Read() method
        while (mySqlDataReader.Read())
        {
            Console.WriteLine("mySqlDataReader[\" CustomerID\"] = " +
                mySqlDataReader["CustomerID"]);
            Console.WriteLine("mySqlDataReader[\" CompanyName\"] = " +
```

```

        mySqlDataReader[ "CompanyName" ] ;
        Console.WriteLine( "mySqlDataReader[ \" ContactName\\" ] = " +
            mySqlDataReader[ "ContactName" ] );
        Console.WriteLine( "mySqlDataReader[ \" Address\\" ] = " +
            mySqlDataReader[ "Address" ] );
    }

    // close the SqlDataReader object using the Close() method
    mySqlDataReader.Close();

    // close the SqlConnection object using the Close() method
    mySqlConnection.Close();
}
}

```

---

The output from this program is as follows:

```

mySqlDataReader[ "CustomerID" ] = ALFKI
mySqlDataReader[ "CompanyName" ] = Alfs Futterkiste
mySqlDataReader[ "ContactName" ] = Maria Anders
mySqlDataReader[ "Address" ] = Obere Str. 57
mySqlDataReader[ "CustomerID" ] = ANATR
mySqlDataReader[ "CompanyName" ] = Ana Trujillo3 Emparedados y helados
mySqlDataReader[ "ContactName" ] = Ana Trujillo
mySqlDataReader[ "Address" ] = Avda. de la Constitución 2222
mySqlDataReader[ "CustomerID" ] = ANTON
mySqlDataReader[ "CompanyName" ] = Antonio Moreno Taquería
mySqlDataReader[ "ContactName" ] = Antonio Moreno
mySqlDataReader[ "Address" ] = Mataderos 2312
mySqlDataReader[ "CustomerID" ] = AROUT
mySqlDataReader[ "CompanyName" ] = Around the Horn
mySqlDataReader[ "ContactName" ] = Thomas Hardy
mySqlDataReader[ "Address" ] = 120 Hanover Sq.
mySqlDataReader[ "CustomerID" ] = BERGS
mySqlDataReader[ "CompanyName" ] = Berglunds snabbköp
mySqlDataReader[ "ContactName" ] = Christina Berglund
mySqlDataReader[ "Address" ] = Berguvsvägen 8

```

## Controlling the Command Behavior Using the *ExecuteReader()* Method

The `ExecuteReader()` method accepts an optional parameter that controls the command behavior. The values for this parameter come from the `System.Data.CommandBehavior` enumeration, for which values are shown in [Table 8.5](#).

**Table 8.5: CommandBehavior ENUMERATION VALUES**

VALUE	DESCRIPTION
CloseConnection	Specifies that when the associated <code>DataReader</code> object is closed, the <code>Connection</code> object is also closed.
Default	Indicates the <code>Command</code> object may return multiple result sets.
KeyInfo	Specifies the <code>Command</code> object returns information about the primary key columns in the result set.
SchemaOnly	Indicates the <code>Command</code> object returns information only about the columns.

SequentialAccess	Enables a <code>DataReader</code> object to read rows that have columns containing large binary values. <code>SequentialAccess</code> causes the <code>DataReader</code> to read the data as a stream. You then use the <code>GetBytes()</code> or <code>GetChars()</code> methods of the <code>DataReader</code> to read the stream. Note: you'll learn the details of <code>DataReader</code> objects in the <a href="#">next chapter</a> .
SingleResult	Specifies the <code>Command</code> object returns a single result set.
SingleRow	Indicates the <code>Command</code> object returns a single row.

You'll see how to use the `SingleRow` and `SchemaOnly` command behaviors next.

## Using the `SingleRow` Command Behavior

You use the `SingleRow` command behavior to indicate that your `Command` object returns a single row. For example, let's say you have a `Command` object named `mySqlCommand` with the `CommandText` property set as follows:

```
mySqlCommand.CommandText =
    "SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice " +
    "FROM Products";
```

Next, the following example passes the `CommandBehavior.SingleRow` value to the `ExecuteReader()` method, indicating that the `Command` object retrieves only the first row:

```
SqlDataReader mySqlDataReader =
    mySqlCommand.ExecuteReader(CommandBehavior.SingleRow);
```

Even though the earlier `SELECT` statement indicates that *all* the rows are to be retrieved from the `Products` table, the `mySqlDataReader` object will be able to read only the first row.

[Listing 8.2](#) illustrates the effect of using `CommandBehavior.SingleRow`.

### Listing 8.2: SINGLEROWCOMMANDBEHAVIOR.CS

```
/*
 SingleRowCommandBehavior.cs illustrates how to control
 the command behavior to return a single row
*/
using System;
using System.Data;
using System.Data.SqlClient;

class SingleRowCommandBehavior
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice " +
            "FROM Products";

        mySqlConnection.Open();

        // pass the CommandBehavior.SingleRow value to the
        // ExecuteReader() method, indicating that the Command object
        // only returns a single row
```

```

SqlDataReader mySqlDataReader =
    mySqlCommand.ExecuteReader(CommandBehavior.SingleRow);

while (mySqlDataReader.Read())
{
    Console.WriteLine("mySqlDataReader[\" ProductID\"] = " +
        mySqlDataReader["ProductID"]);
    Console.WriteLine("mySqlDataReader[\" ProductName\"] = " +
        mySqlDataReader["ProductName"]);
    Console.WriteLine("mySqlDataReader[\" QuantityPerUnit\"] = " +
        mySqlDataReader["QuantityPerUnit"]);
    Console.WriteLine("mySqlDataReader[\" UnitPrice\"] = " +
        mySqlDataReader["UnitPrice"]);
}

mySqlDataReader.Close();
mySqlConnection.Close();
}

```

---

The output from this program is as follows:

```

mySqlDataReader["ProductID"] = 1
mySqlDataReader["ProductName"] = Chai
mySqlDataReader["QuantityPerUnit"] = 10 boxes x 20 bags
mySqlDataReader["UnitPrice"] = 18

```

## Using the *SchemaOnly* Command Behavior

You use the `SchemaOnly` command behavior to indicate that your `Command` object returns information only about the columns retrieved by a `SELECT` statement, or all the columns when you use a `TableDirect` command.

For example, let's say you have a `Command` object named `mySqlCommand` with the `CommandText` property set as follows:

```

mySqlCommand.CommandText =
    "SELECT ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "WHERE ProductID = 1";

```

Next, the following example passes the `CommandBehavior.SchemaOnly` value to the `ExecuteReader()` method, indicating that the `Command` object returns information about the schema:

```

SqlDataReader productsSqlDataReader =
    mySqlCommand.ExecuteReader(CommandBehavior.SchemaOnly);

```

In this example, since the `ProductID`, `ProductName`, and `UnitPrice` columns of the `Products` table were used in the earlier `SELECT` statement, information about those columns is retrieved instead of the column values.

You get the information about the columns using the `GetSchemaTable()` method of your `SqlData-Reader` object. The `GetSchemaTable()` method returns a `DataTable` object with columns that contain the details of the retrieved database columns:

```

DataTable myDataTable = productsSqlDataReader.GetSchemaTable();

```

To display the values in the `DataTable` object, you can use the following loop that displays the `DataTable` column names and the contents of each `DataTable` column:

```

foreach (DataRow myDataRow in myDataTable.Rows)
{
    foreach ( DataColumn myDataColumn in myDataTable.Columns)

```

```

{
    Console.WriteLine(myDataColumn + " = " +
        myDataRow[myDataColumn]);
    if (myDataColumn.ToString() == "ProviderType")
    {
        Console.WriteLine(myDataColumn + " = " +
            ((System.Data.SqlDbType) myDataRow[myDataColumn]));
    }
}
}
}

```

Notice that this code features two `foreach` loops. The outer loop iterates over the `DataRow` objects in `myDataTable`, and the inner loop iterates over the  `DataColumn` objects in the current `DataRow`. Don't worry too much about the details of accessing a `DataTable` just yet: you'll learn the details in [Chapter 10](#), "Using `DataSet` Objects to Store Data."

The `if` statement in the inner `foreach` loop requires a little explanation. What I'm doing is examining the `myDataColumn` to see if it contains the `ProviderType`. `ProviderType` contains a number value that indicates the SQL Server type of the database column. I cast this number to `System.Data.SqlDbType`, which is an enumeration that defines the SQL Server column types, as you'll see later in the "[Supplying Parameters to Commands](#)" section. [Table 8.9](#) in that section shows the `SqlDbType` enumeration values. By casting the `ProviderType` number to `SqlDbType`, you can see the actual name of the SQL Server column type.

The first iteration of the outer loop displays all the  `DataColumn` object values for the first `DataRow` object. This causes the following output to be produced and shows the schema details for the `ProductID` column; notice the `ProviderType` number and name that indicate `ProductID` is a SQL Server int:

```

ColumnName = ProductID
ColumnOrdinal = 0
ColumnSize = 4
NumericPrecision = 0
NumericScale = 0
IsUnicode =
IsKey =
BaseCatalogName =
BaseColumnName = ProductID
BaseSchemaName =
BaseTableName =
DataType = System.Int32
AllowDBNull = False
ProviderType = 8
ProviderType = Int
IsAliased =
IsExpression =
IsIdentity = True
IsAutoIncrement = True
IsRowVersion =
IsHidden =
IsLong = False
IsReadOnly = True

```

The meanings of these results are shown in [Table 8.6](#).

**Table 8.6: SCHEMA COLUMN VALUES**

VALUE	DESCRIPTION
ColumnName	Name of the column.
ColumnOrdinal	Ordinal of the column.

ColumnSize	Maximum length (in characters) of a column value. For fixed-length SQL Server types such as int, the ColumnSize is the length of that type.
NumericPrecision	Total number of digits used to represent a floating-point type. An example of a floating-point type is the SQL Server float type. The total number of digits includes the digits to the left and right of the decimal point.
NumericScale	Total number of digits to the right of the decimal point in a floating-point type.
IsUnicode	Boolean true/false value that indicates whether two rows can have the same value in the current column.
IsKey	Boolean true/false value that indicates whether the column is part of the primary key.
BaseCatalogName	Name of the catalog in the database that contains the column. BaseCatalogName defaults to null.
BaseColumnName	Name of the column in the database. This will differ from the ColumnName if you use an alias for the column. BaseColumnName defaults to null.
BaseSchemaName	Name of the schema in the database that contains the column. BaseSchemaName defaults to null.
BaseTableName	Name of the table or view in the database that contains the column. BaseTableName defaults to null.
DataType	.NET type used to represent the column. You'll learn about the .NET types in the <a href="#">next chapter</a> .
AllowDBNull	Boolean true/false value that indicates whether the column can accept a database null value.
ProviderType	Indicates the column's database type.
IsAliased	Boolean true/false value that indicates whether the column is an alias.
IsExpression	Boolean true/false value that indicates whether the column is an expression.
IsIdentity	Boolean true/false value that indicates whether the column is an identity.
IsAutoIncrement	Boolean true/false value that indicates whether the column is automatically assigned a value for a new row and that value is automatically incremented.
IsRowVersion	Boolean true/false value that indicates whether the column contains a persistent row identifier that cannot be written to.
IsHidden	Boolean true/false value that indicates whether the column is hidden.
IsLong	Boolean true/false value that indicates whether the column contains a binary long object (BLOB). A BLOB contains a long string of binary data.
IsReadOnly	Boolean true/false value that indicates whether the column can be modified.

[Listing 8.3](#) illustrates the effect of using CommandBehavior.SchemaOnly and displays the schema details for the ProductID, ProductName, and UnitPrice columns.

### **Listing 8.3: SCHEMAONLYCOMMANDBEHAVIOR.CS**

```
/*
 SchemaOnlyCommandBehavior.cs illustrates how to read a table schema
 */

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

class SchemaOnlyCommandBehavior
```

```

{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT ProductID, ProductName, UnitPrice " +
            "FROM Products " +
            "WHERE ProductID = 1";
        mySqlConnection.Open();
        // pass the CommandBehavior.SchemaOnly constant to the
        // ExecuteReader() method to get the schema
        SqlDataReader productsSqlDataReader =
            mySqlCommand.ExecuteReader(CommandBehavior.SchemaOnly);
        // read the DataTable containing the schema from the DataReader
        DataTable myDataTable = productsSqlDataReader.GetSchemaTable();
        // display the rows and columns in the DataTable
        foreach (DataRow myDataRow in myDataTable.Rows)
        {
            Console.WriteLine("\nNew column details follow:");
            foreach (DataColumn my DataColumn in myDataTable.Columns)
            {
                Console.WriteLine(my DataColumn + "= " +
                    myDataRow[my DataColumn]);
                if (my DataColumn.ToString() == "ProviderType")
                {
                    Console.WriteLine(my DataColumn + "= " +
                        ((System.Data.SqlDbType) myDataRow[my DataColumn]));
                }
            }
        }
        productsSqlDataReader.Close();
        mySqlConnection.Close();
    }
}

```

---

You should notice the different details for the `ProductID`, `ProductName`, and `UnitPrice` columns in the output that follows:

```

New column details follow:
ColumnName = ProductID
ColumnOrdinal = 0
ColumnSize = 4
NumericPrecision = 0
NumericScale = 0
IsUnicode =
IsKey =
BaseCatalogName =
BaseColumnName = ProductID
BaseSchemaName =
BaseTableName =
DataType = System.Int32

```

```
AllowDBNull = False
ProviderType = 8
ProviderType = Int
IsAliased =
IsExpression =
IsIdentity = True
IsAutoIncrement = True
IsRowVersion =
IsHidden =
IsLong = False
IsReadOnly = True
```

New column details follow:

```
ColumnName = ProductName
ColumnOrdinal = 1
ColumnSize = 40
NumericPrecision = 0
NumericScale = 0
IsUnique =
IsKey =
BaseCatalogName =
BaseColumnName = ProductName
BaseSchemaName =
BaseTableName =
DataType = System.String
AllowDBNull = False
ProviderType = 12
ProviderType = NVarChar
IsAliased =
IsExpression =
IsIdentity = False
IsAutoIncrement = False
IsRowVersion =
IsHidden =
IsLong = False
IsReadOnly = False
```

New column details follow:

```
ColumnName = UnitPrice
ColumnOrdinal = 2
ColumnSize = 8
NumericPrecision = 0
NumericScale = 0
IsUnique =
IsKey =
BaseCatalogName =
BaseColumnName = UnitPrice
BaseSchemaName =
BaseTableName =
DataType = System.Decimal
AllowDBNull = True
ProviderType = 9
ProviderType = Money
IsAliased =
IsExpression =
IsIdentity = False
IsAutoIncrement = False
IsRowVersion =
IsHidden =
IsLong = False
```

```
IsReadOnly = False
```

## Executing a *TableDirect* Statement Using the *ExecuteReader()* Method

When you set the *CommandType* property of a *Command* object to *TableDirect*, you specify that you want to retrieve all the rows and columns of a particular table. You specify the name of the table to retrieve from in the *CommandText* property.

**Warning** *SqlCommand* objects don't support the *CommandType* of *TableDirect*. The example in this section will use an *OleDbCommand* object instead.

As you know, you can use a *SqlConnection* object to connect to SQL Server. You can also use an *OleDbConnection* object to connect to SQL Server. You simply set the provider to *SQLOLEDB* in the connection string passed to the *OleDbConnection* constructor. For example:

```
OleDbConnection myOleDbConnection =
    new OleDbConnection(
        "Provider=SQLOLEDB;server=localhost;database=Northwind;" +
        "uid=sa;pwd=sa"
    );
```

Next, you create an *OleDbConnection* object:

```
OleDbCommand myOleDbCommand = myOleDbConnection.CreateCommand();
```

You then set the *CommandType* of *myOleDbConnection* to *CommandType.TableDirect*:

```
myOleDbCommand.CommandType = CommandType.TableDirect;
```

Next, you specify the name of the table to retrieve from using the *CommandText* property. The following example sets the *CommandText* property of *myOleDbCommand* to *Products*:

```
myOleDbCommand.CommandText = "Products";
```

You next open the database connection:

```
myOleDbConnection.Open();
```

Finally, you execute *myOleDbCommand* using the *ExecuteReader()* method:

```
OleDbDataReader myOleDbDataReader = myOleDbCommand.ExecuteReader();
```

The SQL statement actually executed is *SELECT \* FROM Products*, which retrieves all the rows and columns from the *Products* table.

[Listing 8.4](#) illustrates the code shown in this section.

### Listing 8.4: EXECUTETABLEDIRECT.CS

---

```
/*
 ExecuteTableDirect.cs illustrates how to execute a
 TableDirect command
 */

using System;
using System.Data;
using System.Data.OleDb;

class ExecuteTableDirect
{
    public static void Main()
    {
        OleDbConnection myOleDbConnection =
```

```

new OleDbConnection(
    "Provider=SQLOLEDB;server=localhost;database=Northwind;" +
    "uid=sa;pwd=sa"
);
OleDbCommand myOleDbCommand = myOleDbConnection.CreateCommand();

// set the CommandType property of the OleDbCommand object to
// TableDirect
myOleDbCommand.CommandType = CommandType.TableDirect;

// set the CommandText property of the OleDbCommand object to
// the name of the table to retrieve from
myOleDbCommand.CommandText = "Products";

myOleDbConnection.Open();

OleDbDataReader myOleDbDataReader = myOleDbCommand.ExecuteReader();

// only read the first 5 rows from the OleDbDataReader object
for (int count = 1; count <= 5; count++)
{
    myOleDbDataReader.Read();
    Console.WriteLine("myOleDbDataReader[\"ProductID\"] = " +
        myOleDbDataReader["ProductID"]);
    Console.WriteLine("myOleDbDataReader[\"ProductName\"] = " +
        myOleDbDataReader["ProductName"]);
    Console.WriteLine("myOleDbDataReader[\"QuantityPerUnit\"] = " +
        myOleDbDataReader["QuantityPerUnit"]);
    Console.WriteLine("myOleDbDataReader[\"UnitPrice\"] = " +
        myOleDbDataReader["UnitPrice"]);
}
myOleDbDataReader.Close();
myOleDbConnection.Close();
}
}

```

---

You'll notice that this program displays only the first five rows from the `Products` table, even though all the rows are retrieved.

The output from this program is as follows:

```

myOleDbDataReader["ProductID"] = 1
myOleDbDataReader["ProductName"] = Chai
myOleDbDataReader["QuantityPerUnit"] = 10 boxes x 20 bags
myOleDbDataReader["UnitPrice"] = 18
myOleDbDataReader["ProductID"] = 2
myOleDbDataReader["ProductName"] = Chang
myOleDbDataReader["QuantityPerUnit"] = 24 - 12 oz bottles
myOleDbDataReader["UnitPrice"] = 19
myOleDbDataReader["ProductID"] = 3
myOleDbDataReader["ProductName"] = Aniseed Syrup
myOleDbDataReader["QuantityPerUnit"] = 12 - 550 ml bottles
myOleDbDataReader["UnitPrice"] = 10
myOleDbDataReader["ProductID"] = 4
myOleDbDataReader["ProductName"] = Chef Anton's Cajun Seasoning
myOleDbDataReader["QuantityPerUnit"] = 48 - 6 oz jars
myOleDbDataReader["UnitPrice"] = 22
myOleDbDataReader["ProductID"] = 5
myOleDbDataReader["ProductName"] = Chef Anton's Gumbo Mix
myOleDbDataReader["QuantityPerUnit"] = 36 boxes

```

```
myOleDbDataReader["UnitPrice"] = 21.35
```

## Executing a ***SELECT*** Statement Using the ***ExecuteScalar()*** Method

You use the **ExecuteScalar()** method to execute SQL **SELECT** statements that return a single value; any other values are ignored. The **ExecuteScalar()** method returns the single result as an object of the **System.Object** class. One use for the **ExecuteScalar()** method is to execute a **SELECT** statement that uses an aggregate function such as **COUNT()** to get the number of rows in a table. Aggregate functions are covered in [Chapter 4](#), "Introduction to Transact-SQL Programming."

For example, the following statement sets the **CommandText** property of the **mySqlCommand** object to a **SELECT** that uses the **COUNT()** function. This **SELECT** returns the number of rows in the **Products** table:

```
mySqlCommand.CommandText =
    "SELECT COUNT(*) " +
    "FROM Products";
```

Next, the following example executes the **SELECT** statement using the **ExecuteScalar()** method:

```
int returnValue = (int) mySqlCommand.ExecuteScalar();
```

You'll notice I cast the generic **object** returned by **ExecuteScalar()** to an **int** before storing the result in the **int returnValue** variable.

[Listing 8.5](#) illustrates the use of the **ExecuteScalar()** method.

---

### Listing 8.5: EXECUTESCALAR.CS

```
/*
 ExecuteScalar.cs illustrates how to use the ExecuteScalar()
 method to run a SELECT statement that returns a single value
 */

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

class ExecuteScalar
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT COUNT(*) " +
            "FROM Products";
        mySqlConnection.Open();

        // call the ExecuteScalar() method of the SqlCommand object
        // to run the SELECT statement
        int returnValue = (int) mySqlCommand.ExecuteScalar();
        Console.WriteLine("mySqlCommand.ExecuteScalar() = " +
            returnValue);

        mySqlConnection.Close();
    }
}
```

The output from this program is as follows:

```
mySqlCommand.ExecuteScalar() = 79
```

Of course, your output might vary depending on the number of rows in your `Products` table.

## Executing a Command that Retrieves Data as XML Using the `ExecuteXMLReader()` Method

You use the `ExecuteXmlReader()` method to execute a SQL SELECT statement that returns XML data. The `ExecuteXmlReader()` method returns the results in an `XmlReader` object, which you then use to read the retrieved XML data.

**Note** The `ExecuteXmlReader()` method applies only to the `SqlCommand` class.

SQL Server extends standard SQL to allow you to query the database and get results back as XML. Specifically, you can add a `FOR XML` clause to the end of a `SELECT` statement. The `FOR XML` clause has the following syntax:

```
FOR XML
{RAW | AUTO | EXPLICIT}
[ , XMLDATA ]
[ , ELEMENTS ]
[ , BINARY BASE64 ]
```

[Table 8.7](#) shows the description of the keywords used in the `FOR XML` clause.

**Table 8.7: FOR XML KEYWORDS**

KEYWORD	DESCRIPTION
FOR XML	Specifies that SQL Server is to return results as XML.
RAW	Indicates that each row in the result set is returned as an XML <code>&lt;row&gt;</code> element. Column values become attributes of the <code>&lt;row&gt;</code> element.
AUTO	Specifies that each row in the result set is returned as an XML element with the name of table used in place of the generic <code>&lt;row&gt;</code> element.
EXPLICIT	Indicates that your <code>SELECT</code> statement specifies the parent-child relationship, which is then used by SQL Server to create XML with the appropriate nesting structure.
XMLDATA	Specifies that the Document Type Definition is to be included in the returned XML.
ELEMENTS	Indicates that the columns are returned as subelements of the row. Otherwise, the columns are returned as attributes of the row. You can use this option only with <code>AUTO</code> .
BINARY BASE64	Specifies that any binary data returned by the query is encoded in base 64. If you want to retrieve binary data using <code>RAW</code> and <code>EXPLICIT</code> mode, then you must use <code>BINARY BASE64</code> . In <code>AUTO</code> mode, binary data is returned as a reference by default.

You'll see a simple example of the `FOR XML` clause here, and you'll learn the full details of this clause in [Chapter 16](#), "Using SQL Server's XML Support."

The following example sets the `CommandText` property of `mySqlCommand` to a `SELECT` statement that uses the `FOR XML AUTO` clause. This `SELECT` statement returns the first five rows from the `Products` table as XML.

```
mySqlCommand.CommandText =
"SELECT TOP 5 ProductID, ProductName, UnitPrice " +
"FROM Products " +
"ORDER BY ProductID " +
"FOR XML AUTO";
```

Next, the following statement executes the `SELECT` using the `ExecuteXmlReader()` method:

```
XmlReader myXmlReader = mySqlCommand.ExecuteXmlReader();
```

**Note** The `XmlReader` class is defined in the `System.Xml` namespace.

To start reading the XML from the `XmlReader` object, you use the `Read()` method. You then check to make sure you're not at the end of the rows using the `EOF` property of the `XmlReader` object. `EOF` returns `true` if there are no more rows to read, otherwise it returns `false`. You use the `ReadOuterXml()` method to read the actual XML from the `XmlReader` object. The following example illustrates how to read XML from `myXmlReader`:

```
myXmlReader.Read();
while (!myXmlReader.EOF)
{
    Console.WriteLine(myXmlReader.ReadOuterXml());
}
```

[Listing 8.6](#) illustrates the use of the `ExecuteXmlReader()` method.

### **Listing 8.6: EXECUTEXMLREADER.CS**

---

```
/*
 ExecuteXmlReader.cs illustrates how to use the ExecuteXmlReader()
 method to run a SELECT statement that returns XML
 */

using System;
using System.Data;
using System.Data.SqlClient;
using System.Xml;

class ExecuteXmlReader
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        // set the CommandText property of the SqlCommand object to
        // a SELECT statement that retrieves XML
        mySqlCommand.CommandText =
            "SELECT TOP 5 ProductID, ProductName, UnitPrice " +
            "FROM Products " +
            "ORDER BY ProductID " +
            "FOR XML AUTO";

        mySqlConnection.Open();

        // create a SqlDataReader object and call the ExecuteReader()
        // method of the SqlCommand object to run the SELECT statement
        XmlReader myXmlReader = mySqlCommand.ExecuteXmlReader();

        // read the rows from the XmlReader object using the Read() method
        myXmlReader.Read();
        while (!myXmlReader.EOF)
        {
            Console.WriteLine(myXmlReader.ReadOuterXml());
        }
    }
}
```

```
        }  
  
        myXmlReader.Close();  
        mySqlConnection.Close();  
    }  
}
```

---

You'll notice I imported the `System.Xml` namespace near the beginning of this program.

The output from this program is as follows:

```
<Products ProductID="1" ProductName="Chai" UnitPrice="18.0000"/>  
<Products ProductID="2" ProductName="Chang" UnitPrice="19.0000"/>  
<Products ProductID="3" ProductName="Aniseed Syrup" UnitPrice="10.0000"/>  
<Products ProductID="4" ProductName="Chef Anton's Cajun Seasoning"  
UnitPrice="22.0000"/>  
<Products ProductID="5" ProductName="Chef Anton's Gumbo Mix" UnitPrice="21.  
3500"/>
```

Notice that each of the 5 rows from the `Products` table is returned as XML.

**Team LiB**

◀ PREVIOUS ▶ NEXT ▶

## Executing Commands that Modify Information in the Database

You can use the `ExecuteNonQuery()` method of a `Command` object to execute any command that doesn't return a result set from the database. In this section, you'll learn how to use the `ExecuteNonQuery()` method to execute commands that modify information in the database.

You can use the `ExecuteNonQuery()` method to execute SQL `INSERT`, `UPDATE`, and `DELETE` statements. You can also use the `ExecuteNonQuery()` method to call stored procedures that don't return a value, or issue Data Definition Language (DDL) statements such as `CREATE TABLE` and `CREATE INDEX`. (DDL was covered in [Chapter 3](#), "Introduction to the Structured Query Language.") [Table 8.8](#) summarizes the `ExecuteNonQuery()` method.

**Table 8.8: THE `ExecuteNonQuery()` METHOD**

METHOD	RETURN TYPE	DESCRIPTION
<code>ExecuteNonQuery()</code>	<code>int</code>	Used to execute SQL statements that don't return a result set, such as <code>INSERT</code> , <code>UPDATE</code> , and <code>DELETE</code> statements, DDL statements, or stored procedure calls that don't return a result set. The <code>int</code> value returned is the number of database rows affected by the command, if any.

You'll learn how to execute `INSERT`, `UPDATE`, and `DELETE` statements, and how to execute DDL statements in this section. You'll learn how to execute stored procedure calls later in the "[Executing SQL Server Stored Procedures](#)" section.

### Executing `INSERT`, `UPDATE`, and `DELETE` Statements Using the `ExecuteNonQuery()` Method

Let's take a look at an example that executes an `INSERT` statement using the `ExecuteNonQuery()` method. First, a `Command` object is needed:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
```

Next, you set the `CommandText` property of your `Command` object to the `INSERT` statement. The following example sets the `CommandText` property of `mySqlCommand` to an `INSERT` statement that adds a row to the `Customers` table:

```
mySqlCommand.CommandText =
    "INSERT INTO Customers (" +
    " CustomerID, CompanyName" +
    ") VALUES (" +
    "'J2COM', 'Jason Price Corporation'" +
    ");"
```

Finally, you execute the `INSERT` statement using the `ExecuteNonQuery()` method:

```
int numberOfRows = mySqlCommand.ExecuteNonQuery();
```

The `ExecuteNonQuery()` method returns an `int` value that indicates the number of rows affected by the command. In this example, the value returned is the number of rows added to the `Customers` table, which is 1 since one row was added by the `INSERT` statement.

Let's take a look at an example that executes an `UPDATE` statement to modify the new row just added. The following code sets the `CommandText` property of `mySqlCommand` to an `UPDATE` statement that modifies the `CompanyName` column of the new row, and then calls the `ExecuteNonQuery()` method to execute the `UPDATE`:

```
mySqlCommand.CommandText =
    "UPDATE Customers " +
```

```
"SET CompanyName = 'New Company' " +
"WHERE CustomerID = 'J2COM' ";
numberOfRows = mySqlCommand.ExecuteNonQuery();
```

The `ExecuteNonQuery()` method returns the number of rows modified by the `UPDATE` statement, which is 1 since one row was modified.

Finally, let's take a look at an example that executes a `DELETE` statement to remove the new row:

```
mySqlCommand.CommandText =
"DELETE FROM Customers " +
"WHERE CustomerID = 'J2COM' ";
numberOfRows = mySqlCommand.ExecuteNonQuery();
```

`ExecuteNonQuery()` returns 1 again because only one row was removed by the `DELETE` statement.

[Listing 8.7](#) illustrates the use of the `ExecuteNonQuery()` method to execute the `INSERT`, `UPDATE`, and `DELETE` statements shown in this section. This program features a procedure named `DisplayRow()` that retrieves and displays the details of a specified row from the `Customers` table. `DisplayRow()` is used in the program to show the result of the `INSERT` and `UPDATE` statements.

### **Listing 8.7: EXECUTEINSERTUPDATEDELETE.CS**

---

```
/*
ExecuteInsertUpdateDelete.cs illustrates how to use the
ExecuteNonQuery() method to run INSERT, UPDATE,
and DELETE statements
*/
using System;
using System.Data;
using System.Data.SqlClient;

class ExecuteInsertUpdateDelete
{
    public static void DisplayRow(
        SqlCommand mySqlCommand, string CustomerID
    )
    {
        mySqlCommand.CommandText =
            "SELECT CustomerID, CompanyName " +
            "FROM Customers " +
            "WHERE CustomerID = '" + CustomerID + "'";

        SqlDataReader mySqlDataReader =
            mySqlCommand.ExecuteReader();

        while (mySqlDataReader.Read())
        {
            Console.WriteLine("mySqlDataReader[\"CustomerID\"] = " +
                mySqlDataReader["CustomerID"]);
            Console.WriteLine("mySqlDataReader[\"CompanyName\"] = " +
                mySqlDataReader["CompanyName"]);
        }

        mySqlDataReader.Close();
    }

    public static void Main()
    {
        SqlConnection mySqlConnection =
```

```

new SqlConnection(
    "server=localhost;database=Northwind;uid=sa;pwd=sa"
);

// create a SqlCommand object and set its Commandtext property
// to an INSERT statement
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "INSERT INTO Customers ( " +
    " CustomerID, CompanyName" +
    ") VALUES ( " +
    " 'J2COM', 'Jason Price Corporation' " +
    " ) ";
mySqlConnection.Open();

// call the ExecuteNonQuery() method of the SqlCommand object
// to run the INSERT statement
int numberOfRows = mySqlCommand.ExecuteNonQuery();
Console.WriteLine("Number of rows added = " + numberOfRows);
DisplayRow(mySqlCommand, "J2COM");

// set the CommandText property of the SqlCommand object to
// an UPDATE statement
mySqlCommand.CommandText =
    "UPDATE Customers " +
    "SET CompanyName = 'New Company' " +
    "WHERE CustomerID = 'J2COM' ";

// call the ExecuteNonQuery() method of the SqlCommand object
// to run the UPDATE statement
numberOfRows = mySqlCommand.ExecuteNonQuery();
Console.WriteLine("Number of rows updated = " + numberOfRows);
DisplayRow(mySqlCommand, "J2COM");

// set the CommandText property of the SqlCommand object to
// a DELETE statement
mySqlCommand.CommandText =
    "DELETE FROM Customers " +
    "WHERE CustomerID = 'J2COM' ";

// call the ExecuteNonQuery() method of the SqlCommand object
// to run the DELETE statement
numberOfRows = mySqlCommand.ExecuteNonQuery();
Console.WriteLine("Number of rows deleted = " + numberOfRows);

mySqlConnection.Close();
}
}

```

---

The output from this program is as follows:

```

Number of rows added = 1
mySqlDataReader["CustomerID"] = J2COM
mySqlDataReader["CompanyName"] = Jason Price Corporation
Number of rows updated = 1
mySqlDataReader["CustomerID"] = J2COM
mySqlDataReader["CompanyName"] = New Company
Number of rows deleted = 1

```

## Executing DDL Statements Using the *ExecuteNonQuery()* Method

In addition to running `INSERT`, `UPDATE`, and `DELETE` statements, you can also use the `ExecuteNonQuery()` method to execute DDL statements such as `CREATE TABLE`.

Let's take a look at an example that executes a `CREATE TABLE` statement, followed by an `ALTER TABLE` statement, followed by a `DROP TABLE` statement. First, a `Command` object is needed:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
```

Next, you set the `CommandText` property of the `Command` object to the `CREATE TABLE` statement. The following example sets the `CommandText` property of `mySqlCommand` to a `CREATE TABLE` statement that creates a table named `MyPersons` to store information about people:

```
mySqlCommand.CommandText =
    "CREATE TABLE MyPersons (" +
    "    PersonID int CONSTRAINT PK_Persons PRIMARY KEY, " +
    "    FirstName nvarchar(15) NOT NULL, " +
    "    LastName nvarchar(15) NOT NULL, " +
    "    DateOfBirth datetime" +
    " )";
```

Next, you call the `ExecuteNonQuery()` method to execute the `CREATE TABLE` statement:

```
int result = mySqlCommand.ExecuteNonQuery();
```

Since a `CREATE TABLE` statement doesn't affect any rows, `ExecuteNonQuery()` returns the value -1.

The next example executes an `ALTER TABLE` statement to add a foreign key constraint to the `MyPersons` table:

```
mySqlCommand.CommandText =
    "ALTER TABLE MyPersons " +
    "ADD EmployerID nchar(5) CONSTRAINT FK_Persons_Customers " +
    "REFERENCES Customers(CustomerID)";
result = mySqlCommand.ExecuteNonQuery();
```

Once again, `ExecuteNonQuery()` returns -1 since the `ALTER TABLE` statement doesn't affect any rows.

The final example executes a `DROP TABLE` statement to drop the `MyPersons` table:

```
mySqlCommand.CommandText = "DROP TABLE MyPersons";
result = mySqlCommand.ExecuteNonQuery();
```

`ExecuteNonQuery()` returns -1 again.

[Listing 8.8](#) illustrates the use of the `ExecuteNonQuery()` method to execute the DDL statements shown in this section.

### **Listing 8.8: EXECUTEDDL.CS**

---

```
/*
 ExecuteDDL.cs illustrates how to use the ExecuteNonQuery()
 method to run DDL statements
 */

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

class ExecuteDDL
{
```

```

public static void Main()
{
    SqlConnection mySqlConnection =
        new SqlConnection(
            "server=localhost;database=Northwind;uid=sa;pwd=sa"
        );

    SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

    // set the CommandText property of the SqlCommand object to
    // a CREATE TABLE statement
    mySqlCommand.CommandText =
        "CREATE TABLE MyPersons (" +
        "    PersonID int CONSTRAINT PK_Persons PRIMARY KEY," +
        "    FirstName nvarchar(15) NOT NULL," +
        "    LastName nvarchar(15) NOT NULL," +
        "    DateOfBirth datetime" +
        ")";

    mySqlConnection.Open();

    // call the ExecuteNonQuery() method of the SqlCommand object
    // to run the CREATE TABLE statement
    Console.WriteLine("Creating MyPersons table");
    int result = mySqlCommand.ExecuteNonQuery();
    Console.WriteLine("mySqlCommand.ExecuteNonQuery() = " + result);

    // set the CommandText property of the SqlCommand object to
    // an ALTER TABLE statement
    mySqlCommand.CommandText =
        "ALTER TABLE MyPersons " +
        "ADD EmployerID nchar(5) CONSTRAINT FK_Persons_Customers " +
        "REFERENCES Customers(CustomerID)";

    // call the ExecuteNonQuery() method of the SqlCommand object
    // to run the ALTER TABLE statement
    Console.WriteLine("Altering MyPersons table");
    result = mySqlCommand.ExecuteNonQuery();
    Console.WriteLine("mySqlCommand.ExecuteNonQuery() = " + result);

    // set the CommandText property of the SqlCommand object to
    // a DROP TABLE statement
    mySqlCommand.CommandText = "DROP TABLE MyPersons";

    // call the ExecuteNonQuery() method of the SqlCommand object
    // to run the DROP TABLE statement
    Console.WriteLine("Dropping MyPersons table");
    result = mySqlCommand.ExecuteNonQuery();
    Console.WriteLine("mySqlCommand.ExecuteNonQuery() = " + result);

    mySqlConnection.Close();
}
}

```

---

The output from this program is as follows:

```

Creating MyPersons table
mySqlCommand.ExecuteNonQuery() = -1
Altering MyPersons table

```

## Executing Commands that Modify Information in the Database

```
mySqlCommand.ExecuteNonQuery() = -1  
Dropping MyPersons table  
mySqlCommand.ExecuteNonQuery() = -1
```

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## Introducing Transactions

In [Chapter 3](#), you saw how you can group SQL statements together into transactions. The transaction is then committed or rolled back as one unit. For example, in the case of a banking transaction, you might want to withdraw money from one account and deposit it into another. You would then commit both of these changes as one unit, or if there's a problem, roll back both changes. You'll be introduced to using transactions in ADO.NET in this section.

There are three `Transaction` classes: `SqlTransaction`, `OleDbTransaction`, and `OdbcTransaction`, and you use an object of one of these classes to represent a transaction in ADO.NET. I'll show you how to use an object of the `SqlTransaction` class in this section.

Let's consider an example transaction that consists of two `INSERT` statements. The first `INSERT` statement will add a row to the `Customers` table, and the second one will add a row to the `Orders` table. The new row in the `Orders` table will reference the new row in the `Customers` table, and the two `INSERT` statements are as follows:

```
INSERT INTO Customers
    CustomerID, CompanyName
) VALUES
    'J3COM', 'Jason Price Corporation'
)

INSERT INTO Orders (
    CustomerID
) VALUES (
    'J3COM'
)
```

You can use the following steps to perform these two `INSERT` statements using a `SqlTransaction` object:

1. Create a `SqlTransaction` object and start the transaction by calling the `BeginTransaction()` method of the `SqlConnection` object.
2. Create a `SqlCommand` object to hold the SQL statement.
3. Set the `Transaction` property for the `SqlCommand` object to the `SqlTransaction` object created in step 1.
4. Set the `CommandText` property of the `SqlCommand` object to the first `INSERT` statement. This `INSERT` statement adds a row to the `Customers` table.
5. Run the first `INSERT` statement using the `ExecuteNonQuery()` method of the `SqlCommand` object. This method is used because an `INSERT` statement doesn't return a result set.
6. Set the `CommandText` property of the `SqlCommand` object to the second `INSERT` statement. This statement adds a row to the `Orders` table.
7. Run the second `INSERT` statement using the `ExecuteNonQuery()` method of the `SqlCommand` object.
8. Commit the transaction using the `Commit()` method of the `SqlTransaction` object. This makes the two new rows added by the `INSERT` statements permanent in the database.

[Listing 8.9](#) illustrates these steps.

### **Listing 8.9: EXECUTETRANSACTION.CS**

---

```
/*
 ExecuteTransaction.cs illustrates the use of a transaction
 */
```

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

class ExecuteTransaction
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        mySqlConnection.Open();

        // step 1: create a SqlTransaction object and start the transaction
        // by calling the BeginTransaction() method of the SqlConnection
        // object
        SqlTransaction mySqlTransaction =
            mySqlConnection.BeginTransaction();

        // step 2: create a SqlCommand object to hold a SQL statement
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        // step 3: set the Transaction property for the SqlCommand object
        mySqlCommand.Transaction = mySqlTransaction;

        // step 4: set the CommandText property of the SqlCommand object to
        // the first INSERT statement
        mySqlCommand.CommandText =
            "INSERT INTO Customers (" +
            " CustomerID, CompanyName" +
            ") VALUES (" +
            " 'J3COM', 'Jason Price Corporation'" +
            ")";

        // step 5: run the first INSERT statement
        Console.WriteLine("Running first INSERT statement");
        mySqlCommand.ExecuteNonQuery();

        // step 6: set the CommandText property of the SqlCommand object to
        // the second INSERT statement
        mySqlCommand.CommandText =
            "INSERT INTO Orders (" +
            " CustomerID" +
            ") VALUES (" +
            " 'J3COM'" +
            ")";

        // step 7: run the second INSERT statement
        Console.WriteLine("Running second INSERT statement");
        mySqlCommand.ExecuteNonQuery();

        // step 8: commit the transaction using the Commit() method
        // of the SqlTransaction object
        Console.WriteLine("Committing transaction");
        mySqlTransaction.Commit();

        mySqlConnection.Close();
    }
}
```

```
}
```

**Note** If you wanted to undo the SQL statements that make up the transaction, you can use the *Rollback()* method instead of the *Commit()* method. By default, transactions are rolled back. Always use the *Commit()* or *Rollback()* methods to explicitly indicate whether you want to commit or roll back your transactions.

The output from this program is as follows:

```
Running first INSERT statement
Running second INSERT statement
Committing transaction
```

If you want to run the program more than once, you'll need to remove the row added to the *Customers* and *Orders* table using the following *DELETE* statements (you can do this using the Query Analyzer tool):

```
DELETE FROM Orders
WHERE CustomerID = 'J3COM'

DELETE FROM Customers
WHERE CustomerID = 'J3COM'
```

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Supplying Parameters to Commands

In the examples you've seen up to this point, the values for each column have been hard-coded in the SQL statements. For example, in [Listing 8.9](#), shown earlier, the `INSERT` statement that added the row to the `Customers` table was:

```
INSERT INTO Customers
    CustomerID, CompanyName
) VALUES
    'J3COM', 'Jason Price Corporation'
)
```

As you can see, the values for the `CustomerID` and `CompanyName` columns are hard-coded to '`J3COM`' and '`Jason Price Corporation`'. If you had to execute many such `INSERT` statements, hard-coding column values would be tiresome and inefficient. Fortunately, you can use parameters to solve this problem. Parameters allow you specify different column values when running your program.

To execute a command containing parameters, you use the following high-level steps:

1. Create a `Command` object containing a SQL statement with parameter placeholders. These placeholders mark the position where a parameter will be supplied.
2. Add parameters to the `Command` object.
3. Set the parameters to specified values.
4. Execute the command.

Let's take a look at the details of the four steps when using parameters with SQL Server.

### Step 1: Create a *Command* Object Containing a SQL Statement with Parameter Placeholders

This is straightforward: wherever you would normally place a column value in your SQL statement, you specify a parameter placeholder instead. A placeholder marks the position where a value will be supplied later.

The syntax you use for the placeholders depends on the database you are using. With SQL Server, example placeholders would be `@CustomerID` and `@CompanyName`. The following `INSERT` statement uses these placeholders for the `CustomerID`, `CompanyName`, and `ContactName` column values of the `Customers` table:

```
INSERT INTO Customers
    CustomerID, CompanyName, ContactName
) VALUES
    @CustomerID, @CompanyName, @ContactName
)
```

You can use a placeholder anywhere a column value is valid in a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. Here are some examples of `SELECT`, `UPDATE`, and `DELETE` statements with placeholders:

```
SELECT *
FROM Customers
WHERE CustomerID = @CustomerID
```

```
UPDATE Customers
SET CompanyName = @CompanyName
WHERE CustomerID = @CustomerID
```

```
DELETE FROM Customers
WHERE CustomerID = @CustomerID
```

Let's take a look at some code that creates a `SqlCommand` object and sets its `CommandText` property to an `INSERT` statement:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "INSERT INTO Customers ( " +
    " CustomerID, CompanyName, ContactName" +
    " ) VALUES ( " +
    " @CustomerID, @CompanyName, @ContactName" +
    " )";
```

This `INSERT` statement will be used to add a row to the `Customers` table. The column values for this row will be specified using parameters. All that's been done in the previous code is to create a `SqlCommand` object with an `INSERT` statement that has placeholders. Before you can execute this `INSERT` statement, you need to add the actual parameters to the `SqlCommand` object-and you'll do that in the next step.

## Step 2: Add Parameters to the Command Object

To add parameters to your `Command` object, you use the `Add()` method. It is overloaded, and the version used in this section accepts three parameters:

- The placeholder string for the parameter in your SQL statement. For example, `@CustomerID` is the first placeholder in the `INSERT` statement shown in the [previous section](#).
- The type for the column in the database. For SQL Server, these types are defined in the `System.Data.SqlDbType` enumeration. [Table 8.9](#) shows these database types.

**Table 8.9: `SqlDbType` ENUMERATION MEMBERS**

MEMBER	DESCRIPTION
<code>BigInt</code>	A 64-bit signed integer between $-2^{63}$ (-9,223,372,036,854,775,808) and $2^{63}-1$ (9,223,372,036,854,775,807).
<code>Binary</code>	An array of bytes with a maximum length of 8,000.
<code>Bit</code>	An unsigned numeric value that can be 0, 1, or a null reference.
<code>Char</code>	A string of non-Unicode characters with a maximum length of 8,000.
<code>Datetime</code>	A date and time between 12:00:00 AM January 1, 1753 and 11:59:59 PM December 31, 9999. This is accurate to 3.33 milliseconds.
<code>Decimal</code>	Fixed precision and scale numeric value between $-10^{38} + 1$ and $10^{38} - 1$ .
<code>Float</code>	A 64-bit floating-point number between $-1.79769313486232E308$ and $1.79769313486232E308$ with 15 significant figures of precision.
<code>Image</code>	An array of bytes with a maximum length of $2^{31} - 1$ (2,147,483,647).
<code>Int</code>	A 32-bit signed integer between $-2^{31}$ (-2,147,483,648) and $2^{31} - 1$ (2,147,483,647).
<code>Money</code>	A currency value between -922,337,203,685,477.5808 and 922,337,203,685,477.5807. This is accurate to 1/10,000th of a currency unit.
<code>NChar</code>	A string of Unicode characters with a maximum length of 4,000.
<code>Ntext</code>	A string of Unicode characters with a maximum length of $2^{30} - 1$ (1,073,741,823).
<code>NVarChar</code>	A string of Unicode characters with a maximum length of 4,000.
<code>Real</code>	A 32-bit floating-point number between $-3.402823E38$ and $3.402823E38$ with seven significant figures of precision.
<code>SmallDateTime</code>	A date and time between 12:00:00 AM January 1, 1900 and 11:59:59 PM June 6, 2079. This is accurate to 1 minute.

SmallInt	A 16-bit signed integer between $-2^{15}$ (-32,768) and $2^{15} - 1$ (32,767).
SmallMoney	A currency value between -214,748.3648 and 214,748.3647. Accurate to 1/10,000th of a currency unit.
Text	A string of non-Unicode characters with a maximum length of $2^{31} - 1$ (2,147,483,647).
Timestamp	A date and time in the format yyyyymmddhhmmss.
TinyInt	An 8-bit unsigned integer between 0 and $2^8 - 1$ (255).
UniqueIdentifier	A 128-bit integer value (16 bytes) that is unique across all computers and networks.
VarBinary	An array of bytes with a maximum length of 8,000.
VarChar	A string of non-Unicode characters with a maximum length of 4,000.
Variant	A data type that can contain numbers, strings, bytes, or dates.

- The maximum length of the parameter value. You specify this parameter only when using variable length types, for example, Char and VarChar.

Earlier in step 1, the CommandText property for mySqlCommand had three placeholders and was set as follows:

```
mySqlCommand.CommandText =
    "INSERT INTO Customers (" +
    " CustomerID, CompanyName, ContactName" +
    " ) VALUES (" +
    " @CustomerID, @CompanyName, @ContactName" +
    " )";
```

The following statements use the Add( ) method to add the three parameters to mySqlCommand:

```
mySqlCommand.Parameters.Add("@CustomerID", SqlDbType.NChar, 5);
mySqlCommand.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40);
mySqlCommand.Parameters.Add("@ContactName", SqlDbType.NVarChar, 30);
```

Notice that you call the Add( ) method through the Parameters property of mySqlCommand. This requires some explanation. A SqlCommand object stores parameters using a SqlParameterCollection object, which is a collection of SqlParameter objects (a SqlParameter object contains the details of a parameter). One of the SqlParameterCollection methods is Add( ), which you use to add a SqlParameter object to the collection. Therefore, to add a parameter to mySqlCommand, you call the Add( ) method through its Parameters property.

As you can see from the previous code that added the three parameters to mySqlCommand, the @CustomerID parameter is defined as an NChar-a string of Unicode characters with a maximum length of 4,000. A value of 5 is passed as the third parameter to the Add( ) method for @CustomerID, meaning that a maximum of five characters may be supplied as the parameter value. Similarly, the @CompanyName and @ContactName parameters are defined as an NVarChar-a string of Unicode characters-with a maximum length of 40 and 30 characters respectively, as indicated by the third parameter to the Add( ) method. You'll see the setting of these parameters to values in the next step.

### Step 3: Set the Parameters to Specified Values

You use the Value property of each parameter to set it to a specified value in your Command object. These values are substituted for the placeholders in your SQL statement.

The following example uses the Value property to set the values of the parameters added in the [previous section](#):

```
mySqlCommand.Parameters["@CustomerID"].Value = "J4COM";
mySqlCommand.Parameters["@CompanyName"].Value = "J4 Company";
mySqlCommand.Parameters["@ContactName"].Value = "Jason Price";
```

In this example, the `@CustomerID`, `@CompanyName`, and `@ContactName` parameters are set to `J4COM`, `J4 Company`, and `Jason Price` respectively. These values will be substituted for the placeholders in the `INSERT` statement, which becomes

```
INSERT INTO Customers (
    CustomerID, CompanyName, ContactName
) VALUES (
    'J4COM', 'J4 Company', 'Jason Price'
)
```

As you can see, the column values are the same as those specified in the `Value` property for each parameter.

You can also add a parameter and set its value in one step. For example:

```
mySqlCommand.Parameters.Add("@CustomerID", SqlDbType.NChar, 5).Value =
    "J4COM";
```

You can also set a parameter to a null value. As you learned in [Chapter 2](#), "Introduction to Databases," a column defined as null can store a null value. A null value indicates that the column value is unknown. You indicate that a parameter can accept a null value by setting the `IsNullable` property to `true` (the default is `false`). For example:

```
mySqlCommand.Parameters["@ContactName"].IsNullable = true;
```

You can then set the `Value` property of the parameter to null using the `System.DBNull` class. For example:

```
mySqlCommand.Parameters["@ContactName"].Value = DBNull.Value;
```

The `DBNull.Value` property returns a null value. In this example, the final `INSERT` statement becomes:

```
INSERT INTO Customers (
    CustomerID, CompanyName, ContactName
) VALUES (
    'J4COM', 'J4 Company', NULL
)
```

The only thing left to do is to execute the SQL statement.

## Step 4: Execute the Command

To execute the command, you use one of your `Command` object's execute methods. For example:

```
mySqlCommand.ExecuteNonQuery();
```

This runs the `INSERT` statement that adds the new row to the `Customers` table. I used the `ExecuteNonQuery()` method because an `INSERT` statement doesn't return a result set from the database. You can also use this method to execute `UPDATE` and `DELETE` statements. If you were executing a `SELECT` statement, you would use the `ExecuteReader()`, `ExecuteScalar()`, or `ExecuteXmlReader()` methods.

[Listing 8.10](#) illustrates these four steps.

### Listing 8.10: USINGPARAMETERS.CS

---

```
/*
UsingParameters.cs illustrates how to run an INSERT
statement that uses parameters
*/

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

class UsingParameters
{
```

```
public static void Main()
{
    SqlConnection mySqlConnection =
        new SqlConnection(
            "server=localhost;database=Northwind;uid=sa;pwd=sa"
        );

    mySqlConnection.Open();

    // step 1: create a Command object containing a SQL statement
    // with parameter placeholders
    SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
    mySqlCommand.CommandText =
        "INSERT INTO Customers (" +
        "    CustomerID, CompanyName, ContactName" +
        ") VALUES (" +
        "    @CustomerID, @CompanyName, @ContactName" +
        ")";
    
    // step 2: add parameters to the Command object
    mySqlCommand.Parameters.Add("@CustomerID", SqlDbType.NChar, 5);
    mySqlCommand.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40);
    mySqlCommand.Parameters.Add("@ContactName", SqlDbType.NVarChar, 30);
    // step 3: set the parameters to specified values
    mySqlCommand.Parameters["@CustomerID"].Value = "J4COM";
    mySqlCommand.Parameters["@CompanyName"].Value = "J4 Company";
    mySqlCommand.Parameters["@ContactName"].IsNullable = true;
    mySqlCommand.Parameters["@ContactName"].Value = DBNull.Value;

    // step 4: execute the command
    mySqlCommand.ExecuteNonQuery();
    Console.WriteLine("Successfully added row to Customers table");

    mySqlConnection.Close();
}
```

---

The output from this program is as follows:

Successfully added row to Customers table

## Executing SQL Server Stored Procedures

In [Chapter 4](#), you saw how to create and execute SQL Server stored procedures using T-SQL. You execute a stored procedure using the T-SQL EXECUTE statement. In this section, you'll see how to execute SQL Server procedures using ADO.NET.

In [Table 8.1](#), shown earlier in this chapter, I mentioned the CommandType of `StoredProcedure`. Although you can use this `CommandType` to indicate that a command is to execute a stored procedure, you're actually better off using the T-SQL EXECUTE command to execute a stored procedure. This is because you can read values that are returned from a stored procedure through a RETURN statement, which you can't do when setting the `CommandType` to `StoredProcedure`. Also, it's a lot easier to understand your code when you use the EXECUTE command.

There are a couple of ways you can execute a stored procedure depending on whether your procedure returns a result set (a *result set* is one or more rows retrieved from a table by a `SELECT` statement). You'll learn these two ways to execute a stored procedure next.

### Executing a Stored Procedure That Does Not Return a Result Set

If your procedure does *not* return a result set, then you use the following steps to execute it:

1. Create a `Command` object and set its `CommandText` property to an `EXECUTE` statement containing your procedure call.
2. Add any required parameters for the procedure call to your `Command` object, remembering to set the `Direction` property for any output parameters to `ParameterDirection.Output`. These output parameters can be defined using the T-SQL `OUTPUT` keyword in your procedure call, or returned using a `RETURN` statement in your actual procedure.
3. Execute your `Command` object using the `ExecuteNonQuery()` method. You use this method because the procedure doesn't return a result set.
4. Read the values of any output parameters.

You'll see how to use these four steps to call the following two SQL Server stored procedures:

- The first procedure, `AddProduct()`, will return an output parameter defined using the `OUTPUT` keyword.
- The second procedure, `AddProduct2()`, will return an output parameter using the `RETURN` statement.

These examples will show you the possible ways to execute a stored procedure using ADO.NET and read the output parameters.

### Executing the `AddProduct()` Stored Procedure

In [Chapter 4](#), you saw how to create a stored procedure in the SQL Server Northwind database. The procedure you saw was named `AddProduct()`, and [Listing 8.11](#) shows the `AddProduct.sql` script that creates the `AddProduct()` procedure. You saw how to run this script in [Chapter 4](#). If you didn't already run this script when reading [Chapter 4](#), and you want to run the example C# program shown later, you'll need to run this script. `AddProduct()` adds a row to the `Products` table and returns the `ProductID` of the new row as an `OUTPUT` parameter.

#### **Listing 8.11: ADDPRODUCT.SQL**

```
/*
AddProduct.sql creates a procedure that adds a row to the
Products table using values passed as parameters to the
procedure. The procedure returns the ProductID of the new row
in an OUTPUT parameter named @MyProductID
*/
CREATE PROCEDURE AddProduct
    @MyProductID int OUTPUT,
    @MyProductName nvarchar(40),
```

```

@MySupplierID int,
@MyCategoryID int,
@MyQuantityPerUnit nvarchar(20),
@MyUnitPrice money,
@MyUnitsInStock smallint,
@MyUnitsOnOrder smallint,
@MyReorderLevel smallint,
@MyDiscontinued bit

AS
    - insert a row into the Products table
    INSERT INTO Products (
        ProductName, SupplierID, CategoryID, QuantityPerUnit,
        UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel,
        Discontinued
    ) VALUES (
        @MyProductName, @MySupplierID, @MyCategoryID, @MyQuantityPerUnit,
        @MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder, @MyReorderLevel,
        @MyDiscontinued
    )

    - use the SCOPE_IDENTITY() function to get the last
    - identity value inserted into a table performed within
    - the current database session and stored procedure,
    - so SCOPE_IDENTITY returns the ProductID for the new row
    - in the Products table in this case
    SELECT @MyProductID = SCOPE_IDENTITY()

```

---

Notice the OUTPUT parameter named `@MyProductID` returned by `AddProduct()`. Because `AddProduct()` doesn't return a result set, you use the first set of steps outlined earlier. Let's examine the details of these four steps to execute this stored procedure.

### Step 1: Create a Command Object and set its CommandText Property to an EXECUTE Statement

Your first step is to create a `Command` object and set its `CommandText` property to an `EXECUTE` statement containing the call to `AddProduct()`; notice the parameter placeholders used to mark the position where the parameter values will be substituted in [Step 2](#):

```

SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "EXECUTE AddProduct @MyProductID OUTPUT, @MyProductName, " +
    "@MySupplierID, @MyCategoryID, @MyQuantityPerUnit, " +
    "@MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder, " +
    "@MyReorderLevel, @MyDiscontinued";

```

Notice the `OUTPUT` parameter placeholder named `@MyProductID`. This is used to store the `OUTPUT` parameter returned by `AddProduct()`. The other parameter placeholders are used to pass values to `AddProduct()`, which then uses those values in its `INSERT` statement.

### Step 2: Add Any Required Parameters to the Command Object

Your second step is to add any parameters to your `Command` object, remembering to set the `Direction` property for any output parameters to `ParameterDirection.Output`.

In this example, `AddProduct()` expects an output parameter to store the `ProductID` for the new row, and you therefore need to add an output parameter to your `Command` object. You do this by setting the `Direction` property of your parameter to `ParameterDirection.Output`. For example:

```

mySqlCommand.Parameters.Add( "@MyProductID", SqlDbType.Int );
mySqlCommand.Parameters[ "@MyProductID" ].Direction =
    ParameterDirection.Output;

```

The other parameters required to call AddProduct( ) are:

```
mySqlCommand.Parameters.Add(
    "@MyProductName", SqlDbType.NVarChar, 40).Value = "Widget";
mySqlCommand.Parameters.Add(
    "@MySupplierID", SqlDbType.Int).Value = 1;
mySqlCommand.Parameters.Add(
    "@MyCategoryID", SqlDbType.Int).Value = 1;
mySqlCommand.Parameters.Add(
    "@MyQuantityPerUnit", SqlDbType.NVarChar, 20).Value = "1 per box";
mySqlCommand.Parameters.Add(
    "@MyUnitPrice", SqlDbType.Money).Value = 5.99;
mySqlCommand.Parameters.Add(
    "@MyUnitsInStock", SqlDbType.SmallInt).Value = 10;
mySqlCommand.Parameters.Add(
    "@MyUnitsOnOrder", SqlDbType.SmallInt).Value = 5;
mySqlCommand.Parameters.Add(
    "@MyReorderLevel", SqlDbType.SmallInt).Value = 5;
mySqlCommand.Parameters.Add(
    "@MyDiscontinued", SqlDbType.Bit).Value = 1;
```

Notice that the SqlDbType parameter types correspond to the types expected by the AddProduct( ) stored procedure. The values the parameters are set to are then substituted for the placeholders in the EXECUTE statement shown in step 1.

### Step 3: Execute the Command Object Using the ExecuteNonQuery() Method

Your third step is to execute your Command object using the ExecuteNonQuery( ) method. You use ExecuteNonQuery( ) because the AddProduct( ) procedure doesn't return a result set. For example:

```
mySqlCommand.ExecuteNonQuery();
```

### Step 4: Read the Values of any Output Parameters

Your last step is to read the values of any output parameters. AddProduct( ) used one output parameter named @MyProductID. You read this returned value from the Value property of @MyProductID:

```
Console.WriteLine("New ProductID = " +
    mySqlCommand.Parameters["@MyProductID"].Value);
```

This displays the values of the ProductID generated by SQL Server for the new row in the Products table.

[Listing 8.12](#) illustrates these steps to call the AddProduct( ) procedure.

---

### Listing 8.12: EXECUTEADDPRODUCT.CS

---

```
/*
 ExecuteAddProduct.cs illustrates how to call the SQL Server
 AddProduct() stored procedure
 */

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

class ExecuteAddProduct
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
```

```

    );
mySqlConnection.Open();

// step 1: create a Command object and set its CommandText
// property to an EXECUTE statement containing the stored
// procedure call
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "EXECUTE AddProduct @MyProductID OUTPUT, @MyProductName, " +
    "@MySupplierID, @MyCategoryID, @MyQuantityPerUnit, " +
    "@MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder, " +
    "@MyReorderLevel, @MyDiscontinued";

// step 2: add the required parameters to the Command object
mySqlCommand.Parameters.Add("@MyProductID", SqlDbType.Int);
mySqlCommand.Parameters["@MyProductID"].Direction =
    ParameterDirection.Output;
mySqlCommand.Parameters.Add(
    "@MyProductName", SqlDbType.NVarChar, 40).Value = "Widget";
mySqlCommand.Parameters.Add(
    "@MySupplierID", SqlDbType.Int).Value = 1;
mySqlCommand.Parameters.Add(
    "@MyCategoryID", SqlDbType.Int).Value = 1;
mySqlCommand.Parameters.Add(
    "@MyQuantityPerUnit", SqlDbType.NVarChar, 20).Value = "1 per box";
mySqlCommand.Parameters.Add(
    "@MyUnitPrice", SqlDbType.Money).Value = 5.99;
mySqlCommand.Parameters.Add(
    "@MyUnitsInStock", SqlDbType.SmallInt).Value = 10;
mySqlCommand.Parameters.Add(
    "@MyUnitsOnOrder", SqlDbType.SmallInt).Value = 5;
mySqlCommand.Parameters.Add(
    "@MyReorderLevel", SqlDbType.SmallInt).Value = 5;
mySqlCommand.Parameters.Add(
    "@MyDiscontinued", SqlDbType.Bit).Value = 1;

// step 3: execute the Command object using the
// ExecuteNonQuery() method
mySqlCommand.ExecuteNonQuery();

// step 4: read the value of the output parameter
Console.WriteLine("New ProductID = " +
    mySqlCommand.Parameters["@MyProductID"].Value);

mySqlConnection.Close();
}
}

```

---

The output from this program is as follows:

```
New ProductID = 81
```

Of course, depending on the existing rows in your Products table, you'll get a different result.

## Executing the *AddProduct2()* Stored Procedure

As you'll see, the *AddProduct2()* procedure is similar to *AddProduct()*, except that it uses a RETURN statement instead of an OUTPUT parameter to return the ProductID for the new row. [Listing 8.13](#) shows the *AddProduct2.sql* script that creates the *AddProduct2()* procedure. You'll need to run this script before running the C# program.

**Listing 8.13: ADDPRODUCT2.SQL**

```
/*
AddProduct2.sql creates a procedure that adds a row to the
Products table using values passed as parameters to the
procedure. The procedure returns the ProductID of the new row
using a RETURN statement
*/

CREATE PROCEDURE AddProduct2
    @MyProductName nvarchar(40),
    @MySupplierID int,
    @MyCategoryID int,
    @MyQuantityPerUnit nvarchar(20),
    @MyUnitPrice money,
    @MyUnitsInStock smallint,
    @MyUnitsOnOrder smallint,
    @MyReorderLevel smallint,
    @MyDiscontinued bit
AS
    - declare the @MyProductID variable
    DECLARE @MyProductID int

    - insert a row into the Products table
    INSERT INTO Products (
        ProductName, SupplierID, CategoryID, QuantityPerUnit,
        UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel,
        Discontinued
    ) VALUES (
        @MyProductName, @MySupplierID, @MyCategoryID, @MyQuantityPerUnit,
        @MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder, @MyReorderLevel,
        @MyDiscontinued
    )

    - use the SCOPE_IDENTITY() function to get the last
    - identity value inserted into a table performed within
    - the current database session and stored procedure,
    - so SCOPE_IDENTITY returns the ProductID for the new row
    - in the Products table in this case
    SET @MyProductID = SCOPE_IDENTITY()

    RETURN @MyProductID
```

Notice the `RETURN` statement at the end to return `@MyProductID`. Because `AddProduct2()` doesn't return a result set of rows, you use the same four steps shown in the [previous section](#) to execute the procedure using ADO.NET. The only difference is in the construction of your `EXECUTE` command when setting the `CommandText` property in step 1. To call `AddProduct2()` you set the `CommandText` property of your `Command` object as follows:

```
mySqlCommand.CommandText =
    "EXECUTE @MyProductID = AddProduct2 @MyProductName, " +
    "@MySupplierID, @MyCategoryID, @MyQuantityPerUnit, " +
    "@MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder, " +
    "@MyReorderLevel, @MyDiscontinued";
```

Notice the change in the position of the `@MyProductID` parameter: it is shifted to just after the `EXECUTE` and set equal to the value returned by `AddProduct2()`. This change is made because `AddProduct2()` uses a `RETURN` statement to output the `@MyProductID` value. The rest of the C# code required to call `AddProduct2()` is the same as that shown earlier in [Listing 8.12](#).

**Note** Because only the `EXECUTE` is different, I've omitted the program that calls `AddProduct2()` from this book. You can see this program in the `ExecuteAddProduct2.cs` file I've provided. Feel free to examine and run it.

## Executing a Stored Procedure that Does Return a Result Set

If your procedure `does` return a result set, then you use the following steps to execute it:

1. Create a `Command` object and set its `CommandText` property to an `EXECUTE` statement containing your procedure call.
2. Add any required parameters to your `Command` object, remembering to set the `Direction` property for any output parameters to `ParameterDirection.Output`.
3. Execute your command using the `ExecuteReader()` method, storing the returned `DataReader` object.
4. Read the rows in the result set using your `DataReader` object.
5. Close your `DataReader` object. You *must* do this before you can read any output parameters.
6. Read the values of any output parameters.

In the following example, you'll see a stored procedure named `AddProduct3()` that will return a result set along with an output parameter using a `RETURN` statement.

The `AddProduct3()` procedure is similar to `AddProduct2()`, except that it also returns a result set using a `SELECT` statement. This `SELECT` contains the `ProductName` and `UnitPrice` columns for the new row added to the `Products` table. This result set is returned in addition to the `ProductID` of the new row, which is returned using the `RETURN` statement. [Listing 8.14](#) shows the `AddProduct3.sql` script that creates the `AddProduct3()` procedure. You'll need to run this script before running the C# program.

### **Listing 8.14: ADDPRODUCT3.SQL**

---

```
/*
AddProduct3.sql creates a procedure that adds a row to the
Products table using values passed as parameters to the
procedure. The procedure returns the ProductID of the new row
using a RETURN statement and returns a result set containing
the new row
*/
CREATE PROCEDURE AddProduct3
    @MyProductName nvarchar(40),
    @MySupplierID int,
    @MyCategoryID int,
    @MyQuantityPerUnit nvarchar(20),
    @MyUnitPrice money,
    @MyUnitsInStock smallint,
    @MyUnitsOnOrder smallint,
    @MyReorderLevel smallint,
    @MyDiscontinued bit
AS
    - declare the @MyProductID variable
    DECLARE @MyProductID int

    - insert a row into the Products table
    INSERT INTO Products (
        ProductName, SupplierID, CategoryID, QuantityPerUnit,
        UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel,
        Discontinued
    )
```

```

) VALUES (
    @MyProductName, @MySupplierID, @MyCategoryID, @MyQuantityPerUnit,
    @MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder, @MyReorderLevel,
    @MyDiscontinued
)

- use the SCOPE_IDENTITY() function to get the last
- identity value inserted into a table performed within
- the current database session and stored procedure,
- so SCOPE_IDENTITY returns the ProductID for the new row
- in the Products table in this case
SET @MyProductID = SCOPE_IDENTITY()

- return the result set
SELECT ProductName, UnitPrice
FROM Products
WHERE ProductID = @MyProductID

- return @MyProductID
RETURN @MyProductID

```

---

Since you've already seen the basics for the code that execute the six steps shown earlier in this section, I'll go straight to the code with minimal explanation. [Listing 8.15](#) shows the program that calls `AddProduct3()`. The important things to notice are:

### **Listing 8.15: EXECUTEADDPRODUCT3.CS**

---

```

/*
ExecuteAddProduct3.cs illustrates how to call the SQL Server
AddProduct3() stored procedure
*/

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

class ExecuteAddProduct3
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        mySqlConnection.Open();

        // step 1: create a Command object and set its CommandText
        // property to an EXECUTE statement containing the stored
        // procedure call
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "EXECUTE @MyProductID = AddProduct3 @MyProductName, " +
            "@MySupplierID, @MyCategoryID, @MyQuantityPerUnit, " +
            "@MyUnitPrice, @MyUnitsInStock, @MyUnitsOnOrder, " +
            "@MyReorderLevel, @MyDiscontinued";

        // step 2: add the required parameters to the Command object
        mySqlCommand.Parameters.Add("@MyProductID", SqlDbType.Int);
        mySqlCommand.Parameters["@MyProductID"].Direction =
            ParameterDirection.Output;
    }
}

```

```

mySqlCommand.Parameters.Add(
    "@MyProductName", SqlDbType.NVarChar, 40).Value = "Widget";
mySqlCommand.Parameters.Add(
    "@MySupplierID", SqlDbType.Int).Value = 1;
mySqlCommand.Parameters.Add(
    "@MyCategoryID", SqlDbType.Int).Value = 1;
mySqlCommand.Parameters.Add(
    "@MyQuantityPerUnit", SqlDbType.NVarChar, 20).Value = "1 per box";
mySqlCommand.Parameters.Add(
    "@MyUnitPrice", SqlDbType.Money).Value = 5.99;
mySqlCommand.Parameters.Add(
    "@MyUnitsInStock", SqlDbType.SmallInt).Value = 10;
mySqlCommand.Parameters.Add(
    "@MyUnitsOnOrder", SqlDbType.SmallInt).Value = 5;
mySqlCommand.Parameters.Add(
    "@MyReorderLevel", SqlDbType.SmallInt).Value = 5;
mySqlCommand.Parameters.Add(
    "@MyDiscontinued", SqlDbType.Bit).Value = 1;

// step 3: execute the Command object using the ExecuteReader()
// method
SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();

// step 4: read the rows using the DataReader object
while (mySqlDataReader.Read())
{
    Console.WriteLine("mySqlDataReader[\"ProductName\"] = " +
        mySqlDataReader["ProductName"]);
    Console.WriteLine("mySqlDataReader[\"UnitPrice\"] = " +
        mySqlDataReader["UnitPrice"]);
}

// step 5: close the DataReader object
mySqlDataReader.Close();

// step 6: read the value of the output parameter
Console.WriteLine("New ProductID = " +
    mySqlCommand.Parameters["@MyProductID"].Value);

mySqlConnection.Close();
}

```

---

The `ExecuteReader()` method is used to return the result set containing the `ProductName` and `UnitPrice` columns for the new row.

The result set is then read using a `SqlDataReader` object.

The `SqlDataReader` object is closed before the output parameter is read.

The output from this program is as follows:

```

mySqlDataReader["ProductName"] = Widget
mySqlDataReader["UnitPrice"] = 5.99
New ProductID = 83

```

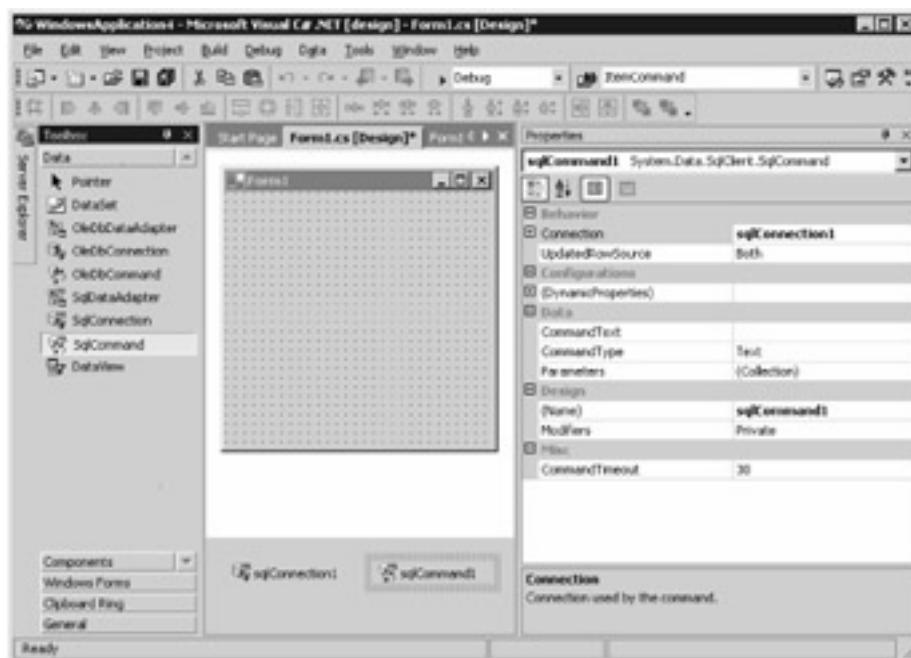
## Creating a *Command* Object Using Visual Studio .NET

To create a `SqlCommand` object using Visual Studio .NET (VS .NET), you drag a `SqlCommand` object from the Data tab of the Toolbox to your form. You can also drag an `OleDbCommand` object from the Data tab of the Toolbox to your form.

Before you perform the procedure explained in this section, do the following:

1. Create a new project named `MyDataReader` containing a Windows application.
2. Add a `SqlConnection` object to your project (refer back to the [previous chapter](#) to see how to add a `SqlConnection` object using VS .NET). This object will have the default name of `sqlConnection1`.
3. Configure your `sqlConnection1` object to access your Northwind database.

Drag a `SqlCommand` object to your form. [Figure 8.1](#) shows a form with a `SqlCommand` object. This object is assigned the default name of `sqlCommand1`.



**Figure 8.1:** A `SqlCommand` object in a form

You then set the `Connection` property for your `sqlCommand1` using the drop-down list to the right of the `Connection` property in the Properties window. You can select an existing `Connection` object from the drop-down list; you can also create a new `Connection` object by selecting `New` from the list. For this example, select your existing `sqlConnection1` object for the `Connection` property of your `sqlCommand1` object, as shown in [Figure 8.1](#).

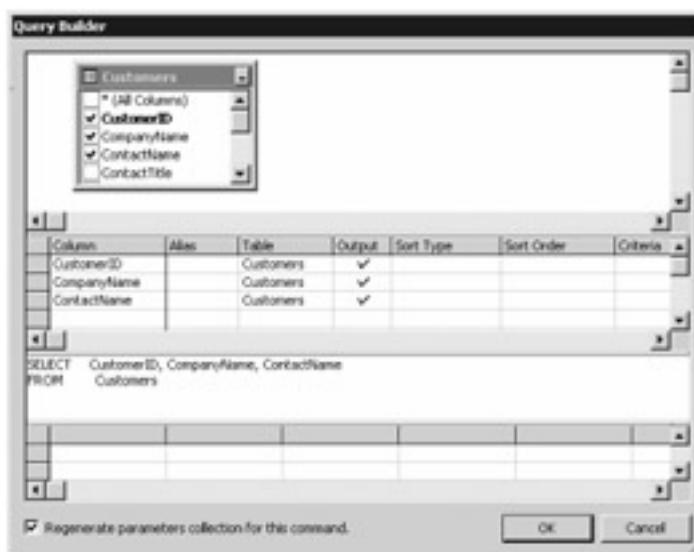
You can use Query Builder to create a SQL statement by clicking on the ellipsis button to the right of the `CommandText` property, and you can set parameters for a command by clicking the ellipsis button to the right of the `Parameters` property. You'll set the `CommandText` property of your `SqlCommand` object to a `SELECT` statement that retrieves the `CustomerID`, `CompanyName`, and `ContactName` columns from the `Customers` table. You'll construct this `SELECT` statement using Query Builder. To get started, click the ellipsis button to the right of the `CommandText` property for your `SqlCommand` object.

In the Add Table dialog, select the `Customers` table, as shown in [Figure 8.2](#). Click the `Add` button to add the `Customers` table to your query. Click the `Close` button to continue.



**Figure 8.2:** Adding the `Customers` table to the query using the Add Table dialog

Next, you construct your query using Query Builder. You select the columns you want to retrieve here. Add the `CustomerID`, `CompanyName`, and `ContactName` columns using Query Builder, as shown in [Figure 8.3](#).



**Figure 8.3:** Adding the `CustomerID`, `CompanyName`, and `ContactName` columns to the query using Query Builder

Click the OK button to continue. The `CommandText` property of your `SqlCommand` object is then set to the `SELECT` statement you created in Query Builder.

**Note** Save your `MyDataReader` project by selecting File > Save All. You'll see the use of the `SqlCommand` object you added to your project in the [next chapter](#).

## Summary

In this chapter, you learned how to execute database commands. There are three `Command` classes: `SqlCommand`, `OleDbCommand`, and `OdbcCommand`. You use a `Command` object to execute a SQL `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. You can also use a `Command` object to execute a stored procedure call, or retrieve all the rows and columns from a specific table; this is known as a `TableDirect` command. You use an object of the `SqlCommand` class to execute a command against a SQL Server database, an object of the `SqlDataReader` class to read the rows retrieved from a SQL Server database, and an object of the `SqlTransaction` class to represent a database transaction in a SQL Server database.

In the [next chapter](#), you'll learn the details of `DataReader` objects.

## Chapter 9: Using *DataReader* Objects to Read Results

### Overview

Reading rows using a `DataReader` object (sometimes known as a *firehose cursor*) is typically faster than reading from a `DataSet`. `DataReader` objects are part of the managed providers, and there are three `DataReader` classes: `SqlDataReader`, `OleDbDataReader`, and `OdbcDataReader`. You use a `DataReader` object to read rows retrieved from the database using a `Command` object.

`DataReader` objects can be used to read rows only in a forward direction. `DataReader` objects act as an alternative to a `DataSet` object (`DataSet` objects allow you to store a copy of the rows from the database, and you can work with that copy while disconnected from the database). You cannot use a `DataReader` to modify rows in the database.

Featured in this chapter:

- The `SqlDataReader` class
- Creating a `SqlDataReader` object
- Reading rows from a `SqlDataReader` object
- Returning strongly typed column values
- Reading null values
- Executing multiple SQL statements
- Using a `DataReader` object in Visual Studio .NET

## The *SqlDataReader* Class

You use an object of the *SqlDataReader* class to read rows retrieved from a SQL Server database, an object of the *OleDbDataReader* class to read rows from any database that supports OLE DB, such as Oracle or Access, and an object of the *OdbcDataReader* class to read rows from any database that supports ODBC. [Table 9.1](#) shows some of the *SqlDataReader* properties.

**Table 9.1: *SqlDataReader* PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
Depth	int	Gets a value indicating the depth of nesting for the current row.
FieldCount	int	Gets the number of columns in the current row.
IsClosed	bool	Gets a <code>bool</code> value indicating whether the data reader is closed.
RecordsAffected	int	Gets the number of rows added, modified, or removed by execution of the SQL statement.

**Note** Although the *SqlDataReader* class is specific to SQL Server, many of the properties and methods in this class are the same as those for the *OleDbDataReader* and *OdbcDataReader* classes.

[Table 9.2](#) shows some of the public *SqlDataReader* methods.

**Table 9.2: *SqlDataReader* METHODS**

METHOD	RETURN TYPE	DESCRIPTION
<code>GetBoolean()</code>	<code>bool</code>	Returns the value of the specified column as a <code>bool</code> .
<code>GetByte()</code>	<code>byte</code>	Returns the value of the specified column as a <code>byte</code> .
<code>GetBytes()</code>	<code>long</code>	Reads a stream of <code>byte</code> values from the specified column into a <code>byte</code> array. The <code>long</code> value returned is the number of <code>byte</code> values read from the column.
<code>GetChar()</code>	<code>char</code>	Returns the value of the specified column as a <code>char</code> .
<code>GetChars()</code>	<code>long</code>	Reads a stream of <code>char</code> values from the specified column into a <code>char</code> array. The <code>long</code> value returned is the number of <code>char</code> values read from the column.
<code>GetDataTypeName()</code>	<code>string</code>	Returns the name of the source data type for the specified column.
<code>GetDateTime()</code>	<code>DateTime</code>	Returns the value of the specified column as a <code>DateTime</code> .
<code>GetDecimal()</code>	<code>decimal</code>	Returns the value of the specified column as a <code>decimal</code> .
<code>GetDouble()</code>	<code>double</code>	Returns the value of the specified column as a <code>double</code> .
<code>GetFieldType()</code>	<code>Type</code>	Returns the <code>Type</code> of the specified column.
<code>GetFloat()</code>	<code>float</code>	Returns the value of the specified column as a <code>float</code> .
<code>GetGuid()</code>	<code>Guid</code>	Returns the value of the specified column as a globally unique identifier (GUID).
<code>GetInt16()</code>	<code>short</code>	Returns the value of the specified column as a <code>short</code> .
<code>GetInt32()</code>	<code>int</code>	Returns the value of the specified column as an <code>int</code> .

GetInt64()	long	Returns the value of the specified column as a long.
GetName()	string	Returns the name of the specified column.
GetOrdinal()	int	Returns the numeric position, or <i>ordinal</i> , of the specified column (first column has an ordinal of 0).
GetSchemaTable()	DataTable	Returns a DataTable that contains details of the columns stored in the data reader.
GetSqlBinary()	SqlBinary	Returns the value of the specified column as a SqlBinary object. The SqlBinary class is declared in the System.Data.SqlTypes namespace. All the GetSql* methods are specific to the SqlDataReader class.
GetSqlBoolean()	SqlBoolean	Returns the value of the specified column as a SqlBoolean object.
GetSqlByte()	SqlByte	Returns the value of the specified column as a SqlByte object.
GetSqlDateTime()	SqlDateTime	Returns the value of the specified column as a SqlDateTime object.
GetSqlDecimal()	SqlDecimal	Returns the value of the specified column as a SqlDecimal object.
GetSqlDouble()	SqlDouble	Returns the value of the specified column as a SqlDouble object.
GetSqlGuid()	SqlGuid	Returns the value of the specified column as a SqlGuid object.
GetSqlInt16()	SqlInt16	Returns the value of the specified column as a SqlInt16 object.
GetSqlInt32()	SqlInt32	Returns the value of the specified column as a SqlInt32 object.
GetSqlInt64()	SqlInt64	Returns the value of the specified column as a SqlInt64 object.
GetSqlMoney()	SqlMoney	Returns the value of the specified column as a SqlMoney object.
GetSqlSingle()	SqlSingle	Returns the value of the specified column as a SqlSingle object.
GetSqlString()	SqlString	Returns the value of the specified column as a SqlString object.
GetSqlValue()	object	Returns the value of the specified column as an object.
GetSqlValues()	int	Copies the value of all the columns in the current row into a specified object array. The int returned by this method is the number of elements in the array.
GetString()	string	Returns the value of the specified column as a string.
GetValue()	object	Returns the value of the specified column as an object.
GetValues()	int	Copies the value of all the columns in the current row into a specified object array. The int returned by this method is the number of elements in the array.
IsDBNull()	bool	Returns a bool that indicates whether the specified column contains a null value.
NextResult()	bool	Moves the data reader to the next row in the result set. The bool returned by this method indicates whether there are more rows in the result set.

Read( )	bool	Moves the data reader to the next row in the result set and reads the row. The bool returned by this method indicates whether there are more rows in the result set.
---------	------	--

**Note**

The *DataTable* column details include the name (stored in the *ColumnName* column of the returned *DataTable*), ordinal (stored in *ColumnOrdinal*), maximum length of the value that may be stored in the column (stored in *ColumnSize*), precision and scale of a numeric column (stored in *NumericPrecision* and *NumericScale*), among others. Precision is the total number of digits that make up a number, and scale is the total number of digits to the right of the decimal point. You saw how to read a schema using a program in the [previous chapter](#).

**Tip**

The *System.Data.SqlTypes* namespace provides classes for native data types used within SQL Server. These classes provide a safer and faster alternative to other data types returned by the other *Get\** methods. Using the classes in this namespace helps prevent type conversion errors caused by loss of precision. Because other data types are converted to and from *SqlTypes* behind the scenes, explicitly creating and using objects within this namespace results in faster code as well. You'll learn more about the *SqlTypes* namespace later in the "[Using the GetSql\\* Methods to Read Column Values](#)" section.

**Team LiB****◀ PREVIOUS****NEXT ▶**

## Creating a *SqlDataReader* Object

You can create a `DataReader` object only by calling the `ExecuteReader()` method of a `Command` object. `Command` objects were covered in the [previous chapter](#). For example, the following code creates the required objects and executes a `SELECT` statement that retrieves the top five rows from the `Products` table of the SQL Server Northwind database, storing the returned rows in a `SqlDataReader` object:

```
SqlConnection mySqlConnection =
    new SqlConnection(
        "server=localhost;database=Northwind;uid=sa;pwd=sa"
    );
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 5 ProductID, ProductName, UnitPrice, " +
    "UnitsInStock, Discontinued " +
    "FROM Products " +
    "ORDER BY ProductID";
mySqlConnection.Open();
SqlDataReader productsSqlDataReader =
    mySqlCommand.ExecuteReader();
```

Notice that the `SqlDataReader` object returned by the `ExecuteReader()` method is stored in the `productsSqlDataReader` object. You'll see how to use `productsSqlDataReader` in the following section.

## Reading Rows from a *SqlDataReader* Object

You read the rows from a `DataReader` object using the `Read()` method. This method returns the Boolean `true` value when there is another row to read, otherwise it returns `false`.

You can read an individual column value in a row from a `DataReader` by passing the name of the column in square brackets. For example, to read the `CustomerID` column, you use `productsSqlDataReader[ "ProductID" ]`. You can also specify the column you want to get by passing a numeric value in brackets. For example, `productsSqlDataReader[ 0 ]` also returns the `ProductID` column value.

**Tip** The difference between these two ways of reading a column value is performance: using numeric column positions instead of column names results in faster execution of the code.

Let's take a look at two code snippets that illustrate these two ways of reading column values. The first code snippet uses the column names to read the column values:

```
while (productsSqlDataReader.Read())
{
    Console.WriteLine(productsSqlDataReader[ "ProductID" ]);
    Console.WriteLine(productsSqlDataReader[ "ProductName" ]);
    Console.WriteLine(productsSqlDataReader[ "UnitPrice" ]);
    Console.WriteLine(productsSqlDataReader[ "Discontinued" ]);
}
```

The second code snippet uses the numeric column positions to read the column values:

```
while (productsSqlDataReader.Read())
{
    Console.WriteLine(productsSqlDataReader[ 0 ]);
    Console.WriteLine(productsSqlDataReader[ 1 ]);
    Console.WriteLine(productsSqlDataReader[ 2 ]);
    Console.WriteLine(productsSqlDataReader[ 3 ]);
}
```

Although the second code snippet is faster, it is less flexible since you have to hard-code the numeric column positions. If the column positions in the `SELECT` statement are changed, you need to change the hard-coded column positions in the code-and this is a maintenance nightmare. Also, hard-coding the column positions makes your programs more difficult to read.

There is a solution to this problem: you can call the `GetOrdinal()` method of your `DataReader` object. The `GetOrdinal()` method returns the position of a column given its name; this position is known as the column's *ordinal*. You can then use the position returned by `GetOrdinal()` to get the column values from your `DataReader`.

Let's take a look at some code that uses the `GetOrdinal()` method to obtain the positions of the columns from the example `SELECT` statement:

```
int productIDColPos =
    productsSqlDataReader.GetOrdinal("ProductID");
int productNameColPos =
    productsSqlDataReader.GetOrdinal("ProductName");
int unitPriceColPos =
    productsSqlDataReader.GetOrdinal("UnitPrice");
int discontinuedColPos =
    productsSqlDataReader.GetOrdinal("Discontinued");
```

You can then use these `int` values to get the column values from `productsSqlDataReader`:

```
while (productsSqlDataReader.Read())
{
    Console.WriteLine(productsSqlDataReader[productIDColPos]);
```

```

Console.WriteLine(productsSqlDataReader[productNameColPos]);
Console.WriteLine(productsSqlDataReader[unitPriceColPos]);
Console.WriteLine(productsSqlDataReader[discontinuedColPos]);
}

```

This way gives you the best of both worlds: high performance and flexibility.

**Warning** When you've finished reading the rows from your *DataReader* object, close it using the *Close()* method. The reason for this is that a *DataReader* object ties up the *Connection* object, and no other commands can be executed while there is an open *DataReader* for that *Connection*.

The following example closes *productsSqlDataReader* using the *Close()* method:

```
productsSqlDataReader.Close();
```

Once you've closed your *DataReader*, you can execute other commands using your *Connection* object.

[Listing 9.1](#) uses the code examples shown in this section.

## Listing 9.1: USINGCOLUMNORDINALS.CS

---

```

/*
 UsingColumnOrdinals.cs illustrates how to use the GetOrdinal()
 method of a DataReader object to get the numeric positions of
 a column
 */

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

class UsingColumnOrdinals
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        mySqlCommand.CommandText =
            "SELECT TOP 5 ProductID, ProductName, UnitPrice, " +
            "UnitsInStock, Discontinued " +
            "FROM Products " +
            "ORDER BY ProductID";

        mySqlConnection.Open();

        SqlDataReader productsSqlDataReader =
            mySqlCommand.ExecuteReader();

        // use the GetOrdinal() method of the DataReader object
        // to obtain the numeric positions of the columns
        int productIDColPos =
            productsSqlDataReader.GetOrdinal("ProductID");
        int productNameColPos =
            productsSqlDataReader.GetOrdinal("ProductName");
        int unitPriceColPos =
            productsSqlDataReader.GetOrdinal("UnitPrice");
    }
}

```

## Reading Rows from a SqlDataReader Object

```
int unitsInStockColPos =
    productsSqlDataReader.GetOrdinal("UnitsInStock");
int discontinuedColPos =
    productsSqlDataReader.GetOrdinal("Discontinued");

while (productsSqlDataReader.Read())
{
    Console.WriteLine("ProductID = " +
        productsSqlDataReader[productIDColPos]);
    Console.WriteLine("ProductName = " +
        productsSqlDataReader[productNameColPos]);
    Console.WriteLine("UnitPrice = " +
        productsSqlDataReader[unitPriceColPos]);
    Console.WriteLine("UnitsInStock = " +
        productsSqlDataReader[unitsInStockColPos]);
    Console.WriteLine("Discontinued = " +
        productsSqlDataReader[discontinuedColPos]);
}

productsSqlDataReader.Close();
mySqlConnection.Close();
}
```

---

The output from this program is as follows:

```
ProductID = 1
ProductName = Chai
UnitPrice = 18
UnitsInStock = 39
Discontinued = False
ProductID = 2
ProductName = Chang
UnitPrice = 19
UnitsInStock = 17
Discontinued = False
ProductID = 3
ProductName = Aniseed Syrup
UnitPrice = 10
UnitsInStock = 13
Discontinued = False
ProductID = 4
ProductName = Chef Anton's Cajun Seasoning
UnitPrice = 22
UnitsInStock = 53
Discontinued = False
ProductID = 5
ProductName = Chef Anton's Gumbo Mix
UnitPrice = 21.35
UnitsInStock = 0
Discontinued = True
```

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Returning Strongly Typed Column Values

Up to this point, you've retrieved column values from a `DataReader` only as generic objects of the `System.Object` class (such objects are often referred to as being of the C# `object` type).

**Note** All classes in C# are derived from the `System.Object` class.

I'll rewrite the `while` loop shown in the example in the [previous section](#) to show how you store column values as objects of the `System.Object` class:

```
while (productsSqlDataReader.Read())
{
    object productID =
        productsSqlDataReader[productIDColPos];
    object productName =
        productsSqlDataReader[productNameColPos];
    object unitPrice =
        productsSqlDataReader[unitPriceColPos];
    object unitsInStock =
        productsSqlDataReader[unitsInStockColPos];
    object discontinued =
        productsSqlDataReader[discontinuedColPos];

    Console.WriteLine("productID = " + productID);
    Console.WriteLine("productName = " + productName);
    Console.WriteLine("unitPrice = " + unitPrice);
    Console.WriteLine("unitsInStock = " + unitsInStock);
    Console.WriteLine("discontinued = " + discontinued);
}
```

This code results in the same output as [Listing 9.1](#). All I did in [Listing 9.1](#) is to explicitly show that a `DataReader` returns a column value as an object of the `System.Object` class by default. When an object of the `System.Object` class is displayed by the `Console.WriteLine()` method, the object is first implicitly converted to a string and then displayed.

That's fine for just displaying the column values, but what if you want to perform some kind of calculation with a value? To do that, you must first cast the value to a specific type. The following example casts the `unitPrice` object to a `decimal` and then multiplies it by 1.2:

```
decimal newUnitPrice = (decimal) unitPrice * 1.2m;
```

**Note** You add an `m` to the end of a literal number to indicate it is of the `decimal` type.

Casting an object to a specific type works, but it's not very elegant. It also goes against the one of the main benefits of a modern programming language: use of strong typing. Strongly typing means that you pick the type of a variable or object when declaring it. The main benefit of strong typing is that you're less likely to have runtime errors in your programs that are caused by using the wrong type. This is because the compiler checks your code to make sure the context of the type is correct.

The bottom line is that you should endeavor to make all your variables and objects of the appropriate type to begin with, and use casting only when you have no other choice. In this case, you have a choice: instead of casting, you can use one of the `DataReader` object's `Get*` methods to return a column value in an appropriate type.

**Note** I use the asterisk in `Get*` to indicate there are many methods that start with `Get`. You can see all the `Get*` methods in [Table 9.2](#), shown earlier.

For example, one of the `Get*` methods is `GetInt32()`, which returns a column value as an `int` value. The following code shows the use of the `GetInt32()` method to obtain the column value for the `ProductID` column as an `int`:

```
int productID =
    productsSqlDataReader.GetInt32(productIDColPos);
```

As you can see, you pass the ordinal of the column that has the value you want to obtain to the `Get*` method. You saw how to get a column's ordinal value in the [previous section](#).

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Using the *Get\** Methods to Read Column Values

Before I show you the other `Get *` methods that read column values, you need to know the standard C# types and the values they support. You need to know these so that you can understand the type compatibilities between C# and SQL Server shown later. [Table 9.3](#) shows the standard C# types, along with the underlying .NET type and the values that can be stored in the C# type.

**Table 9.3: STANDARD C# AND .NET TYPES**

C# TYPE	.NET TYPE	VALUES
<code>bool</code>	<code>Boolean</code>	A Boolean <code>true</code> or <code>false</code> value.
<code>byte</code>	<code>Byte</code>	An 8-bit unsigned integer between 0 and $2^8 - 1$ (255).
<code>char</code>	<code>Char</code>	A 16-bit Unicode character.
<code>DateTime</code>	<code>DateTime</code>	A date and time between 12:00:00 AM January 1, 0001 and 11:59:59 PM December 31, 9999.
<code>decimal</code>	<code>Decimal</code>	A fixed precision and scale number between approximately $+/-1.0 *10^{-28}$ and approximately $+/-7.9 *10^{28}$ with 28 significant figures of precision.
<code>double</code>	<code>Double</code>	A 64-bit floating-point number between approximately $+/-5 *10^{-324}$ and approximately $+/-1.7 *10^{308}$ with 15 to 16 significant figures of precision.
<code>float</code>	<code>Single</code>	A 32-bit floating-point number between approximately $+/-1.5 *10^{-45}$ to approximately $+/-3.4 *10^{38}$ with 7 significant figures of precision.
<code>Guid</code>	<code>Guid</code>	A 128-bit unsigned integer value (16 bytes) that is unique across all computers and networks.
<code>int</code>	<code>Int32</code>	A 32-bit signed integer between $-2^{31}$ (-2,147,483,648) and $2^{31} - 1$ (2,147,483,647).
<code>long</code>	<code>Int64</code>	A 64-bit signed integer between $-2^{63}$ (-9,223,372,036,854,775,808) and $2^{63} - 1$ (9,223,372,036,854,775,807).
<code>sbyte</code>	<code>SByte</code>	An 8-bit signed integer between $-2^7$ (-128) and $2^7 - 1$ (127).
<code>short</code>	<code>Int16</code>	A 16-bit signed integer between $-2^{15}$ (-32,768) and $2^{15} - 1$ (32,767).
<code>string</code>	<code>String</code>	A variable-length string of 16-bit Unicode characters.
<code>uint</code>	<code>UInt32</code>	A 32-bit unsigned integer between 0 and $2^{32} - 1$ (4,294,967,295).
<code>ulong</code>	<code>UInt64</code>	A 64-bit unsigned integer between 0 and $2^{64} - 1$ (18,446,744,073,709,551,615).
<code>ushort</code>	<code>UInt16</code>	A 16-bit unsigned integer between 0 and $2^{16} - 1$ (65,535).

**Note** The standard C# types are defined in the `System` namespace.

[Table 9.4](#) shows the SQL Server types, the compatible standard C# types, and the `DataReader` `Get*` methods that return each C# type. You use this table to figure out which method to call to get a specific column type. For example, if you need to get the value of a `bign` column, you call the `GetInt64()` method that returns a `long`.

**Table 9.4: SQL SERVER TYPES, COMPATIBLE STANDARD C# TYPES, AND GET\* METHODS**

SQL SERVER TYPE	COMPATIBLE STANDARD C# TYPE	GET* METHOD
<code>binary</code>	<code>byte[]</code>	<code>GetBytes()</code>
<code>bign</code>	<code>long</code>	<code>GetInt64()</code>
<code>bit</code>	<code>bool</code>	<code>GetBoolean()</code>

char	string	GetString()
datetime	DateTime	GetDateTime()
decimal	decimal	GetDecimal()
float	double	GetDouble()
image	byte[]	GetBytes()
int	int	GetInt32()
money	decimal	GetDecimal()
nchar	string	GetString()
ntext	string	GetString()
nvarchar	string	GetString()
numeric	decimal	GetDecimal()
real	float	GetFloat()
smalldatetime	DateTime	GetDateTime()
smallint	short	GetInt16()
smallmoney	decimal	GetDecimal()
sql_varient	object	GetValue()
text	string	GetString()
timestamp	byte[]	GetBytes()
tinyint	byte	GetByte()
varbinary	byte[]	GetBytes()
varchar	string	GetString()
uniqueidentifier	Guid	GetGuid()

**Note** You can see the SQL Server types and the values supported by those types in [Table 2.3](#) of [Chapter 2](#), "Introduction to Databases."

**Note** The `Get*` methods are defined in all of the `DataReader` classes and work for all databases.

Next you'll see how to use some of the methods shown in [Table 9.4](#).

## An Example of Using the *Get\** Methods

Let's take a look at an example that reads the `ProductID`, `ProductName`, `UnitPrice`, `UnitsInStock`, and `Discontinued` columns from the `Products` table using the `Get*` methods.

To figure out which `Get*` method to use to retrieve a particular SQL Server column type, you use [Table 9.4](#), shown earlier. For example, the `ProductID` column is a SQL Server `int`, and looking up that SQL Server type in [Table 9.4](#), you can see you use the `GetInt32()` method to obtain the column value as a C# `int`. [Table 9.5](#) summarizes the column names, SQL Server types, `Get*` methods, and C# return types required to retrieve the five columns from the `Products` table.

**Table 9.5: Products TABLE COLUMNS, TYPES, AND METHODS**

COLUMN NAME	SQL SERVER COLUMN TYPE	GET* METHOD	C# RETURN TYPE
<code>ProductID</code>	<code>int</code>	<code>GetInt32()</code>	<code>int</code>
<code>ProductName</code>	<code>nvarchar</code>	<code>GetString()</code>	<code>string</code>
<code>UnitPrice</code>	<code>money</code>	<code>GetDecimal()</code>	<code>decimal</code>
<code>UnitsInStock</code>	<code>smallint</code>	<code>GetInt16()</code>	<code>short</code>
<code>Discontinued</code>	<code>bit</code>	<code>GetBoolean()</code>	<code>bool</code>

Let's assume that you already have a `SqlDataReader` object named `productsSqlDataReader` and that it may be used to read the five columns from the `Products` table. The following `while` loop uses the `Get*` methods and returned C# types shown in [Table 9.5](#) to obtain the column values from `productsSqlDataReader`:

```
while (productsSqlDataReader.Read())
{
    int productID =
        productsSqlDataReader.GetInt32(productIDColPos);
    Console.WriteLine("productID = " + productID);

    string productName =
        productsSqlDataReader.GetString(productNameColPos);
    Console.WriteLine("productName = " + productName);

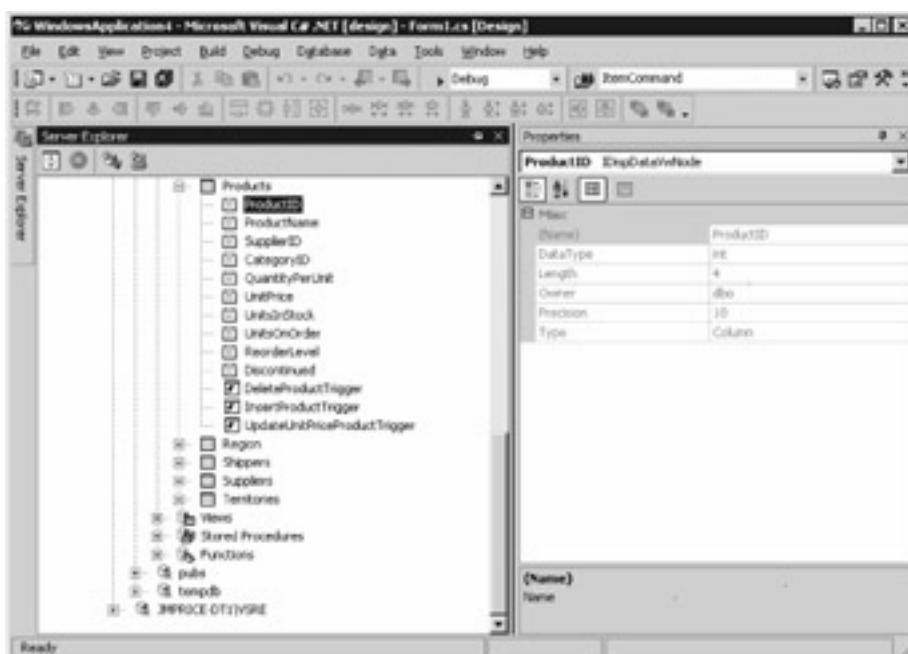
    decimal unitPrice =
        productsSqlDataReader.GetDecimal(unitPriceColPos);
    Console.WriteLine("unitPrice = " + unitPrice);

    short unitsInStock =
        productsSqlDataReader.GetInt16(unitsInStockColPos);
    Console.WriteLine("unitsInStock = " + unitsInStock);

    bool discontinued =
        productsSqlDataReader.GetBoolean(discontinuedColPos);
    Console.WriteLine("discontinued = " + discontinued);
}
```

As you can see, five variables of the appropriate type are created in this `while` loop, each of which is used to store the result from the `Get*` method. For example, the `productID` variable is used to store the `ProductID` column value, and since `ProductID` is of the SQL Server `int` type, the appropriate C# type for the `productID` variable is `int`. To get the `ProductID` column value as a C# `int`, you call the `GetInt32()` method. Similarly, the `productName` variable is a C# `string` that is used to store the `ProductName` column value. This column is of the `nvarchar` SQL Server type, and to get the `ProductName` column value, the `GetString()` method is used.

Of course, this code depends on your knowing the type of the database column. If you don't know the type of a column, you can get it using Visual Studio .NET's Server Explorer. For example, [Figure 9.1](#) shows the details of the ProductID column of the Products table. As you can see, ProductID is an int.



**Figure 9.1:** Obtaining the type of a column using Visual Studio .NET's Server Explorer

Before closing this section, I will show you how to get the .NET type and database type of a column using C#. You get the .NET type used to represent a column using the `GetFieldType()` method of your `DataReader` object. For example:

```
Console.WriteLine("ProductID .NET type = " +
    productsSqlDataReader.GetFieldType(productIdColPos));
```

This example displays:

```
ProductID .NET type = System.Int32
```

As you can see, the `System.Int32` .NET type is used to represent the `ProductID` column. The `System.Int32` .NET type corresponds to the C# `int` type. You can see this type correspondence in [Table 9.3](#), shown earlier.

You can get the database type for a column using the `GetDataTypeName()` method of your `DataReader` object. For example:

```
Console.WriteLine("ProductID database type = " +
    productsSqlDataReader.GetDataTypeName(productIdColPos));
```

This example displays:

```
ProductID database type = int
```

As you can see, the `ProductID` column is of the SQL Server `int` type.

[Listing 9.2](#) uses the code examples shown in this section.

## **Listing 9.2: STRONGLYTYPEDCOLUMNVALUES.CS**

```
/*
StronglyTypedColumnValues.cs illustrates how to read
column values as C# types using the Get* methods
*/
using System;
using System.Data;
using System.Data.SqlClient;
```

```

class StronglyTypedColumnValues
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        mySqlCommand.CommandText =
            "SELECT TOP 5 ProductID, ProductName, UnitPrice, " +
            "UnitsInStock, Discontinued " +
            "FROM Products " +
            "ORDER BY ProductID";

        mySqlConnection.Open();

        SqlDataReader productsSqlDataReader =
            mySqlCommand.ExecuteReader();

        int productIDColPos =
            productsSqlDataReader.GetOrdinal("ProductID");
        int productNameColPos =
            productsSqlDataReader.GetOrdinal("ProductName");
        int unitPriceColPos =
            productsSqlDataReader.GetOrdinal("UnitPrice");
        int unitsInStockColPos =
            productsSqlDataReader.GetOrdinal("UnitsInStock");
        int discontinuedColPos =
            productsSqlDataReader.GetOrdinal("Discontinued");

        // use the GetFieldType() method of the DataReader object
        // to obtain the .NET type of a column
        Console.WriteLine("ProductID .NET type = " +
            productsSqlDataReader.GetFieldType(productIDColPos));
        Console.WriteLine("ProductName .NET type = " +
            productsSqlDataReader.GetFieldType(productNameColPos));
        Console.WriteLine("UnitPrice .NET type = " +
            productsSqlDataReader.GetFieldType(unitPriceColPos));
        Console.WriteLine("UnitsInStock .NET type = " +
            productsSqlDataReader.GetFieldType(unitsInStockColPos));
        Console.WriteLine("Discontinued .NET type = " +
            productsSqlDataReader.GetFieldType(discontinuedColPos));

        // use the GetDataTypeName() method of the DataReader object
        // to obtain the database type of a column
        Console.WriteLine("ProductID database type = " +
            productsSqlDataReader.GetDataTypeName(productIDColPos));
        Console.WriteLine("ProductName database type = " +
            productsSqlDataReader.GetDataTypeName(productNameColPos));
        Console.WriteLine("UnitPrice database type = " +
            productsSqlDataReader.GetDataTypeName(unitPriceColPos));
        Console.WriteLine("UnitsInStock database type = " +
            productsSqlDataReader.GetDataTypeName(unitsInStockColPos));
        Console.WriteLine("Discontinued database type = " +
            productsSqlDataReader.GetDataTypeName(discontinuedColPos));
    }
}

```

## An Example of Using the Get\* Methods

```
// read the column values using Get* methods that
// return specific C# types
while (productsSqlDataReader.Read())
{
    int productID =
        productsSqlDataReader.GetInt32(productIDColPos);
    Console.WriteLine("productID = " + productID);

    string productName =
        productsSqlDataReader.GetString(productNameColPos);
    Console.WriteLine("productName = " + productName);

    decimal unitPrice =
        productsSqlDataReader.GetDecimal(unitPriceColPos);
    Console.WriteLine("unitPrice = " + unitPrice);

    short unitsInStock =
        productsSqlDataReader.GetInt16(unitsInStockColPos);
    Console.WriteLine("unitsInStock = " + unitsInStock);

    bool discontinued =
        productsSqlDataReader.GetBoolean(discontinuedColPos);
    Console.WriteLine("discontinued = " + discontinued);
}

productsSqlDataReader.Close();
mySqlConnection.Close();
}
```

---

The output from this program is as follows:

```
ProductID .NET type = System.Int32
ProductName .NET type = System.String
UnitPrice .NET type = System.Decimal
UnitsInStock .NET type = System.Int16
Discontinued .NET type = System.Boolean
ProductID database type = int
ProductName database type = nvarchar
UnitPrice database type = money
UnitsInStock database type = smallint
Discontinued database type = bit
productID = 1
productName = Chai
unitPrice = 18
unitsInStock = 39
discontinued = False
productID = 2
productName = Chang
unitPrice = 19
unitsInStock = 17
discontinued = False
productID = 3
productName = Aniseed Syrup
unitPrice = 10
unitsInStock = 13
discontinued = False
productID = 4
productName = Chef Anton's Cajun Seasoning
```

```

unitPrice = 22
unitsInStock = 53
discontinued = False
productID = 5
productName = Chef Anton's Gumbo Mix
unitPrice = 21.35
unitsInStock = 0
discontinued = True

```

## Using the *GetSql\** Methods to Read Column Values

In addition to using the *Get\** methods to read column values as standard C# types, if you are using SQL Server, you can also use the *GetSql\** methods. The *GetSql\** methods return values as *Sql\** types, which correspond to the actual types used by SQL Server in the database.

**Note** You can see all the *GetSql\** methods in [Table 9.2](#), shown earlier.

The *GetSql\** methods and *Sql\** types are defined in the `System.Data.SqlTypes` namespace, and they are specific to SQL Server. In addition, the *GetSql\** methods are specific to the `SqlDataReader` class. Using the *GetSql\** methods and *Sql\** types helps prevent type conversion errors caused by loss of precision in numeric values.

The *GetSql\** methods are also faster than their *Get\** counterparts. This is because the *GetSql\** methods don't need to convert between SQL Server types and the standard C# types, which the *Get\** methods have to do.

**Tip** If you are using SQL Server, always use the *GetSql\** methods and *Sql\** types rather than the *Get\** methods and the standard C# types. I showed you the *Get\** methods earlier only because they work with non-SQL Server databases.

[Table 9.6](#) shows the *Sql\** types and the values that may be stored in those types.

**Table 9.6: *Sql\** TYPES**

Sq1* TYPE	VALUES
<code>SqlBinary</code>	A variable-length string of binary data.
<code>SqlBoolean</code>	An integer with either a 1 or 0 value.
<code>SqlByte</code>	An 8-bit unsigned integer value between 0 and $2^8 - 1$ (255).
<code>SqlDateTime</code>	A date and time between 12:00:00 AM January 1, 1753 and 11:59:59 PM December 31, 9999. This is accurate to 3.33 milliseconds.
<code>SqlDecimal</code>	Fixed precision and scale numeric value between $-10^{38} + 1$ and $10^{38} - 1$ .
<code>SqlDouble</code>	A 64-bit floating-point number between -1.79769313486232E308 and 1.79769313486232E308 with 15 significant figures of precision.
<code>SqlGuid</code>	A 128-bit integer value (16 bytes) that is unique across all computers and networks.
<code>SqlInt16</code>	A 16-bit signed integer between $-2^{15}$ (-32,768) and $2^{15} - 1$ (32,767).
<code>SqlInt32</code>	A 32-bit signed integer between $-2^{31}$ (-2,147,483,648) and $2^{31} - 1$ (2,147,483,647).
<code>SqlInt64</code>	A 64-bit signed integer between $-2^{63}$ (-9,223,372,036,854,775,808) and $2^{63} - 1$ (9,223,372,036,854,775,807).
<code>SqlMoney</code>	A currency value between -922,337,203,685,477.5808 and 922,337,203,685,477.5807. This is accurate to 1/10,000th of a currency unit.
<code>SqlSingle</code>	A 32-bit floating-point number between -3.402823E38 and 3.402823E38 with seven significant figures of precision.
<code>SqlString</code>	A variable-length string of characters.

[Table 9.7](#) shows the SQL server types, the corresponding `Sql*` types, and the `GetSql*` methods used to read a column as the `Sql*` type.

**Table 9.7: SQL SERVER TYPES, COMPATIBLE `Sql*` TYPES, AND `GetSql*` METHODS**

SQL SERVER TYPE	Sql* TYPE	GetSql* METHOD
<code>bigint</code>	<code>SqlInt64</code>	<code>GetSqlInt64()</code>
<code>int</code>	<code>SqlInt32</code>	<code>GetSqlInt32()</code>
<code>smallint</code>	<code>SqlInt16</code>	<code>GetSqlInt16()</code>
<code>tinyint</code>	<code>SqlByte</code>	<code>GetSqlByte()</code>
<code>bit</code>	<code>SqlBoolean</code>	<code>GetSqlBoolean()</code>
<code>decimal</code>	<code>SqlDecimal</code>	<code>GetSqlDecimal()</code>
<code>numeric</code>	<code>SqlDecimal</code>	<code>GetSqlDecimal()</code>
<code>money</code>	<code>SqlMoney</code>	<code>GetSqlMoney()</code>
<code>smallmoney</code>	<code>SqlMoney</code>	<code>GetSqlMoney()</code>
<code>float</code>	<code>SqlDouble</code>	<code>GetSqlDouble()</code>
<code>real</code>	<code>SqlSingle</code>	<code>GetSqlSingle()</code>
<code>datetime</code>	<code>SqlDateTime</code>	<code>GetSqlDateTime()</code>
<code>smalldatetime</code>	<code>SqlDateTime</code>	<code>GetSqlDateTime()</code>
<code>char</code>	<code>SqlString</code>	<code>GetSqlString()</code>
<code>varchar</code>	<code>SqlString</code>	<code>GetSqlString()</code>
<code>text</code>	<code>SqlString</code>	<code>GetSqlString()</code>
<code>nchar</code>	<code>SqlString</code>	<code>GetSqlString()</code>
<code>nvarchar</code>	<code>SqlString</code>	<code>GetSqlString()</code>
<code>ntext</code>	<code>SqlString</code>	<code>GetSqlString()</code>
<code>binary</code>	<code>SqlBinary</code>	<code>GetSqlBinary()</code>
<code>varbinary</code>	<code>SqlBinary</code>	<code>GetSqlBinary()</code>
<code>image</code>	<code>SqlBinary</code>	<code>GetSqlBinary()</code>
<code>sql_varient</code>	<code>object</code>	<code>GetSqlValue()</code>
<code>timestamp</code>	<code>SqlBinary</code>	<code>GetSqlBinary()</code>
<code>uniqueidentifier</code>	<code>SqlGuid</code>	<code>GetSqlGuid()</code>

Next you'll see how to use some of the methods shown in [Table 9.7](#).

## An Example of Using the `GetSql*` Methods

Let's take a look at an example that reads the `ProductID`, `ProductName`, `UnitPrice`, `UnitsInStock`, and `Discontinued` columns from the `Products` table using the `GetSql*` methods.

To figure out which `GetSql*` method to use to retrieve a particular column type, you use [Table 9.7](#), shown earlier. For example, the `ProductID` column is a SQL Server `int`, and looking up that type in [Table 9.7](#), you can see you use the `GetSqlInt32()` method

to obtain the column value as a C# `SqlInt32`. [Table 9.8](#) summarizes the column names, SQL Server types, `GetSql*` methods, and `Sql*` return types for the columns retrieved from the `Products` table.

**Table 9.8: Products TABLE COLUMNS, TYPES, AND `GetSql*` METHODS**

COLUMN NAME	SQL SERVER COLUMN TYPE	GETSQL* METHOD	SQL* RETURN TYPE
ProductID	int	<code>GetInt32()</code>	<code>SqlInt32</code>
ProductName	nvarchar	<code>GetString()</code>	<code>SqlString</code>
UnitPrice	money	<code>GetSqlMoney()</code>	<code>SqlMoney</code>
UnitsInStock	smallint	<code>GetSqlInt16()</code>	<code>SqlInt16</code>
Discontinued	bit	<code>GetSqlBoolean()</code>	<code>SqlBoolean</code>

Let's assume that you already have a `SqlDataReader` object named `productsSqlDataReader` and it may be used to read the columns from the `Products` table. The following `while` loop uses the `GetSql*` methods and returned `Sql*` types shown earlier in [Table 9.8](#) to obtain the column values from `productsSqlDataReader`:

```
while (productsSqlDataReader.Read())
{
    SqlInt32 productID =
        productsSqlDataReader.GetInt32(productIDColPos);
    Console.WriteLine("productID = " + productID);

    SqlString productName =
        productsSqlDataReader.GetString(productNameColPos);
    Console.WriteLine("productName = " + productName);

    SqlMoney unitPrice =
        productsSqlDataReader.GetSqlMoney(unitPriceColPos);
    Console.WriteLine("unitPrice = " + unitPrice);

    SqlInt16 unitsInStock =
        productsSqlDataReader.GetInt16(unitsInStockColPos);
    Console.WriteLine("unitsInStock = " + unitsInStock);

    SqlBoolean discontinued =
        productsSqlDataReader.GetSqlBoolean(discontinuedColPos);
    Console.WriteLine("discontinued = " + discontinued);
}
```

[Listing 9.3](#) uses this `while` loop.

**Listing 9.3: STRONGLYTYPEDCOLUMNVALUESSQL.CS**


---

```
/*
 StronglyTypedColumnValuesSql.cs illustrates how to read
 column values as Sql* types using the GetSql* methods
 */

using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;

class StronglyTypedColumnValuesSql
{
```

```
public static void Main()
{
    SqlConnection mySqlConnection =
        new SqlConnection(
            "server=localhost;database=Northwind;uid=sa;pwd=sa"
        );

    SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

    mySqlCommand.CommandText =
        "SELECT TOP 5 ProductID, ProductName, UnitPrice, " +
        "UnitsInStock, Discontinued " +
        "FROM Products " +
        "ORDER BY ProductID";

    mySqlConnection.Open();

    SqlDataReader productsSqlDataReader =
        mySqlCommand.ExecuteReader();

    int productIDColPos =
        productsSqlDataReader.GetOrdinal("ProductID");
    int productNameColPos =
        productsSqlDataReader.GetOrdinal("ProductName");
    int unitPriceColPos =
        productsSqlDataReader.GetOrdinal("UnitPrice");
    int unitsInStockColPos =
        productsSqlDataReader.GetOrdinal("UnitsInStock");
    int discontinuedColPos =
        productsSqlDataReader.GetOrdinal("Discontinued");

    // read the column values using GetSql* methods that
    // return specific Sql* types
    while (productsSqlDataReader.Read())
    {
        SqlInt32 productID =
            productsSqlDataReader.GetSqlInt32(productIDColPos);
        Console.WriteLine("productID = " + productID);

        SqlString productName =
            productsSqlDataReader.GetSqlString(productNameColPos);
        Console.WriteLine("productName = " + productName);

        SqlMoney unitPrice =
            productsSqlDataReader.GetSqlMoney(unitPriceColPos);
        Console.WriteLine("unitPrice = " + unitPrice);

        SqlInt16 unitsInStock =
            productsSqlDataReader.GetSqlInt16(unitsInStockColPos);
        Console.WriteLine("unitsInStock = " + unitsInStock);

        SqlBoolean discontinued =
            productsSqlDataReader.GetSqlBoolean(discontinuedColPos);
        Console.WriteLine("discontinued = " + discontinued);
    }

    productsSqlDataReader.Close();
    mySqlConnection.Close();
}
```

The output from this program is as follows:

```
productID = 1
productName = Chai
unitPrice = 18
unitsInStock = 39
discontinued = False
productID = 2
productName = Chang
unitPrice = 19
unitsInStock = 17
discontinued = False
productID = 3
productName = Aniseed Syrup
unitPrice = 10
unitsInStock = 13
discontinued = False
productID = 4
productName = Chef Anton's Cajun Seasoning
unitPrice = 22
unitsInStock = 53
discontinued = False
productID = 5
productName = Chef Anton's Gumbo Mix
unitPrice = 21.35
unitsInStock = 0
discontinued = True
```

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Reading Null Values

As you learned in [Chapter 2](#), a column defined as null can store a null value. A null value indicates that the column value is unknown. A standard C# type cannot store a null value, but a `Sql*` type can.

Let's consider an example of reading a null value from the database. Say you've performed a `SELECT` statement that retrieves the `UnitPrice` column for a row from the `Products` table, and that the `UnitPrice` column contains a null value. If you try to store that null value in a standard C# type (such as a `decimal`) using the following code:

```
decimal unitPrice =
    productsSqlDataReader.GetDecimal(unitPriceColPos);
```

then you'll get the following exception:

```
System.Data.SqlTypes.SqlNullValueException
```

You'll also get this exception if you try to store the null value in an `object`, as shown in the following example:

```
object unitPriceObj =
    productsSqlDataReader["UnitPrice"];
```

You can check if a column contains a null value using the `IsDBNull()` method of a `DataReader` object. This method returns a Boolean `true` or `false` value that indicates whether the column value is null. You can then use that Boolean result to decide what to do. For example:

```
if (productsSqlDataReader.IsDBNull(unitPriceColPos))
{
    Console.WriteLine("UnitPrice column contains a null value");
}
else
{
    unitPrice = productsSqlDataReader.GetDecimal(unitPriceColPos);
}
```

Because `productsSqlDataReader.IsDBNull(unitPriceColPos)` returns `true`, this example displays:

UnitPrice column contains a null value

As mentioned, a `Sql*` type can store a null value. A null value is stored as `Null`. For example:

```
SqlMoney unitPrice =
    productsSqlDataReader.GetSqlMoney(unitPriceColPos);
Console.WriteLine("unitPrice = " + unitPrice);
```

This example displays:

```
unitPrice = Null
```

Each of the `Sql*` types also has a Boolean property named `IsNull` that is `true` when the `Sql*` object contains a null value. For example:

```
Console.WriteLine("unitPrice.IsNull = " + unitPrice.IsNull);
```

This example displays:

```
unitPrice.IsNull = True
```

`True` is displayed because `unitPrice` contains `Null`.

## Executing Multiple SQL Statements

Typically, your C# program and the database will run on different computers and communicate over a network. Each time you execute a command in your program, it has to travel over the network to the database and be executed by the database, and any results must be sent back across the network to your program. That's a lot of network traffic! One way to potentially reduce network traffic is to execute multiple SQL statements at a time.

In this section, you'll see how to execute multiple SELECT statements and retrieve results, and you'll see how to execute multiple SELECT, INSERT, UPDATE, and DELETE statements that are interleaved.

### Executing Multiple *SELECT* Statements

Let's take a look at how you execute multiple SELECT statements and retrieve the results. The following code first creates a SqlCommand object named mySqlCommand and sets its CommandText property to three different SELECT statements:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 5 ProductID, ProductName " +
    "FROM Products " +
    "ORDER BY ProductID;" +
    "SELECT TOP 3 CustomerID, CompanyName " +
    "FROM Customers " +
    "ORDER BY CustomerID;" +
    "SELECT TOP 6 OrderID, CustomerID " +
    "FROM Orders " +
    "ORDER BY OrderID;" ;
```

Notice that all of the SELECT statements are separated by semi-colons.

---

### Using Table Joins

Be careful to retrieve only the rows and columns you actually need. Also, make sure you use SELECT statements that retrieve rows from multiple tables. For example, if you want to see all the orders placed by the customer with the CustomerID of ALFKI, don't perform two separate SELECT statements against the Customers and Orders tables. Instead, use a table join, as shown in the following SELECT statement:

```
SELECT Customers.CustomerID, CompanyName, OrderID
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
AND Customers.CustomerID = 'ALFKI';
```

---

To run earlier SQL statements, you call the ExecuteReader() method, which returns a SqlDataReader object:

```
SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();
```

The command will return three result sets, one for each SELECT statement. To read the first result set, you call the Read() method of mySqlDataReader. The Read() method returns `False` when there are no more rows to read. Once you're at the end of a result set, you call the NextResult() method of mySqlDataReader before reading the next result set. The NextResult() method advances mySqlDataReader onto the next result set and returns `False` when there are no more result sets.

The following code illustrates the use of the Read() and NextResult() methods to read the three result sets from the SELECT statements:

```
do
{
```

```

while (mySqlDataReader.Read())
{
    Console.WriteLine("mySqlDataReader[0] = " +
        mySqlDataReader[0]);
    Console.WriteLine("mySqlDataReader[1] = " +
        mySqlDataReader[1]);
}
} while (mySqlDataReader.NextResult());

```

Notice the use of the outer do...while loop, which tests the return value from `mySqlDataReader.NextResult()` at the end. Because a do...while loop checks the condition at the end of the loop, this means that the statements in the do...while loop execute at least once. You want to put the call to `NextResult()` at the end because it *first* attempts to advance `mySqlDataReader` to the next result set and then returns the Boolean result that indicates whether there is another result set to move to. If you put the call to `NextResult()` in a regular `while` loop, then `mySqlDataReader` would skip over the first result set, and you don't want to do that!

[Listing 9.4](#) illustrates how to execute multiple SELECT statements and read the results.

#### **Listing 9.4: EXECUTEMULTIPLESELECTS.CS**

---

```

/*
 ExecuteMultipleSelects.cs illustrates how to execute
 multiple SELECT statements using a SqlCommand object
 and read the results using a SqlDataReader object
 */

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

class ExecuteSelect
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        // set the CommandText property of the SqlCommand object to
        // the mutliple SELECT statements
        mySqlCommand.CommandText =
            "SELECT TOP 5 ProductID, ProductName " +
            "FROM Products " +
            "ORDER BY ProductID;" +
            "SELECT TOP 3 CustomerID, CompanyName " +
            "FROM Customers " +
            "ORDER BY CustomerID;" +
            "SELECT TOP 6 OrderID, CustomerID " +
            "FROM Orders " +
            "ORDER BY OrderID;" ;

        mySqlConnection.Open();
        SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();

        // read the result sets from the SqlDataReader object using
        // the Read() and NextResult() methods
        do

```

```

    {
        while (mySqlDataReader.Read())
        {
            Console.WriteLine("mySqlDataReader[0] = " +
                mySqlDataReader[0]);
            Console.WriteLine("mySqlDataReader[1] = " +
                mySqlDataReader[1]);
        }
        Console.WriteLine(""); // visually split the results
    } while (mySqlDataReader.NextResult());

    mySqlDataReader.Close();
    mySqlConnection.Close();
}
}

```

---

The output from this program is as follows:

```

mySqlDataReader[0] = 1
mySqlDataReader[1] = Chai
mySqlDataReader[0] = 2
mySqlDataReader[1] = Chang
mySqlDataReader[0] = 3
mySqlDataReader[1] = Aniseed Syrup
mySqlDataReader[0] = 4
mySqlDataReader[1] = Chef Anton's Cajun Seasoning
mySqlDataReader[0] = 5
mySqlDataReader[1] = Chef Anton's Gumbo Mix

mySqlDataReader[0] = ALFKI
mySqlDataReader[1] = Alfreds Futterkiste
mySqlDataReader[0] = ANATR
mySqlDataReader[1] = Ana Trujillo3 Emparedados y helados
mySqlDataReader[0] = ANTON
mySqlDataReader[1] = Antonio Moreno Taquería

mySqlDataReader[0] = 10248
mySqlDataReader[1] = VINET
mySqlDataReader[0] = 10249
mySqlDataReader[1] = TOMSP
mySqlDataReader[0] = 10250
mySqlDataReader[1] = HANAR
mySqlDataReader[0] = 10251
mySqlDataReader[1] = VICTE
mySqlDataReader[0] = 10252
mySqlDataReader[1] = SUPRD
mySqlDataReader[0] = 10253
mySqlDataReader[1] = HANAR

```

## Executing Multiple **SELECT, INSERT, UPDATE, and DELETE** Statements

You can interleave multiple SELECT, INSERT, UPDATE, and DELETE statements. This can save network traffic because you're sending multiple SQL statements to the database in one go. The following code first creates a SqlCommand object named mySqlCommand and sets its CommandText property to multiple interleaved SQL statements:

```

mySqlCommand.CommandText =
    "INSERT INTO Customers (CustomerID, CompanyName) " +
    "VALUES ('J5COM', 'Jason 5 Company'); " +

```

```

"SELECT CustomerID, CompanyName " +
"FROM Customers " +
"WHERE CustomerID = 'J5COM' ;" +
"UPDATE Customers " +
"SET CompanyName = 'Another Jason Company' " +
"WHERE CustomerID = 'J5COM' ;" +
"SELECT CustomerID, CompanyName " +
"FROM Customers " +
"WHERE CustomerID = 'J5COM' ;" +
"DELETE FROM Customers " +
"WHERE CustomerID = 'J5COM' ;"

```

The SQL statements are as follows:

- The `INSERT` statement adds a new row to the `Customers` table.
- The first `SELECT` statement retrieves the new row.
- The `UPDATE` statement modifies the `CompanyName` column of the row.
- The second `SELECT` statement retrieves the row again.
- The `DELETE` statement removes the row.

You can use the same `do...while` loop as shown in the [previous section](#) to retrieve the two result sets returned by the `SELECT` statements. The same loop works even though the example executes inter-leaved non-`SELECT` statements. It works because only the `SELECT` statements return result sets and the `NextResult()` method returns `true` only for the `SELECT` statements; it returns `false` for the other SQL statements. Therefore, `NextResult()` returns `false` for the `INSERT` statement and advances to result set for the first `SELECT` statement, and so on.

[Listing 9.5](#) illustrates how to execute multiple SQL statements.

### **Listing 9.5: EXECUTEMULTIPLESQL.CS**

---

```

/*
 ExecuteMultipleSQL.cs illustrates how to execute
 multiple SQL statements using a SqlCommand object
*/
using System;
using System.Data;
using System.Data.SqlClient;

class ExecuteMultipleSQL
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

        // set the CommandText property of the SqlCommand object to
        // the INSERT, UPDATE, and DELETE statements
        mySqlCommand.CommandText =
            "INSERT INTO Customers (CustomerID, CompanyName) " +
            "VALUES ('J5COM', 'Jason 5 Company');" +
            "SELECT CustomerID, CompanyName " +

```

```

"FROM Customers " +
"WHERE CustomerID = 'J5COM' ; " +
"UPDATE Customers " +
"SET CompanyName = 'Another Jason Company' " +
"WHERE CustomerID = 'J5COM' ; " +
"SELECT CustomerID, CompanyName " +
"FROM Customers " +
"WHERE CustomerID = 'J5COM' ; " +
"DELETE FROM Customers " +
"WHERE CustomerID = 'J5COM' ; " ;

mySqlConnection.Open();

SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();

// read the result sets from the SqlDataReader object using
// the Read() and NextResult() methods
do
{
    while (mySqlDataReader.Read())
    {
        Console.WriteLine("mySqlDataReader[0] = " +
            mySqlDataReader[0]);
        Console.WriteLine("mySqlDataReader[1] = " +
            mySqlDataReader[1]);
    }
    Console.WriteLine(""); // visually split the results
} while (mySqlDataReader.NextResult());

mySqlDataReader.Close();
mySqlConnection.Close();
}

```

---

The output from this program is as follows:

```

mySqlDataReader[0] = J5COM
mySqlDataReader[1] = Jason 5 Company

mySqlDataReader[0] = J5COM
mySqlDataReader[1] = Another Jason Company

```

**Team LiB**

[◀ PREVIOUS](#)

[NEXT ▶](#)

**Team LiB**

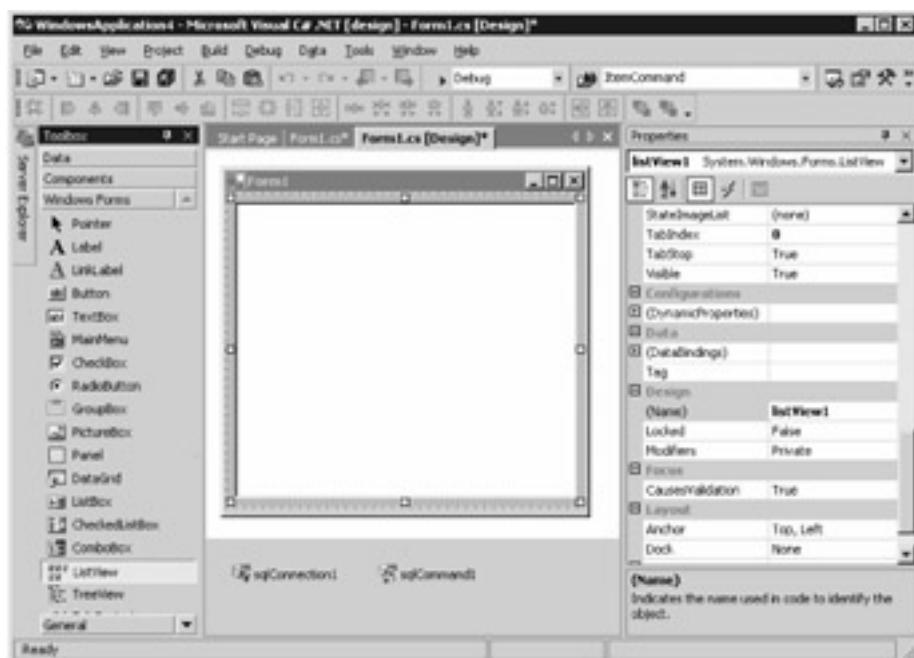
## Using a *DataReader* Object in Visual Studio .NET

You can't visually create a *DataReader* object in Visual Studio .NET (VS .NET); you can only create them using program statements.

In this section, you'll see how to create a *SqlDataReader* object and use it to retrieve the result set from a *SqlCommand* object, which you saw how to create using VS .NET in the [previous chapter](#). That *SqlCommand* object contained a *SELECT* statement that retrieved the *CustomerID*, *CompanyName*, and *ContactName* columns from the *Customers* table. You'll see how to execute this *SELECT* statement, read the result set using the *SqlDataReader* object, and display the result set in a *ListView* control. A *ListView* control allows you to view information laid out in a grid.

**Note** You can either modify the *MyDataReader* project you created in the [previous chapter](#), or if you don't want to follow along with the instructions in this section, you can simply open the completed VS .NET project contained in the *DataReader* directory. To open the completed project, select File ➤ Open ➤ Project, browse to the *vs .NET projects\DataReader* directory, and open the *windowsApplication4.csproj* file.

If you are modifying your existing Windows application, drag a *ListView* control to your form. [Figure 9.2](#) shows a form with a *ListView* control. Make sure the *Name* property of your *ListView* is set to *listView1* (this is the default name, so you shouldn't have to change it).



**Figure 9.2:** Adding a *ListView* control to the form

**Warning** If you opened the completed project, you don't have to add a *ListView* control; it's already on the completed form. You will need to change the *ConnectionString* property of the *sqlConnection1* object so that it connects to your SQL Server Northwind database. Once you've set your *ConnectionString*, you can run the form by selecting Debug ➤ Start Without Debugging.

Next, double-click an area on your form outside the *ListView* control. This causes VS .NET to display the code editor, and you'll see the cursor positioned in the *Form1\_Load()* method; this method is called when your form is initially loaded at runtime. Typically, this is the method by which you want to execute your database operations. Set your *Form1\_Load()* method to the following code:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    sqlConnection1.Open();
    System.Data.SqlClient.SqlDataReader mySqlDataReader =
        sqlCommand1.ExecuteReader();
    while (mySqlDataReader.Read())
    {
```

```

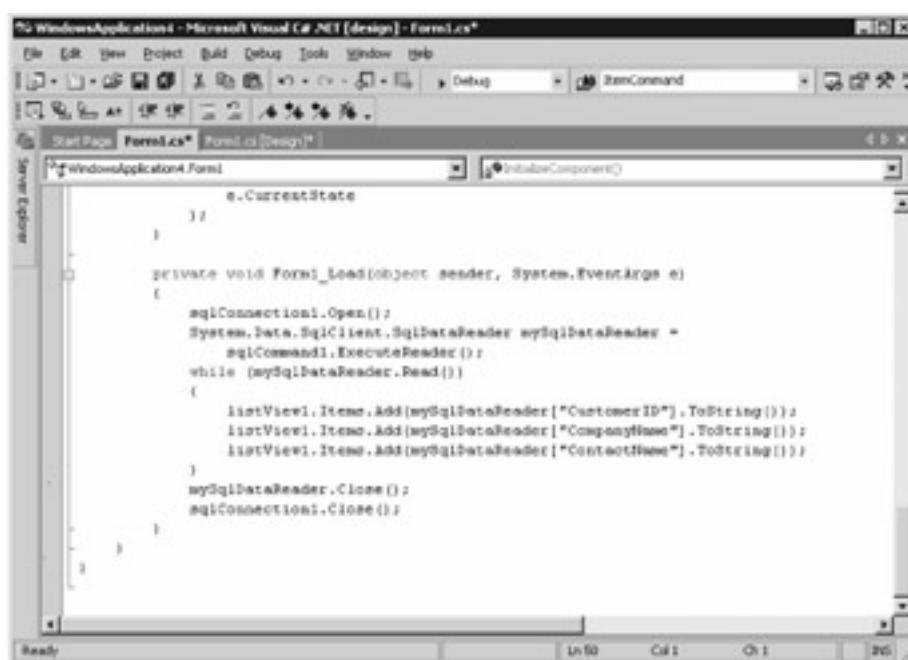
listView1.Items.Add(mySqlDataReader[ "CustomerID" ].ToString());
listView1.Items.Add(mySqlDataReader[ "CompanyName" ].ToString());
listView1.Items.Add(mySqlDataReader[ "ContactName" ].ToString());
}
mySqlDataReader.Close();
sqlConnection1.Close();
}

```

Notice you add an item to the `ListView` control using the `Add()` method, which is accessed using the `Items` property. The `Add()` method expects a `string` parameter, and you therefore call the `ToString()` method to convert the object returned by the `SqlDataReader` object to a `string`. Also notice you include the namespace when referencing the `SqlDataReader` class: you use `System.Data.SqlClient.SqlDataReader` when creating the `SqlDataReader` object.

The previous code opens the database connection, creates a `SqlDataReader` object, and uses it to read the rows from the result set returned by the `SqlCommand` object. Each column of the result set is then added to the `ListView` control using the `Add()` method.

[Figure 9.3](#) shows the completed `Form1_Load()` method.



**Figure 9.3:** The completed `Form1_Load()` method

Before you run your form, you'll need to add a substring containing the password for the database connection to the `ConnectionString` property of your `SqlConnection` object. For my installation of SQL Server, the password to access the Northwind database is `sa`, and my `ConnectionString` property is set to:

```
data source=localhost;initial catalog=Northwind;persist security info=False;user id=sa;pwd=sa;workstation id=JMPRICE-DT1;packet size=4096
```

Notice the substring `pwd=sa` in this string to set the password.

Finally, run your form by pressing `Ctrl+F5` on your keyboard, or select `Debug` à `Start Without Debugging`. [Figure 9.4](#) shows the running form.

ALFKI	Alfreds	Maria	ANATR	Ana	Ana
Futte...	Anders		Trujill...	Trujillo	
ANTON	Antonio	Antonio	AROUT	Around	Thomas
More...	Moreno			the ...	Hardy
BERGS	Bergl...	Christ...	BLAUS	Blauer	Hanna
snab...	Bergl...		See D...		Moos
BLONP	Blond...	Frédé...	BOLID	Bólido	Martín
père ...	Citeaux			Comi...	Sommer
BONAP	Bon	Laure...	BOTTM	Botto...	Elizab...
app'	Lebihan			Markets	Lincoln
BSBEV	B's	Victoria	CACTU	Cactus	Patricio
Beve...	Ashw...			Comi...	Simp...

**Figure 9.4:** The running form

Save your MyDataReader project by selecting File > Save All. You'll use this project in later chapters. If you used the completed DataReader project rather than modifying your existing project, don't worry: you'll be able to use the completed DataReader project in the later chapters also.

Team LiB

◀ PREVIOUS    NEXT ▶

**Team LiB****◀ PREVIOUS****NEXT ▶**

## Summary

In this chapter, you learned how to use a `DataReader` object to read results returned from the database. `DataReader` objects are part of the managed providers, and there are three `DataReader` classes: `SqlDataReader`, `OleDbDataReader`, and `OdbcDataReader`. You use an object of the `SqlDataReader` class to read rows retrieved from a SQL Server database; an object of the `OleDbDataReader` class to read rows from any database that supports OLE DB, such as Oracle or Access; and an object of the `OdbcDataReader` class to read rows from any database that supports ODBC.

`DataReader` objects can be used to read rows only in a forward direction, and you cannot use them to modify the database, but they do offer very fast retrieval of rows. `DataReader` objects act as an alternative to a `DataSet` object. `DataSet` objects allow you to store a copy of the rows from the database, and you can work with that copy while disconnected from the database.

**Team LiB****◀ PREVIOUS****NEXT ▶**

## Chapter 10: Using *Dataset* Objects to Store Data

### Overview

In this chapter, you'll learn the details of using `DataSet` objects to store results returned from the database. `DataSet` objects allow you to store a copy of the information from the database, and you can work with that local copy while disconnected from the database. Unlike managed provider objects such as the `SqlDataReader`, a `DataSet` is generic and therefore works with any database. A `DataSet` object also allows you to read rows in any order and modify rows—unlike a `SqlDataReader`, which allows you to read rows only in a sequential forward direction. That's not to say `SqlDataReader` objects are bad: as you learned in [Chapter 9](#), they offer very fast access to data.

You'll also learn the details of using a `DataAdapter` object to read rows from the database into a `DataSet`. The `DataAdapter` is part of the managed provider classes, and there are three `DataAdapter` classes: `SqlDataAdapter`, `OleDbDataAdapter`, and `OdbcDataAdapter`.

You use a `DataAdapter` to copy rows from the database to your `DataSet` and also push any changes you make to the rows in your `DataSet` to the database. You'll see how to make changes to the rows in a `DataSet` and push those changes to the database in [Chapter 11](#), "Using `DataSet` Objects to Modify Data." In this chapter, you'll focus on copying rows from the database and storing them in a `DataSet`.

Featured in this chapter:

- The `SqlDataAdapter` class
- Creating a `SqlDataAdapter` object
- The `DataSet` class
- Creating a `DataSet` object
- Populating a `DataSet` object
- Using the `Merge()` method
- Writing and reading XML using a `DataSet` object
- Mapping tables and columns
- Creating and using strongly typed `DataSet` classes
- Creating a `DataAdapter` object and `DataSet` object using Visual Studio .NET

## The *SqlDataAdapter* Class

You use an object of the `SqlDataAdapter` class to synchronize data stored in a `DataSet` object with a SQL Server database. You use an object of the `OleDbDataAdapter` class to synchronize data with a database that supports OLE DB, such as Oracle or Access. You use an object of the `OdbcDataAdapter` class to synchronize data with a database that supports ODBC.

**Note** Although the `SqlDataAdapter` class is specific to SQL Server, many of the properties and methods in this class are the same as those for the `OleDbDataAdapter` and `OdbcDataAdapter` classes.

[Table 10.1](#) shows some of the `SqlDataAdapter` properties.

**Table 10.1: `SqlDataAdapter` PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
<code>AcceptChangesDuringFill</code>	<code>bool</code>	Gets or sets a <code>bool</code> that indicates whether the <code>AcceptChanges()</code> method is called after a <code>DataRow</code> object has been added, modified, or removed in a <code>DataTable</code> object. The default is <code>true</code> .
<code>ContinueUpdateOnError</code>	<code>bool</code>	<p>Gets or sets a <code>bool</code> that indicates whether to continue updating the database when an error occurs.</p> <p>When set to <code>true</code>, no exception is thrown when an error occurs during the update of a row. The update of the row is skipped and the error information is placed in the <code>RowError</code> property of the <code>DataRow</code> that caused the error. The <code>DataAdapter</code> continues to update subsequent rows.</p> <p>When set to <code>false</code>, an exception is thrown when an error occurs. The default is <code>false</code>.</p>
<code>DeleteCommand</code>	<code>SqlCommand</code>	Gets or sets a command containing a SQL <code>DELETE</code> statement or stored procedure call to remove rows from the database.
<code>InsertCommand</code>	<code>SqlCommand</code>	Gets or sets a command containing a SQL <code>INSERT</code> statement or stored procedure call to add rows to the database.

MissingMappingAction	MissingMappingAction	<p>Gets or sets the action to take when the incoming table or column doesn't have a matching table or column in the <code>TableMappings</code> collection.</p> <p>The values for this action come from the <code>System.Data</code>.</p> <p><code>MissingMappingAction</code> enumeration with the members <code>Error</code>, <code>Ignore</code>, and <code>Passthrough</code>:</p> <ul style="list-style-type: none"> <li>• <code>Error</code> means a <code>SystemException</code> is thrown.</li> <li>• <code>Ignore</code> means the table or column is ignored and not read.</li> <li>• <code>Passthrough</code> means the table or column is added to the <code>DataSet</code> with its original name.</li> </ul> <p>The default is <code>Passthrough</code>.</p>
MissingSchemaAction	MissingSchemaAction	<p>Gets or sets the action to take when the incoming column doesn't have a matching column in the <code>DataTable</code> object's <code>Column</code> collection.</p> <p>The values for this action come from the <code>System.Data</code>.</p> <p><code>MissingSchemaAction</code> enumeration with the members <code>Add</code>, <code>AddWithKey</code>, <code>Error</code>, and <code>Ignore</code>:</p> <ul style="list-style-type: none"> <li>• <code>Add</code> means the column is added to the <code>DataTable</code>.</li> <li>• <code>AddWithKey</code> means the column and primary key information is added to the <code>DataTable</code>.</li> <li>• <code>Error</code> means a <code>SystemException</code> is thrown.</li> <li>• <code>Ignore</code> means the column is ignored and not read.</li> </ul> <p>The default is <code>Add</code>.</p>
SelectCommand	SqlCommand	Gets or sets a command containing a SQL <code>SELECT</code> statement or stored procedure call to retrieve rows from the database.
TableMappings	DataTableMappingCollection	Gets a <code>DataTableMappingCollection</code> that contains the mapping between a database table and a <code>DataTable</code> object in the <code>DataSet</code> .
UpdateCommand	SqlCommand	Gets or sets a command containing a SQL <code>UPDATE</code> statement or stored procedure call to modify rows in the database.

[Table 10.2](#) shows some of the `SqlDataAdapter` methods.

**Table 10.2: `sqlDataAdapter` METHODS**

METHOD	RETURN TYPE	DESCRIPTION
<code>Fill()</code>	<code>int</code>	Overloaded. Synchronizes the rows in the <code>DataSet</code> object to match those in the database. The <code>int</code> returned by this method is the number of rows synchronized in the <code>DataSet</code> with the database.
<code>FillSchema()</code>	<code>DataTable[]</code>	Overloaded. Adds a <code>DataTable</code> to a <code>DataSet</code> object and configures the schema to match the database.
<code>GetFillParameters()</code>	<code>IDataParameter[]</code>	Returns an array of any parameters set for the SQL <code>SELECT</code> statement.
<code>Update()</code>	<code>int</code>	Overloaded. Calls the respective SQL <code>INSERT</code> , <code>UPDATE</code> , or <code>DELETE</code> statements or stored procedure call (stored in the <code>InsertCommand</code> , <code>UpdateCommand</code> , and <code>DeleteCommand</code> properties, respectively) for each row that has been added, modified, or removed from a <code>DataTable</code> object. The <code>int</code> returned by this method is the number of rows updated.

[Table 10.3](#) shows some of the `SqlDataAdapter` events.

**Table 10.3: `sqlDataAdapter` EVENTS**

EVENT	EVENT HANDLER	DESCRIPTION
<code>FillError</code>	<code>FillErrorEventHandler</code>	Fires when an error occurs during a fill operation.
<code>RowUpdating</code>	<code>RowUpdatingEventHandler</code>	Fires <i>before</i> a row is added, modified, or deleted in the database.
<code>RowUpdated</code>	<code>RowUpdatedEventHandler</code>	Fires <i>after</i> a row is added, modified, or deleted in the database.

You'll learn how to use some of these properties and methods to store data in `DataSet` objects in this chapter. You'll see how to use the other properties, methods, and the events in [Chapter 11](#), in which you'll learn how to modify data in `DataSet` objects, and then push those modifications to the database.

## Creating a `SqlDataAdapter` Object

You create a `SqlDataAdapter` object using one of the following `SqlDataAdapter` constructors:

```
SqlDataAdapter()
SqlDataAdapter(SqlCommand mySqlCommand)
SqlDataAdapter(string selectCommandString, SqlConnection mySqlConnection)
SqlDataAdapter(string selectCommandString, string connectionString)
```

where

`mySqlCommand` specifies your `SqlCommand` object.

**selectCommandString** specifies your SELECT statement or stored procedure call.

**mySqlConnection** specifies your SqlConnection object.

**connectionString** specifies your connection string to connect to the database.

The following example uses the `SqlDataAdapter()` constructor to create a `SqlDataAdapter` object:

```
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
```

Before using `mySqlDataAdapter` to populate a `DataSet`, you must set its `SelectCommand` property to a `SqlCommand` that contains a SELECT command or stored procedure call. The following example creates a `SqlCommand` object with its `CommandText` property set to a SELECT statement that will retrieve the top five rows from the `Products` table, and sets the `CommandText` property of `mySqlDataAdapter` to that `SqlCommand` object:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 5 ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID";
mySqlDataAdapter.SelectCommand = mySqlCommand;
```

The next example uses the `SqlDataAdapter(SqlCommand mySqlCommand)` constructor:

```
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter(mySqlCommand);
```

The next example uses the `SqlDataAdapter(string selectCommandString, SqlConnection mySqlConnection)` constructor:

```
SqlConnection mySqlConnection =
    new SqlConnection(
        "server=localhost;database=Northwind;uid=sa;pwd=sa");
string selectCommandString =
    "SELECT TOP 10 ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID";
SqlDataAdapter mySqlDataAdapter =
    new SqlDataAdapter(selectCommandString, mySqlConnection);
```

The final example uses the `SqlDataAdapter(string selectCommandString, string connectionString)` constructor:

```
string selectCommandString =
    "server=localhost;database=Northwind;uid=sa;pwd=sa";
string connectionString =
    "SELECT TOP 10 ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID";
SqlDataAdapter mySqlDataAdapter =
    new SqlDataAdapter(selectCommandString, connectionString);
```

**Warning** This constructor causes the `SqlDataAdapter` object to create a separate `SqlConnection` object. This is typically undesirable from a performance perspective because opening a connection to the database using a `SqlConnection` object takes a relatively long time. You should therefore avoid using the `SqlDataAdapter(string selectCommandString, string connectionString)` constructor. Instead, you should use an existing `SqlConnection` object with your `SqlDataAdapter` object.

A `DataAdapter` object doesn't store rows: it merely acts as a conduit between the database and an object of the `DataSet` class. In the [next section](#), you'll learn about the `DataSet` class.

**Team LiB**

◀ PREVIOUS    NEXT ▶

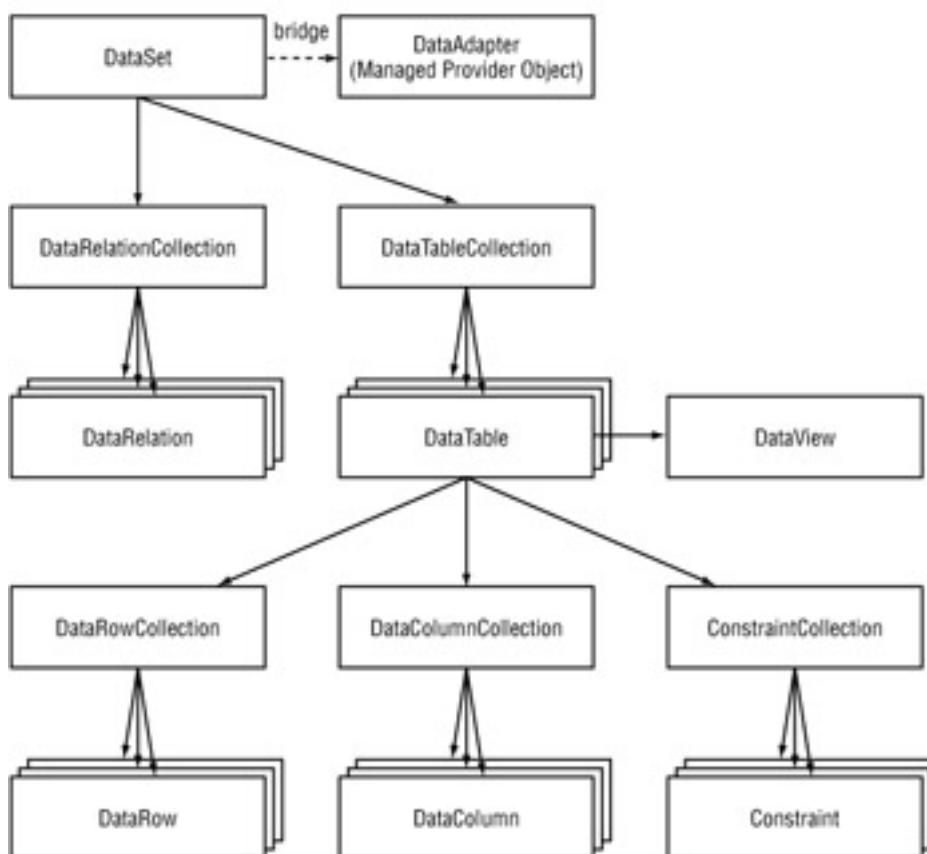
## Team LiB

## The **DataSet** Class

You use an object of the **DataSet** class to represent a local copy of the information stored in the database. You can make changes to that local copy in your **DataSet** and then later synchronize those changes with the database through a **DataAdapter**. A **DataSet** can represent database structures such as tables, rows, and columns. You can even add constraints to your locally stored tables to enforce unique and foreign key constraints.

[Figure 10.1](#) shows the **DataSet** and its relationship to some of the objects you can store within it. As you can see from this figure, you can store multiple **DataTable** objects in a **DataSet**, and so on.

Generic Data Set Objects



**Figure 10.1:** Some of the **DataSet** objects

[Table 10.4](#) shows some of the **DataSet** properties.

**Table 10.4: DataSet PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
CaseSensitive	bool	Gets or sets a <code>bool</code> value that indicates whether string comparisons within <code>DataTable</code> objects are case-sensitive.
DataSetName	string	Gets or sets the name of the current <code>DataSet</code> object.
DefaultViewManager	<code>DataViewManager</code>	Gets a custom view of the data stored in the <code>DataSet</code> object. You use a view to filter, search, and navigate the <code>DataSet</code> .

EnforceConstraints	bool	Gets or sets a bool value that indicates whether constraint rules are followed when updating information in the <code>DataSet</code> object.
ExtendedProperties	PropertyCollection	Gets a collection ( <code>PropertyCollection</code> ) of user information. You can use the <code>PropertyCollection</code> to store strings with any additional information you want. You use the <code>Add()</code> method through <code>ExtendedProperties</code> to add a string.
HasErrors	bool	Gets a bool value that indicates whether there are errors in any of the rows in the tables of the <code>DataSet</code> object.
Locale	CultureInfo	Gets or sets a <code>CultureInfo</code> object for the <code>DataSet</code> . A <code>CultureInfo</code> object contains information about a specific culture including its name, writing system, and calendar.
Namespace	string	Gets or sets the namespace for the <code>DataSet</code> object. The namespace is a string that is used when reading and writing an XML document using the <code>ReadXml()</code> , <code>WriteXml()</code> , <code>ReadXmlSchema()</code> , and <code>WriteXmlSchema()</code> methods. The namespace is used to scope the XML attributes and elements.
Prefix	string	Gets or sets the XML prefix for the <code>DataSet</code> namespace. The prefix is used in an XML document to identify the elements that belong to the <code>DataSet</code> object's namespace.
Relations	DataRelationCollection	Gets the collection of relations ( <code>DataRelationCollection</code> ) that allows navigation from a parent table to a child table. A <code>DataRelationCollection</code> consists of <code>DataRelation</code> objects.
Tables	DataTableCollection	Gets the collection of tables ( <code>DataTableCollection</code> ) that contains the <code>DataTable</code> objects stored in the <code>DataSet</code> .

[Table 10.5](#) shows some of the `DataSet` methods.

**Table 10.5: DataSet METHODS**

METHOD	RETURN TYPE	DESCRIPTION
<code>AcceptChanges()</code>	void	Commits all the changes made to the <code>DataSet</code> object since it was loaded or since the last time the <code>AcceptChanges()</code> method was called.
<code>BeginInit()</code>	void	Used by the Visual Studio .NET designer to initialize a <code>DataSet</code> used in a form or component.
<code>Clear()</code>	void	Removes all rows from all tables in the <code>DataSet</code> object.

<code>Clone()</code>	<code>DataSet</code>	Clones the structure of the <code>DataSet</code> object and returns that clone. The clone contains all the schemas, relations, and constraints.
<code>Copy()</code>	<code>DataSet</code>	Copies the structure and data of the <code>DataSet</code> object and returns that copy. The copy contains all the schemas, relations, constraints, and data.
<code>EndInit()</code>	<code>void</code>	Used by the Visual Studio .NET designer to end initialization of a <code>DataSet</code> used in a form or component.
<code>GetChanges()</code>	<code>DataSet</code>	Overloaded. Gets a copy of all the changes made to the <code>DataSet</code> object since it was last loaded or since the last time the <code>AcceptChanges()</code> method was called.
<code>GetXmL()</code>	<code>string</code>	Returns the XML representation of the data stored in the <code>DataSet</code> object.
<code>GetXmLSchema()</code>	<code>string</code>	Returns the XML representation of the schema for the <code>DataSet</code> object.
<code>HasChanges()</code>	<code>bool</code>	Overloaded. Returns a <code>bool</code> value that indicates whether the <code>DataSet</code> object has changes that haven't been committed.
<code>Merge()</code>	<code>void</code>	Overloaded. Merges this <code>DataSet</code> with another specified <code>DataSet</code> object.
<code>ReadXmL()</code>	<code>XmlReadMode</code>	Overloaded. Loads the data from an XML file into the <code>DataSet</code> object.
<code>ReadXmLSchema()</code>	<code>void</code>	Overloaded. Loads a schema from an XML file into the <code>DataSet</code> object.
<code>RejectChanges()</code>	<code>void</code>	Undoes all the changes made to the <code>DataSet</code> object since it was created or since the last time the <code>AcceptChanges()</code> method was called.
<code>Reset()</code>	<code>void</code>	Resets the <code>DataSet</code> object to its original state.
<code>WriteXmL()</code>	<code>void</code>	Overloaded. Writes out the data from the <code>DataSet</code> object to an XML file.
<code>WriteXmLSchema()</code>	<code>void</code>	Overloaded. Writes out the schema of the <code>DataSet</code> object to an XML file.

[Table 10.6](#) shows one of the `DataSet` events.

#### Table 10.6: `DataSet` EVENT

EVENT	EVENT HANDLER	DESCRIPTION
<code>MergeFailed</code>	<code>MergeFailedEventHandler</code>	Fires when an attempt is made add a <code>DataRow</code> to a <code>DataSet</code> when a <code>DataRow</code> with the same primary key value already exists in that <code>DataSet</code> .

In the [next section](#), you'll learn how to create a `DataSet` object.

## Creating a `DataSet` Object

You create a `DataSet` object using one of the following `DataSet` constructors:

```
DataSet()
DataSet(string dataSetNameString)
```

where `dataSetNameString` is the string assigned to the `DataSetName` property of your `DataSet` object. The setting of the `DataSetName` property is optional.

The following example uses the `DataSet()` constructor to create a `DataSet` object:

```
DataSet myDataSet = new DataSet();
```

The next example uses the `DataSet(string dataSetNameString)` constructor to create a `DataSet` object:

```
DataSet myDataSet = new DataSet("myDataSet");
```

## Populating a *DataSet* Object

In this section, you'll learn how to populate a `DataSet` using the `Fill()` method of a `DataAdapter`. Specifically, you'll see how to populate a `DataSet` using

- A `SELECT` statement
- A range of rows
- A stored procedure

### Using a *SELECT* Statement

Before you populate a `DataSet` you first need a `Connection`, a `Command`, and a `DataAdapter`:

```
SqlConnection mySqlConnection =
    new SqlConnection(
        "server=localhost;database=Northwind;uid=sa;pwd=sa"
    );
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 5 ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
```

Notice the `mySqlCommand` object contains a `SELECT` statement that retrieves the `ProductID`, `ProductName`, and `UnitPrice` columns of the top five rows from the `Products` table.

---

### RETRIEVING FROM MULTIPLE TABLES

Of course, you're not limited to a `SELECT` statement that retrieves from a single table. You can use a `SELECT` statement that retrieves from multiple tables using a join, however, you should typically avoid doing that because a `DataTable` is meant to be used to store rows from a *single* database table.

---

Next, to populate `myDataSet` with the rows from the `Products` table, you call the `Fill()` method of `mySqlDataAdapter`. For example:

```
int numberOfRows = mySqlDataAdapter.Fill(myDataSet, "Products");
```

The `int` returned by the `Fill()` method is the number of rows synchronized between the `DataSet` and the database via the `DataAdapter`. In the previous example, the `int` is the number of rows copied from the `Products` table to `myDataSet` and is set to 5—the number of rows retrieved by the `SELECT` statement shown earlier.

The first parameter to the `Fill()` method is your `DataSet`, and the second parameter is a string containing the name you want to assign to the `DataTable` created in your `DataSet`.

**Note** The name you assign to your `DataTable` doesn't have to be the same as the name of the database table. You can use any string of text, though typically you should still use the same name, since it will help you keep track of what database table was used to populate the `DataTable`.

When you call the `Fill()` method for the first time, the following steps are performed by ADO.NET:

1. The SELECT statement in your `SqlCommand` is executed.
2. A new `DataTable` object is created in your `DataSet`.
3. Your `DataTable` is populated with the result set returned by the SELECT statement.

If you're finished with the database after calling the `Fill()` method, you should close your `Connection` object using the `Close()` method:

```
mySqlConnection.Close();
```

**Note** The `Fill()` method will actually open and close the `Connection` for you if you don't open it first, however, it is better to explicitly open and close the `Connection` because that way it is clearer what your program is doing. Also, if you're calling the `Fill()` method repeatedly over a short span of code, you'll want to keep the database connection open and close it only when you're finished.

The `DataSet` is now populated with a `DataTable` named `Products`. You can read the `Products` `DataTable` from `myDataSet` using the following example:

```
DataTable myDataTable = myDataSet.Tables["Products"];
```

You can also read the `Products` `DataTable` using an `int` value:

```
DataTable myDataTable = myDataSet.Tables[0];
```

You can display the column values for each row in `myDataTable` using the following `foreach` loop that iterates over the `DataRow` objects stored in `myDataTable`; notice the use of the `myDataTable` object's `Rows` property:

```
foreach (DataRow myDataRow in myDataTable.Rows)
{
    Console.WriteLine("ProductID = " + myDataRow["ProductID"]);
    Console.WriteLine("ProductName = " + myDataRow["ProductName"]);
    Console.WriteLine("UnitPrice = " + myDataRow["UnitPrice"]);
}
```

The `Rows` property returns a `DataRowCollection` object that allows you to access all the `DataRow` objects stored in `myDataTable`. You can read each column value in a `DataRow` using the name of the column; for example, to read the `ProductID` column value you use `myDataRow["ProductID"]`. You can also use the numeric position of the column; for example, `myDataRow[0]` returns the value for the first column. This is the `ProductID` column.

You can also use the following code to iterate over all the `DataTable`, `DataRow`, and  `DataColumn` objects stored in `myDataSet`:

```
foreach (DataTable myDataTable in myDataSet.Tables)
{
    foreach (DataRow myDataRow in myDataTable.Rows)
    {
        foreach (DataColumn myDataColumn in myDataTable.Columns)
        {
            Console.WriteLine(myDataColumn + "= " +
                myDataRow[myDataColumn]);
        }
    }
}
```

```

        }
    }
}
```

Notice you don't need to know the names of the `DataTable` or  `DataColumn` objects to display them. The call to the `WriteLine()` method displays `myDataColumn`, which returns the name of the column, and `myDataRow[myDataColumn]`, which returns the column value for the current row.

**Note** You'll see the details of the `DataTable`, `DataRow`, and  `DataColumn` classes in [Chapter 11](#).

[Listing 10.1](#) shows a program that uses the code examples shown in this section.

### **Listing 10.1: POPULATEDATASETUSINGSELECT.CS**

---

```

/*
 PopulateDataSetUsingSelect.cs illustrates how to populate a DataSet
 object using a SELECT statement
*/

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

class PopulateDataSetUsingSelect
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        // create a SqlCommand object and set its CommandText property
        // to a SELECT statement that retrieves the top 5 rows from
        // the Products table
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 5 ProductID, ProductName, UnitPrice " +
            "FROM Products " +
            "ORDER BY ProductID";

        // create a SqlDataAdapter object and set its SelectCommand
        // property to the SqlCommand object
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;

        // create a DataSet object
        DataSet myDataSet = new DataSet();

        // open the database connection
        mySqlConnection.Open();

        // use the Fill() method of the SqlDataAdapter object to
        // retrieve the rows from the table, storing the rows locally
        // in a DataTable of the DataSet object
        Console.WriteLine("Retrieving rows from the Products table");
        int numberOfRows = mySqlDataAdapter.Fill(myDataSet, "Products");
        Console.WriteLine("numberOfRows = " + numberOfRows);
        // close the database connection
        mySqlConnection.Close();
    }
}
```

```

// get the DataTable object from the DataSet object
DataTable myDataTable = myDataSet.Tables["Products"];

// display the column values for each row in the DataTable,
// using a DataRow object to access each row in the DataTable
foreach (DataRow myDataRow in myDataTable.Rows)
{
    Console.WriteLine("ProductID = " + myDataRow["ProductID"]);
    Console.WriteLine("ProductName = " + myDataRow["ProductName"]);
    Console.WriteLine("UnitPrice = " + myDataRow["UnitPrice"]);
}
}
}

```

---

The output from this program is as follows:

```

Retrieving rows from the Products table
numberOfRows = 5
ProductID = 1
ProductName = Chai
UnitPrice = 18
ProductID = 2
ProductName = Chang
UnitPrice = 19
ProductID = 3
ProductName = Aniseed Syrup
UnitPrice = 10
ProductID = 4
ProductName = Chef Anton's Cajun Seasoning
UnitPrice = 22
ProductID = 5
ProductName = Chef Anton's Gumbo Mix
UnitPrice = 21.35

```

## Using a Range of Rows

In this section, you'll learn how to populate a `DataSet` with a range of rows. Now, the `Fill()` method is overloaded and a partial list of `Fill()` methods is as follows:

```

int Fill(DataSet myDataSet)
int Fill(DataTable myDataTable)
int Fill(DataSet myDataSet, string dataTableName)
int Fill(DataSet myDataSet, int startRow, int numRows,
        string dataTableName)

```

where

**dataTableNames** specifies a string containing the name of the `DataTable` to fill.

**startRow** is an `int` that specifies the position of the row in the result set to read (starting at 0).

**NumOfRows** is an `int` that specifies the number rows to read.

The range of rows from `startRow` to `startRow + numRows` is then stored in the `DataTable`. The `int` returned by the `Fill()` method is the number of rows retrieved from the database.

As you can see, the final `Fill()` method allows you to populate a `DataSet` with a range of rows. The following example shows the use of this `Fill()` method to store a range of rows. It retrieves the top five rows from the `Products` table, but stores only three rows in the `Products DataTable`, starting at position 1 (because rows are numbered starting at 0, position 1 corresponds to the *second* row in the result set returned by the `SELECT` statement):

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 5 ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
int numberOfRows = mySqlDataAdapter.Fill(myDataSet, 1, 3, "Products");
```

The `numberOfRows` variable is set to 3—the number of rows `myDataSet` was populated with. One thing to remember is the `DataAdapter` still retrieves all five rows from the `Products` table, but only three are actually used to populate the `DataSet`: the other two are thrown away.

[Listing 10.2](#) shows a program that uses the code examples shown in this section.

## **Listing 10.2: POPULATEDATASETUSINGRANGE.CS**

---

```
/*
 PopulateDataSetUsingRange.cs illustrates how to populate a DataSet
 object with a range of rows from a SELECT statement
 */

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

class PopulateDataSetUsingRange
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        // create a SqlCommand object and set its CommandText property
        // to a SELECT statement that retrieves the top 5 rows from
        // the Products table
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 5 ProductID, ProductName, UnitPrice " +
            "FROM Products " +
            "ORDER BY ProductID";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();

        // use the Fill() method of the SqlDataAdapter object to
        // retrieve the rows from the table, storing a range of rows
        // in a DataTable of the DataSet object
        Console.WriteLine("Retrieving rows from the Products table");
        int numberOfRows = mySqlDataAdapter.Fill(myDataSet, 1, 3, "Products");
        Console.WriteLine("numberOfRows = " + numberOfRows);
```

```

mySqlConnection.Close();

DataTable myDataTable = myDataSet.Tables["Products"];

foreach (DataRow myDataRow in myDataTable.Rows)
{
    Console.WriteLine("ProductID = " + myDataRow["ProductID"]);
    Console.WriteLine("ProductName = " + myDataRow["ProductName"]);
    Console.WriteLine("UnitPrice = " + myDataRow["UnitPrice"]);
}
}
}

```

---

The output from this program is as follows:

```

Retrieving rows from the Products table
numberOfRows = 3
ProductID = 2
ProductName = Chang
UnitPrice = 19
ProductID = 3
ProductName = Aniseed Syrup
UnitPrice = 10
ProductID = 4
ProductName = Chef Anton's Cajun Seasoning
UnitPrice = 22

```

## Using a Stored Procedure

You can also populate a `DataSet` object using a stored procedure that returns a result set. For example, the SQL Server Northwind database contains a stored procedure called `CustOrderHist()` that returns the products and total number of the products ordered by a customer. The customer's `CustomerID` is passed as a parameter to `CustOrderHist()`.

[Listing 10.3](#) shows the definition of the `CustOrderHist()` stored procedure.

### Listing 10.3: CUSTORDERHIST() STORED PROCEDURE

---

```

CREATE PROCEDURE CustOrderHist @CustomerID nchar(5)
AS
SELECT ProductName, Total=SUM(Quantity)
FROM Products P, [Order Details] OD, Orders O, Customers C
WHERE C.CustomerID = @CustomerID
AND C.CustomerID = O.CustomerID
AND O.OrderID = OD.OrderID
AND OD.ProductID = P.ProductID
GROUP BY ProductName

```

---

**Note** You don't have to create the `CustOrderHist()` procedure yourself. It's already defined in the Northwind database.

Calling `CustOrderHist()` and populating a `DataSet` with the returned result set is straightforward. For example, the following code creates a `SqlCommand` object, sets its `CommandText` object to an `EXECUTE` statement that calls `CustOrderHist()`, and sets the `@CustomerID` parameter to `ALFKI` (parameters are covered in [Chapter 8](#), "Executing Database Commands"):

```

SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "EXECUTE CustOrderHist @CustomerID";

```

```
mySqlCommand.Parameters.Add(
    "@CustomerID", SqlDbType.NVarChar, 5).Value = "ALFKI";
```

You then use code similar to that shown in the [previous section](#) to populate a `DataSet` with the result set returned by `CustOrderHist()`:

```
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
int numberOfRows = mySqlDataAdapter.Fill(myDataSet, "CustOrderHist");
mySqlConnection.Close();
```

The `CustOrderHist` `DataTable` contained within `myDataSet` is populated with the result set returned by the `CustOrderHist()` procedure.

[Listing 10.4](#) shows a program that uses the code examples shown in this section.

#### **Listing 10.4: POPULATEDATASETUSINGPROCEDURE.CS**

---

```
/*
 PopulateDataSetUsingProcedure.cs illustrates how to populate a
 DataSet object using a stored procedure
 */

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

class PopulateDataSetUsingProcedure
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        // create a SqlCommand object and set its CommandText property
        // to call the CustOrderHist() stored procedure
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "EXECUTE CustOrderHist @CustomerID";
        mySqlCommand.Parameters.Add(
            "@CustomerID", SqlDbType.NVarChar, 5).Value = "ALFKI";

        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        Console.WriteLine("Retrieving rows from the CustOrderHist() Procedure");
        int numberOfRows = mySqlDataAdapter.Fill(myDataSet, "CustOrderHist");
        Console.WriteLine("numberOfRows = " + numberOfRows);
        mySqlConnection.Close();

        DataTable myDataTable = myDataSet.Tables["CustOrderHist"];
        foreach (DataRow myDataRow in myDataTable.Rows)
        {
            Console.WriteLine("ProductName = " + myDataRow["ProductName"]);
            Console.WriteLine("Total = " + myDataRow["Total"]);
        }
    }
}
```

```

        }
    }
}
```

---

The output from this program is as follows:

```

Retrieving rows from the CustOrderHist() Procedure
numberOfRows = 11
ProductName = Aniseed Syrup
Total = 6
ProductName = Chartreuse verte
Total = 21
ProductName = Escargots de Bourgogne
Total = 40
ProductName = Flotemysost
Total = 20
ProductName = Grandma's Boysenberry Spread
Total = 16
ProductName = Lakkalikööri
Total = 15
ProductName = Original Frankfurter grüne Soße
Total = 2
ProductName = Raclette Courdavault
Total = 15
ProductName = Rössle Sauerkraut
Total = 17
ProductName = Spegesild
Total = 2
ProductName = Vegie-spread
Total = 20
```

## Populating a *DataSet* with Multiple *DataTable* Objects

You can populate a *DataSet* with multiple *DataTable* objects. You might want to do that when you need to access the information stored in multiple tables in the database.

You may use any of the following techniques to populate a *DataSet* with multiple *DataTable* objects:

- Use multiple *SELECT* statements in the same *SelectCommand*.
- Change the *CommandText* property of the *SelectCommand* before each call to the *Fill()* method.
- Use multiple *DataAdapter* objects to populate the same *DataSet*.

Let's take a look at each of these techniques.

### Using Multiple *SELECT* Statements in the Same *SelectCommand*

The following example sets the *CommandText* property of a *SqlCommand* object to two separate *SELECT* statements:

```

SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 2 ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID;" +
    "SELECT CustomerID, CompanyName " +
    "FROM Customers " +
    "WHERE CustomerID = 'ALFKI';";
```

Notice that each SELECT statement is separated by a semicolon (;). When these SELECT statements are run, two result sets are returned: one containing the two rows from the Products table, the second containing the one row from the Customers table. These two result sets are stored in separate DataTable objects by the following code:

```
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
int numberOfRows = mySqlDataAdapter.Fill(myDataSet);
mySqlConnection.Close();
```

Notice the use of the Fill(myDataSet) method, which doesn't specify the name of the DataTable to be created. Instead, the names of the two DataTable objects used to store the result sets are automatically set to the default of Table and Table1. Table stores the result set from the Products table, and Table1 stores the result set from the Customers table.

The name of a DataTable object is stored in its TableName property, which you can change. For example, the following code changes the name of the Table DataSet to Products and the Table1 DataSet to Customers:

```
myDataSet.Tables["Table"].TableName = "Products";
myDataSet.Tables["Table1"].TableName = "Customers";
```

[Listing 10.5](#) shows a program that uses the code examples shown in this section.

### **Listing 10.5: MULTIPLEDATABLES.CS**

---

```
/*
MutlipleDataTables.cs illustrates how to populate a DataSet
with multiple DataTable objects using multiple SELECT statements
*/

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

class MultipleDataTables
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        // create a SqlCommand object and set its CommandText property
        // to mutliple SELECT statements
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 2 ProductID, ProductName, UnitPrice " +
            "FROM Products " +
            "ORDER BY ProductID;" +
            "SELECT CustomerID, CompanyName " +
            "FROM Customers " +
            "WHERE CustomerID = 'ALFKI';";

        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        int numberOfRows = mySqlDataAdapter.Fill(myDataSet);
```

```

Console.WriteLine("numberOfRows = " + numberOfRows);
mySqlConnection.Close();

// change the TableName property of the DataTable objects
myDataSet.Tables["Table"].TableName = "Products";
myDataSet.Tables["Table1"].TableName = "Customers";

foreach (DataTable myDataTable in myDataSet.Tables) {
    Console.WriteLine("\nReading from the " +
        myDataTable.TableName + "DataTable");
    foreach (DataRow myDataRow in myDataTable.Rows)
    {
        foreach ( DataColumn myDataColumn in myDataTable.Columns)
        {
            Console.WriteLine(myDataColumn + "= " +
                myDataRow[myDataColumn]);
        }
    }
}
}

```

---

The output from this program is as follows:

```

numberOfRows = 3

Reading from the Products DataTable
ProductID = 1
ProductName = Chai
UnitPrice = 18
ProductID = 2
ProductName = Chang
UnitPrice = 19

Reading from the Customers DataTable
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste

```

## Changing the *CommandText* Property of the *SelectCommand*

You can also populate a *DataSet* with multiple *DataTable* objects by changing the *CommandText* property of the *SelectCommand* for your *DataAdapter* object before each call to the *Fill()* method. First, the following code populates a *DataSet* with a *DataTable* containing two rows from the *Products* table:

```

SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 2 ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
int numberOfRows = mySqlDataAdapter.Fill(myDataSet, "Products");

```

The *myDataSet* object now contains a *DataTable* named *Products*.

Next, the *CommandText* property for the *SelectCommand* of *mySqlDataAdapter* is changed to a *SELECT* statement that retrieves rows from the *Customers* table, and the *Fill()* method is called again:

```

mySqlDataAdapter.SelectCommand.CommandText =
    "SELECT CustomerID, CompanyName " +
    "FROM Customers " +
    "WHERE CustomerID = 'ALFKI'";
numberOfRows = mySqlDataAdapter.Fill(myDataSet, "Customers");
mySqlConnection.Close();

```

The myDataSet object now contains an additional DataTable named *Customers*.

[Listing 10.6](#) shows a program that uses the code examples shown in this section.

### **Listing 10.6: MULTIPLEDATABLES2.CS**

---

```

/*
 MutlipleDataTables2.cs illustrates how to populate a DataSet
 object with multiple DataTable objects by changing the
 CommandText property of a DataAdapter object's SelectCommand
 */

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

class MultipleDataTables2
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 2 ProductID, ProductName, UnitPrice " +
            "FROM Products " +
            "ORDER BY ProductID";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        int numberOfRows = mySqlDataAdapter.Fill(myDataSet, "Products");
        Console.WriteLine("numberOfRows = " + numberOfRows);

        // change the CommandText property of the SelectCommand
        mySqlDataAdapter.SelectCommand.CommandText =
            "SELECT CustomerID, CompanyName " +
            "FROM Customers " +
            "WHERE CustomerID = 'ALFKI'";
        numberOfRows = mySqlDataAdapter.Fill(myDataSet, "Customers");
        Console.WriteLine("numberOfRows = " + numberOfRows);

        mySqlConnection.Close();

        foreach (DataTable myDataTable in myDataSet.Tables) {
            Console.WriteLine("\nReading from the " +
                myDataTable.TableName + "DataTable");
            foreach (DataRow myDataRow in myDataTable.Rows)
            {
                foreach ( DataColumn myDataColumn in myDataTable.Columns)

```

```
        {
            Console.WriteLine(myDataColumn + " = " +
                myDataRow[myDataColumn]);
        }
    }
}
}
```

---

The output from this program is as follows:

```
numberOfRows = 2
numberOfRows = 1

Reading from the Products DataTable
ProductID = 1
ProductName = Chai
UnitPrice = 18
ProductID = 2
ProductName = Chang
UnitPrice = 19
```

```
Reading from the Customers DataTable
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
```

## Using Multiple *DataAdapter* Objects to Populate the Same *DataSet* Object

You can also populate the same *DataSet* with multiple *DataTable* objects using different *DataAdapter* objects. For example, assume you already have a *DataSet* named *myDataSet* that was populated using a *SqlDataAdapter* named *mySqlDataAdapter*, and that *myDataSet* currently contains a *DataTable* named *Products*. The following example creates another *SqlDataAdapter* and uses it to populate *myDataSet* with another *DataTable* named *Customers*:

```
SqlDataAdapter mySqlDataAdapter2 = new SqlDataAdapter();
mySqlDataAdapter2.SelectCommand = mySqlCommand;
mySqlDataAdapter2.SelectCommand.CommandText =
    "SELECT CustomerID, CompanyName " +
    "FROM Customers " +
    "WHERE CustomerID = 'ALFKI'";
numberOfRows = mySqlDataAdapter2.Fill(myDataSet, "Customers");
```

[Listing 10.7](#) shows a program that uses the code examples shown in this section.

### **Listing 10.7: MULTIPLEDATATABLES3.CS**

---

```
/*
MutlipleDataTables3.cs illustrates how to populate a DataSet
object with multiple DataTable objects using multiple
DataAdapter objects to populate the same DataSet object
*/
using System;
using System.Data;
using System.Data.SqlClient;
class MultipleDataTables3
{
    public static void Main()
    {
```

## The DataSet Class

```
SqlConnection mySqlConnection =
    new SqlConnection(
        "server=localhost;database=Northwind;uid=sa;pwd=sa"
    );

SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 2 ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID";
SqlDataAdapter mySqlDataAdapter1 = new SqlDataAdapter();
mySqlDataAdapter1.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
int numberOfRows = mySqlDataAdapter1.Fill(myDataSet, "Products");
Console.WriteLine("numberOfRows = " + numberOfRows);

// create another DataAdapter object
SqlDataAdapter mySqlDataAdapter2 = new SqlDataAdapter();
mySqlDataAdapter2.SelectCommand = mySqlCommand;
mySqlDataAdapter2.SelectCommand.CommandText =
    "SELECT CustomerID, CompanyName " +
    "FROM Customers " +
    "WHERE CustomerID = 'ALFKI'";
numberOfRows = mySqlDataAdapter2.Fill(myDataSet, "Customers");
Console.WriteLine("numberOfRows = " + numberOfRows);
mySqlConnection.Close();

foreach (DataTable myDataTable in myDataSet.Tables) {
    Console.WriteLine("\nReading from the " +
        myDataTable.TableName + " DataTable");
    foreach (DataRow myDataRow in myDataTable.Rows)
    {
        foreach ( DataColumn myDataColumn in myDataTable.Columns)
        {
            Console.WriteLine(myDataColumn + " = " +
                myDataRow[myDataColumn]);
        }
    }
}
}
```

---

The output from this program is as follows:

```
numberOfRows = 2
numberOfRows = 1

Reading from the Products DataTable
ProductID = 1
ProductName = Chai
UnitPrice = 18
ProductID = 2
ProductName = Chang
UnitPrice = 19

Reading from the Customers DataTable
CustomerID = ALFKI
```

```
CompanyName = Alfreds Futterkiste
```

## Merging *DataRow*, *DataSet*, and *DataTable* Objects into Another *DataSet*

In this section, you'll learn how to use the `Merge()` method to merge *DataRow*, *DataSet*, and *DataTable* objects into another *DataSet*. You might want to do this when you have multiple sources of data; for example, you might get data from many regional offices that is sent to headquarters, and you need to merge all that data into one *DataSet*.

The `Merge()` method is overloaded as follows:

```
void Merge(DataRow[] myDataRows)
void Merge(DataSet myDataSet)
void Merge(DataTable myDataTable)
void Merge(DataSet myDataSet, bool preserveChanges)
void Merge(DataRow[] myDataRows, bool preserveChanges,
    MissingSchemaAction myMissingSchemaAction)
void Merge(DataSet myDataSet, bool preserveChanges,
    MissingSchemaAction myMissingSchemaAction)
void Merge(DataTable myDataTable, bool preserveChanges,
    MissingSchemaAction myMissingSchemaAction)
```

where

- **PreserveChanges** specifies whether changes in the current *DataSet* (the *DataSet* with the `Merge()` method that is called) are to be kept.
- **MyMissingSchemaAction** specifies the action to take when the current *DataSet* doesn't have the same tables or columns as the *DataRow*, *DataSet*, or *DataTable* being merged into that *DataSet*.

You set `myMissingSchemaAction` to one of the constants defined in the `System.Data.MissingSchemaAction` enumeration. [Table 10.7](#) shows the constants defined in the `MissingSchemaAction` enumeration.

**Table 10.7: MissingSchemaAction ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
Add	The column or table is added to the current <i>DataSet</i> . Add is the default.
AddWithKey	The column and primary key information is added to the current <i>DataSet</i> .
Error	A <code>SystemException</code> is thrown.
Ignore	The column or table is ignored and not read.

[Listing 10.8](#) illustrates the use of the `Merge()` method.

**Listing 10.8: MERGE.CS**

```
/*
Merge.cs illustrates how to use the Merge() method
*/
using System;
using System.Data;
using System.Data.SqlClient;

class Merge
{
    public static void Main()
{
```

```

{
    SqlConnection mySqlConnection =
        new SqlConnection(
            "server=localhost;database=Northwind;uid=sa;pwd=sa"
        );

    SqlCommand mySqlCommand = mySqlConnection.CreateCommand();

    // populate myDataSet with three rows from the Customers table
    mySqlCommand.CommandText =
        "SELECT CustomerID, CompanyName, ContactName, Address " +
        "FROM Customers " +
        "WHERE CustomerID IN ('ALFKI', 'ANATR', 'ANTON')";
    SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
    mySqlDataAdapter.SelectCommand = mySqlCommand;
    DataSet myDataSet = new DataSet();
    mySqlConnection.Open();
    mySqlDataAdapter.Fill(myDataSet, "Customers");

    // populate myDataSet2 with two rows from the Customers table
    mySqlCommand.CommandText =
        "SELECT CustomerID, CompanyName, ContactName, Address " +
        "FROM Customers " +
        "WHERE CustomerID IN ('AROUT', 'BERGS')";
    DataSet myDataSet2 = new DataSet();
    mySqlDataAdapter.Fill(myDataSet2, "Customers2");

    // populate myDataSet3 with five rows from the Products table
    mySqlCommand.CommandText =
        "SELECT TOP 5 ProductID, ProductName, UnitPrice " +
        "FROM Products " +
        "ORDER BY ProductID";
    DataSet myDataSet3 = new DataSet();
    mySqlDataAdapter.Fill(myDataSet3, "Products");

    mySqlConnection.Close();

    // merge myDataSet2 into myDataSet
    myDataSet.Merge(myDataSet2);

    // merge myDataSet3 into myDataSet
    myDataSet.Merge(myDataSet3, true, MissingSchemaAction.Add);

    // display the rows in myDataSet
    foreach (DataTable myDataTable in myDataSet.Tables)
    {
        Console.WriteLine("\nReading from the " + myDataTable + "DataTable");
        foreach (DataRow myDataRow in myDataTable.Rows)
        {
            foreach ( DataColumn myDataColumn in myDataTable.Columns)
            {
                Console.WriteLine(myDataColumn + " = " +
                    myDataRow[myDataColumn]);
            }
        }
    }
}

```

---

The output from this program is as follows:

```
Reading from the Customers DataTable
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
ContactName = Maria Anders
Address = Obere Str. 57
CustomerID = ANATR
CompanyName = Ana Trujillo3 Emparedados y helados
ContactName = Ana Trujillo
Address = Avda. de la Constitución 2222
CustomerID = ANTON
CompanyName = Antonio Moreno Taquería
ContactName = Antonio Moreno
Address = Mataderos 2312
```

```
Reading from the Customers2 DataTable
CustomerID = AROUT
CompanyName = Around the Horn
ContactName = Thomas Hardy
Address = 120 Hanover Sq.
CustomerID = BERGS
CompanyName = Berglunds snabbköp
ContactName = Christina Berglund
Address = Berguvsvägen 8
```

```
Reading from the Products DataTable
ProductID = 1
ProductName = Chai
UnitPrice = 18
ProductID = 2
ProductName = Chang
UnitPrice = 19
ProductID = 3
ProductName = Aniseed Syrup
UnitPrice = 10
ProductID = 4
ProductName = Chef Anton's Cajun Seasoning
UnitPrice = 22
ProductID = 5
ProductName = Chef Anton's Gumbo Mix
UnitPrice = 21.35
```

## Writing and Reading XML Using a *DataSet* Object

XML is a convenient format for moving information around. You can write out the contents of the `DataTable` objects contained in a `DataSet` to an XML file using the `WriteXml()` method. The XML file written by this method contains the `DataTable` column names and values.

You can write out the schema of a `DataSet` object to an XML file using the `WriteXmlSchema()` method. The XML file written by this method contains the structure of the `DataTable` objects contained in the `DataSet`. You can also get the XML in a `DataSet` using the `GetXml()` method, which returns the XML in a string.

You can read the contents of the `DataTable` objects in an XML file into a `DataSet` using the `ReadXml()` method. You can also read the schema contained in an XML file using the `ReadXmlSchema()` method.

**Note** SQL Server also contains extensive built-in XML functionality, which you'll learn about in [Chapter 16](#), "Using SQL Server's XML Support."

### Using the `WriteXml()` Method

Let's say you have a `DataSet` object named `myDataSet`. Assume that `myDataSet` has a `DataTable` that contains the `CustomerID`, `CompanyName`, `ContactName`, and `Address` columns for the top two rows from the `Customers` table. The following code shows this:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 2 CustomerID, CompanyName, ContactName, Address " +
    "FROM Customers " +
    "ORDER BY CustomerID";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
Console.WriteLine("Retrieving rows from the Customers table");
mySqlDataAdapter.Fill(myDataSet, "Customers");
mySqlConnection.Close();
```

You can write out the contents of `myDataSet` to an XML file using the `WriteXml()` method. For example:

```
myDataSet.WriteXml("myXmlFile.xml");
```

This writes an XML file named `myXmlFile.xml`, as shown in [Listing 10.9](#).

#### **Listing 10.9: MYXMLFILE.XML**

---

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
    <Customers>
        <CustomerID>ALFKI</CustomerID>
        <CompanyName>Alfreds Futterkiste</CompanyName>
        <ContactName>Maria Anders</ContactName>
        <Address>Obere Str. 57</Address>
    </Customers>
    <Customers>
        <CustomerID>ANATR</CustomerID>
        <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
        <ContactName>Ana Trujillo</ContactName>
        <Address>Avda. de la Constitución 2222</Address>
    </Customers>
</NewDataSet>
```

As you can see, this file contains the columns for the rows retrieved from the `Customers` table.

The `WriteXml()` method is overloaded as follows:

```
void WriteXml(Stream myStream);
void WriteXml(string fileName);
void WriteXml(TextWriter myTextWriter);
void WriteXml(XmlWriter myXmlWriter);
void WriteXml(stream myStream, XmlWriteMode myXmlWriteMode);
void WriteXml(string fileName, XmlWriteMode myXmlWriteMode);
void WriteXml(TextWriter myTextWriter, XmlWriteMode myXmlWriteMode);
void WriteXml(XmlWriter myXmlWriter, XmlWriteMode myXmlWriteMode);
```

where `myXmlWriteMode` is a constant from the `System.Data.XmlWriteMode` enumeration that specifies how to write XML data and the schema. [Table 10.8](#) shows the constants defined in the `XmlWriteMode` enumeration.

**Table 10.8: `XmlWriteMode` ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
<code>DiffGram</code>	Writes out the <code>DataSet</code> as a <code>DiffGram</code> , which contains the original values and the changes to those values to make them current. You can generate a <code>DiffGram</code> that contains only the changes by calling the <code>GetChanges()</code> method of your <code>DataSet</code> , and then call <code>WriteXml()</code> .
<code>IgnoreSchema</code>	Writes out only the data in the <code>DataSet</code> , without writing the schema. <code>IgnoreSchema</code> is the default.
<code>WriteSchema</code>	Writes out the schema in the <code>DataSet</code> .

The following example shows the use of the `XmlWriteMode.WriteSchema` constant:

```
myDataSet.WriteXml("myXmlFile2.xml", XmlWriteMode.WriteSchema);
```

This writes an XML file named `myXmlFile2.xml`, as shown in [Listing 10.10](#).

### **Listing 10.10: MYXMLFILE2.XML**

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <xsd:schema id="NewDataSet" targetNamespace="" xmlns="" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xsd:element name="NewDataSet" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="Customers">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="CustomerID" type="xsd:string" minOccurs="0" />
                <xsd:element name="CompanyName" type="xsd:string" minOccurs="0" />
                <xsd:element name="ContactName" type="xsd:string" minOccurs="0" />
                <xsd:element name="Address" type="xsd:string" minOccurs="0" />
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</NewDataSet>
```

```

</xsd:schema>
<Customers>
  <CustomerID>ALFKI</CustomerID>
  <CompanyName>Alfreds Futterkiste</CompanyName>
  <ContactName>Maria Anders</ContactName>
  <Address>Obere Str. 57</Address>
</Customers>
<Customers>
  <CustomerID>ANATR</CustomerID>
  <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
  <ContactName>Ana Trujillo</ContactName>
  <Address>Avda. de la Constitución 2222</Address>
</Customers>
</NewDataSet>

```

---

As you can see, this file contains the schema definition for the columns used in the original SELECT statement, as well as the column values for the rows retrieved.

## Using the *WriteXmlSchema()* Method

You can write out the schema of myDataSet to an XML file using the *WriteXmlSchema()* method. For example:

```
myDataSet.WriteXmlSchema( "myXmlSchemaFile.xml" );
```

This writes an XML file named *myXmlSchemaFile.xml*, as shown in [Listing 10.11](#).

### **Listing 10.11: MYXMLSCHEMAFILE.XML**

```

<?xml version="1.0" standalone="yes"?>
<xsd:schema id="NewDataSet" targetNamespace="" xmlns="" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xsd:element name="NewDataSet" msdata:IsDataSet="true">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="Customers">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="CustomerID" type="xsd:string" minOccurs="0" />
              <xsd:element name="CompanyName" type="xsd:string" minOccurs="0" />
              <xsd:element name="ContactName" type="xsd:string" minOccurs="0" />
              <xsd:element name="Address" type="xsd:string" minOccurs="0" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

---

As you can see, this file contains the schema definition for the columns retrieved from the *Customers* table by the original SELECT statement.

## Using the *ReadXml()* Method

You can read the contents of an XML file into a *DataSet* object using the *ReadXml()* method. This method reads the rows and

columns from the XML file into `DataTable` objects of the `DataSet`. For example, the following statement uses the `ReadXml()` method to read the XML file `myXmlFile.xml` previously written by the `WriteXml()` method:

```
myDataSet.ReadXml("myXmlFile.xml");
```

The `ReadXml()` method is overloaded as follows:

```
void ReadXml(Stream myStream);
void ReadXml(string fileName);
void ReadXml(TextReader myTextReader);
void ReadXml(XmlReader myXmlReader);
void ReadXml(stream myStream, XmlReadMode myXmlReadMode);
void ReadXml(string fileName, XmlReadMode myXmlReadMode);
void ReadXml(TextReader myTextReader, XmlReadMode myXmlReadMode);
void ReadXml(XmlReader myXmlReader, XmlReadMode myXmlReadMode);
```

where `myXmlReadMode` is a constant from the `System.Data.XmlReadMode` enumeration that specifies how to read XML data and the schema. [Table 10.9](#) shows the constants defined in the `XmlReadMode` enumeration.

**Table 10.9: `XmlReadMode` ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
Auto	<p>Reads the XML file in an appropriate manner:</p> <ul style="list-style-type: none"> <li>If the XML file contains a <code>DiffGram</code>, then <code>XmlReadMode</code> is set to <code>DiffGram</code>.</li> <li>If the <code>DataSet</code> already contains a schema or the XML file contains a schema, then <code>XmlReadMode</code> is set to <code>ReadSchema</code>.</li> <li>If the <code>DataSet</code> doesn't contain a schema and the XML file doesn't contain a schema, then <code>XmlReadMode</code> is set to <code>InferSchema</code>.</li> </ul> <p><code>Auto</code> is the default.</p>
DiffGram	Reads the XML file as a <code>DiffGram</code> , which contains the original values and the changes to those values to make them current. The changes are then applied to your <code>DataSet</code> . This is similar to calling the <code>Merge()</code> method of a <code>DataSet</code> in that changes from one <code>DataSet</code> are merged with another.
Fragment	Reads an XML file that contains inline XDR schema fragments such as those generated by executing <code>SELECT</code> statements containing <code>FOR XML</code> clauses.
IgnoreSchema	Reads out only the data in the <code>DataSet</code> , without reading the schema.
InferSchema	Infers the schema of the XML file by examining the data stored in it.
ReadSchema	Reads the schema from the XML file into the <code>DataSet</code> .

The following example shows the use of the `XmlReadMode.ReadSchema` constant:

```
myDataSet.ReadXml("myXmlFile2.xml", XmlReadMode.ReadSchema);
```

[Listing 10.12](#) illustrates how to write and read XML files using ADO.NET.

### **Listing 10.12: WRITEANDREADXML.CS**

```
/*
 WriteAndReadXml.cs illustrates how to write and read XML files
 */

using System;
using System.Data;
```

```
using System.Data.SqlClient;

class WriteAndReadXML
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 2 CustomerID, CompanyName, ContactName, Address " +
            "FROM Customers " +
            "ORDER BY CustomerID";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        Console.WriteLine("Retrieving rows from the Customers table");
        mySqlDataAdapter.Fill(myDataSet, "Customers");
        mySqlConnection.Close();

        // use the WriteXml() method to write the DataSet out to an
        // XML file
        Console.WriteLine("Writing rows out to an XML file named " +
            "myXmlFile.xml using the WriteXml() method");
        myDataSet.WriteXml("myXmlFile.xml");

        Console.WriteLine("Writing schema out to an XML file named " +
            "myXmlFile2.xml using the WriteXml() method");
        myDataSet.WriteXml("myXmlFile2.xml", XmlWriteMode.WriteSchema);

        // use the WriteXmlSchema() method to write the schema of the
        // DataSet out to an XML file
        Console.WriteLine("Writing schema out to an XML file named " +
            "myXmlSchemaFile.xml using the WriteXmlSchema() method");
        myDataSet.WriteXmlSchema("myXmlSchemaFile.xml");

        // use the Clear() method to clear the current rows in the DataSet
        myDataSet.Clear();

        // use the ReadXml() method to read the contents of the XML file
        // into the DataSet
        Console.WriteLine("Reading rows from myXmlFile.xml " +
            "using the ReadXml() method");
        myDataSet.ReadXml("myXmlFile.xml");

        DataTable myDataTable = myDataSet.Tables["Customers"];
        foreach (DataRow myDataRow in myDataTable.Rows)
        {
            Console.WriteLine("CustomerID = " + myDataRow["CustomerID"]);
            Console.WriteLine("CompanyName = " + myDataRow["CompanyName"]);
            Console.WriteLine("ContactName = " + myDataRow["ContactName"]);
            Console.WriteLine("Address = " + myDataRow["Address"]);
        }
    }
}
```

The output from this program is as follows:

```
Retrieving rows from the Customers table
Writing rows out to an XML file named myXmlFile.xml using
the WriteXml() method
Writing schema out to an XML file named myXmlFile2.xml
using the WriteXml() method
Writing schema out to an XML file named myXmlSchemaFile.xml
using the WriteXmlSchema() method
Reading rows from myXmlFile.xml using the ReadXml() method
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
ContactName = Maria Anders
Address = Obere Str. 57
CustomerID = ANATR
CompanyName = Ana Trujillo3 Emparedados y helados
ContactName = Ana Trujillo
Address = Avda. de la Constitución 2222
```

**Team LiB**

 PREVIOUS

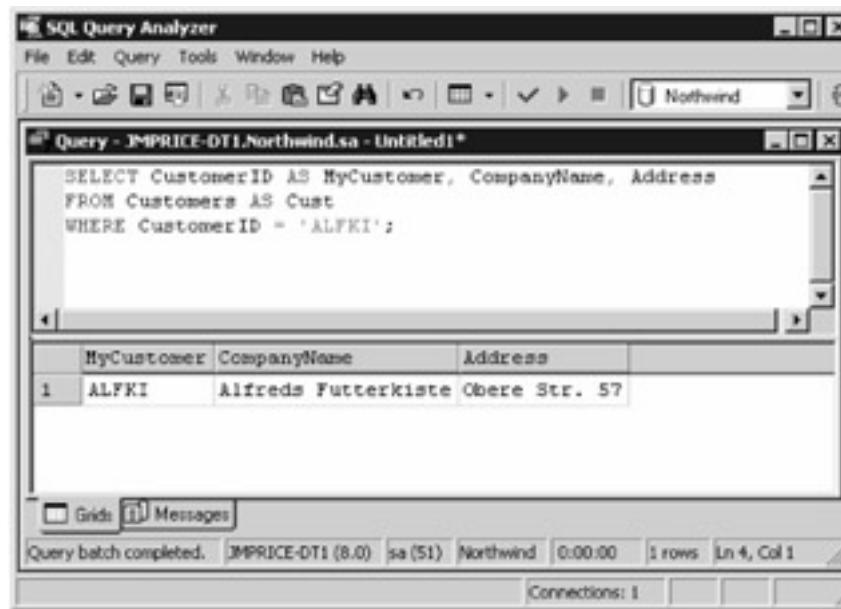
NEXT 

## Mapping Tables and Columns

In [Chapter 3](#), "Introduction to Structured Query Language (SQL)," you learned that the `AS` keyword is used to specify an alias for a table or column. The following example uses the `AS` keyword to alias the `CustomerID` column as `MyCustomer` and also alias the `Customers` table as `Cust`:

```
SELECT CustomerID AS MyCustomer, CompanyName, Address
FROM Customers AS Cust
WHERE CustomerID = 'ALFKI';
```

[Figure 10.2](#) shows the results of this `SELECT` statement.



**Figure 10.2:** Using the `AS` keyword

The following code uses this `SELECT` statement to populate a `DataSet` object named `myDataSet`:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT CustomerID AS MyCustomer, CompanyName, Address " +
    "FROM Customers AS Cust " +
    "WHERE CustomerID = 'ALFKI'";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet, "Customers");
mySqlConnection.Close();
```

Notice the `Fill()` method specifies the name of the `DataTable` as `Customers`, which is known as the source `DataTable` name.

To map a `DataTable` in your `DataSet`, you create an object of the `DataTableMapping` class using the `Add()` method; this class belongs to the `System.Data.Common` namespace, which you should import into your program. The following example creates a `DataTableMapping` object named `myDataTableMapping`, passing `Customers` and `Cust` to the `Add()` method:

```
DataTableMapping myDataTableMapping =
    mySqlDataAdapter.TableMappings.Add("Customers", "Cust");
```

Notice that the `Add()` method is called through the `TableMappings` property. The `TableMappings` property returns an object of the `TableMappingCollection` class. This object is a collection of `TableMapping` objects, and you use a `TableMapping`

object to map the source name to a different `DataTable` name, therefore, the previous example maps the source name of `Customers` to `Cust`.

You can read this mapping using the `SourceTable` and `DataSetTable` properties of `myDataTableMapping`. For example:

```
Console.WriteLine("myDataTableMapping.SourceTable = " +
    myDataTableMapping.SourceTable);
Console.WriteLine("myDataTableMapping.DataSetTable = " +
    myDataTableMapping.DataSetTable);
```

This example displays the following:

```
myDataTableMapping.DataSetTable = Cust
myDataTableMapping.SourceTable = Customers
```

You should also change the `TableName` property of the `DataTable` object in your `DataSet` to keep the names consistent:

```
myDataSet.Tables["Customers"].TableName = "Cust";
```

**Tip** It is important that you change the `TableName` since it will otherwise keep the original name of `Customers`, which is a little confusing when you've already specified the mapping from `Customers` to `Cust` earlier.

Next, to alias the `CustomerID` column as `MyCustomer`, you call the `Add()` method through the `ColumnMappings` property of `myDataTableMapping`:

```
myDataTableMapping.ColumnMappings.Add("CustomerID", "MyCustomer");
```

The `ColumnMappings` property returns an object of the `DataColumnMappingCollection` class. This object is a collection of `DataColumnMapping` objects. You use a `DataColumnMapping` object to map a column name from the database to a different `DataColumn` name, therefore, the previous example maps the `CustomerID` column name from the database to the `DataColumn` name `MyCustomer`.

[Listing 10.13](#) illustrates how to map table and column names using the code shown in this section.

### **Listing 10.13: MAPPINGS.CS**

---

```
/*
    Mappings.cs illustrates how to map table and column names
*/

using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.Common;

class Mappings
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT CustomerID AS MyCustomer, CompanyName, Address " +
            "FROM Customers AS Cust " +
            "WHERE CustomerID = 'ALFKI'";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
```

## Mapping Tables and Columns

```
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet, "Customers");
mySqlConnection.Close();

// create a DataTableMapping object
DataTableMapping myDataTableMapping =
    mySqlDataAdapter.TableMappings.Add("Customers", "Cust");

// change the TableName property of the DataTable object
myDataSet.Tables["Customers"].TableName = "Cust";

// display the DataSetTable and SourceTable properties
Console.WriteLine("myDataTableMapping.DataSetTable = " +
    myDataTableMapping.DataSetTable);
Console.WriteLine("myDataTableMapping.SourceTable = " +
    myDataTableMapping.SourceTable);

// map the CustomerID column to MyCustomer
myDataTableMapping.ColumnMappings.Add("CustomerID", "MyCustomer");

DataTable myDataTable = myDataSet.Tables["Cust"];
foreach (DataRow myDataRow in myDataTable.Rows)
{
    Console.WriteLine("CustomerID = " + myDataRow["MyCustomer"]);
    Console.WriteLine("CompanyName = " + myDataRow["CompanyName"]);
    Console.WriteLine("Address = " + myDataRow["Address"]);
}
}
```

The output from this program is as follows:

```
myDataTableMapping.DataSetTable = Cust
myDataTableMapping.SourceTable = Customers
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
Address = Obere Str. 57
```

**Team LiB**

[◀ PREVIOUS](#)

[NEXT ▶](#)

## Reading a Column Value Using Strongly Typed *DataSet* Classes

A strongly typed *DataSet* object allows you read a column value using a property with the same name as the column. For example, to read the *CustomerID* of a column, you can use *myDataRow.CustomerID* rather than *myDataRow[ "CustomerID" ]*. This is a nice feature because the compiler can then catch any errors in column spellings at compile time rather than runtime. For example, if you incorrectly spelled *CustomerID* as *CustimerID*, then the mistake would be caught by the compiler.

Another feature of a strongly typed *DataSet* is that when you work with it in VS .NET, IntelliSense automatically pops up the properties and methods of the *DataSet* when you are typing. You can then pick the property or method from the list, rather than have to type it all in.

The downside to using a strongly typed *DataSet* is that you must do some initial work to generate it before you can use it. If the columns in your database tables don't change very often, then you should consider using strongly typed *DataSet* objects. On the other hand, if your database tables change a lot, you should probably avoid them because you'll need to regenerate the strongly typed *DataSet* to keep it synchronized with the definition of the database table.

**Note** You'll find a completed VS .NET example project for this section in the *StronglyTypedDataSet* directory. You can open this project in VS .NET by selecting File > Open > Project and opening the *windowsApplication4.csproj* file. You'll need to change the *ConnectionString* property of the *sqlConnection1* object to connect to your SQL Server Northwind database. You can also follow along with the instructions in this section by copying the *DataReader* directory to another directory and using that project as your starting point.

## Creating a Strongly Typed *DataSet* Class

In this section, you'll create a strongly typed *DataSet* class that is used to access the *Customers* table. If you're following along with these instructions, open the *DataReader* project in VS .NET and double-click *Form1.cs* in the Solution Explorer window. You open the Solution Explorer window by selecting View > Solution Explorer.

Next, select File > Add New Item. Select Data Set from the Templates area and enter *MyDataSet.xsd*, as shown in [Figure 10.3](#).



**Figure 10.3:** Adding a new Data Set

Click the Open button to continue.

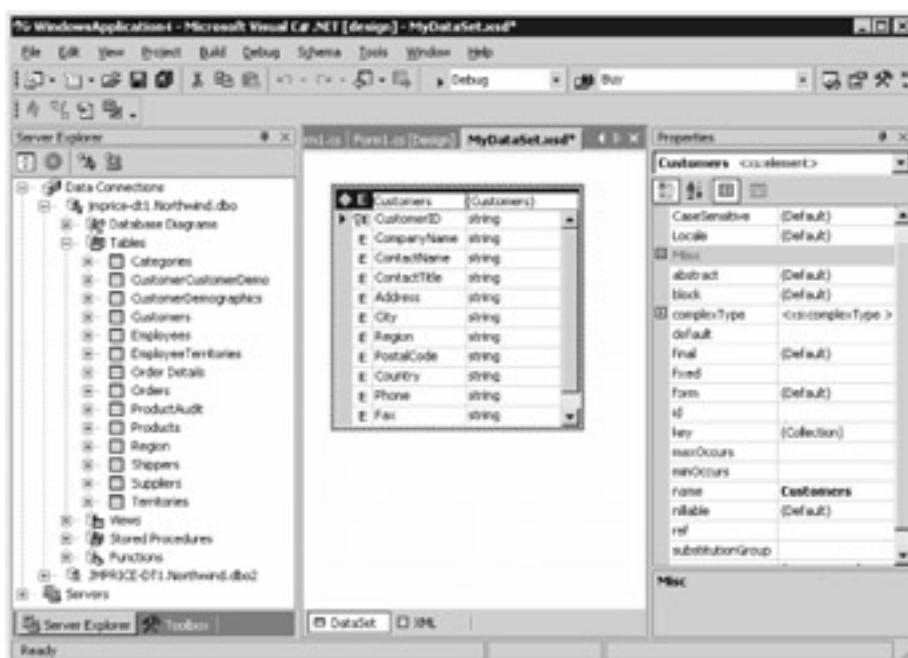
VS .NET will add *MyDataSet.xsd* to your project, as shown in [Figure 10.4](#).

**Figure 10.4:** MyDataSet .xsd

At the bottom left of [Figure 10.4](#), you'll notice two tabs: DataSet and XML. The DataSet tab is displayed by default and you use it to see the visual view of your DataSet. The XML tab allows you to see the XML file of your DataSet.

Next, make sure you've opened the Server Explorer window; to open the window, select View > Server Explorer. Open the Data Connections node and either use an existing connection to your Northwind database or create a new one by right-clicking on the Data Connections node and selecting Add Connection from the pop-up menu.

Double-click your connection and drill down to the table, view, or stored procedure you want use, and then drag it onto your form. Go ahead and drag the Customers table onto your form. [Figure 10.5](#) shows the form once the Customers table has been added.

**Figure 10.5:** Customers table added to form

**Note** You can add multiple tables to your form and define relations between them.

Next, save your work by selecting File > Save All or press Ctrl+S on your keyboard.

Your project now contains a XSD file named `MyDataSet .xsd`, as shown in [Listing 10.14](#). You can view this file by clicking the XML tab at the bottom of the XML Designer window.

**Listing 10.14: MYDATASET.XSD**

```

<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="MyDataSet"
targetNamespace="http://tempuri.org/MyDataSet.xsd" elementFormDefault="qualified"
attributeFormDefault="qualified"
xmlns="http://tempuri.org/MyDataSet.xsd"
xmlns:mstns="http://tempuri.org/MyDataSet.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="MyDataSet" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Customers">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CustomerID" type="xs:string" />
              <xs:element name="CompanyName" type="xs:string" />
              <xs:element name="ContactName" type="xs:string" minOccurs="0" />
              <xs:element name="ContactTitle" type="xs:string" minOccurs="0" />
              <xs:element name="Address" type="xs:string" minOccurs="0" />
              <xs:element name="City" type="xs:string" minOccurs="0" />
              <xs:element name="Region" type="xs:string" minOccurs="0" />
              <xs:element name="PostalCode" type="xs:string" minOccurs="0" />
              <xs:element name="Country" type="xs:string" minOccurs="0" />
              <xs:element name="Phone" type="xs:string" minOccurs="0" />
              <xs:element name="Fax" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:unique name="MyDataSetKey1" msdata:PrimaryKey="true">
      <xs:selector xpath=".//mstns:Customers" />
      <xs:field xpath="mstns:CustomerID" />
    </xs:unique>
  </xs:element>
</xs:schema>

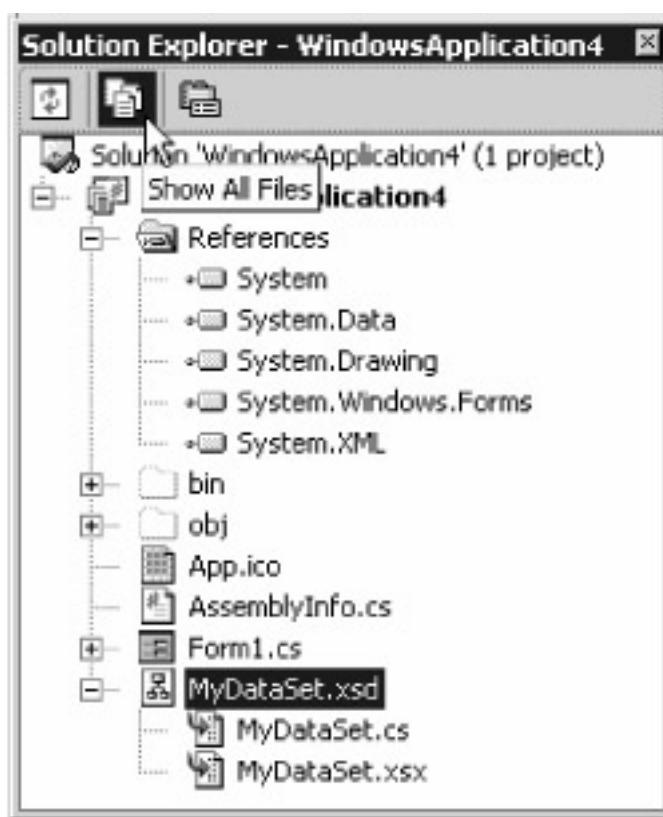
```

Notice that this file contains the details of the columns in the `Customers` table.

Your project also contains a new class file named `MyDataSet.cs`, which contains your strongly typed `DataSet` class. You can view the contents of this file using the Solution Explorer window. You open the Solution Explorer window by selecting **View > Solution Explorer**.

**Note** To view the `MyDataSet.cs` file, click the Show All Files button in the Solution Explorer window.

Next, expand the node beneath `MyDataSet.xsd`. You'll see `MyDataSet.cs`, as shown in [Figure 10.6](#), and a file named `MyDataSet.xsx`, which contains layout information for the visual view of your `DataSet`.



**Figure 10.6:** Viewing all the files using the Solution Explorer window

Go ahead and open `MyDataSet.cs` by double-clicking it in the Solution Explorer window. View the code for this form by selecting `View > Code`. One of the classes declared in that file is `MyDataSet`. This class is derived from the `DataSet` class. You'll use it in the [next section](#) to create a strongly typed `DataSet` object to access the `Customers` table.

## Using a Strongly Typed *DataSet* Class

Once you have your strongly typed `MyDataSet` class, you can create an object of that class using the following code:

```
MyDataSet myDataSet = new MyDataSet();
```

You can also create a strongly typed `DataTable` table object using the `MyDataSet.CustomersDataTable` class and populate it with rows from the `Customers` table. For example, you can set the `Form1_Load()` method of your form to retrieve the `CustomerID`, `CompanyName`, and `Address` column values from the `Customers` table and add them to a `ListView` control named `listview1`. To do this, double-click `Form1.cs` in the Solution Explorer windows, view the code, and set the `Form1_Load()` method as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Data.SqlClient.SqlCommand mySqlCommand =
        sqlConnection1.CreateCommand();
    mySqlCommand.CommandText =
        "SELECT CustomerID, CompanyName, Address " +
        "FROM Customers " +
        "WHERE CustomerID = 'ALFKI'";
    System.Data.SqlClient.SqlDataAdapter mySqlDataAdapter =
        new System.Data.SqlClient.SqlDataAdapter();
    mySqlDataAdapter.SelectCommand = mySqlCommand;
    MyDataSet myDataSet = new MyDataSet();
    sqlConnection1.Open();
    mySqlDataAdapter.Fill(myDataSet, "Customers");
    sqlConnection1.Close();
    MyDataSet.CustomersDataTable myDataTable = myDataSet.Customers;
    foreach (MyDataSet.CustomersRow myDataRow in myDataTable.Rows)
    {
```

```
listView1.Items.Add(myDataRow.CustomerID);  
listView1.Items.Add(myDataRow.CompanyName);  
listView1.Items.Add(myDataRow.Address);  
}  
}
```

The `myDataRow.CustomerID` property returns the value for the `CustomerID` column, and so on. Compile and run your form in one step by selecting `Debug > Start Without Debugging`. [Figure 10.7](#) shows the running form.



**Figure 10.7:** The running form

**Note** The `MyDataSet` class contains a number of methods that allow you to modify the rows stored in a `MyDataSet` object. These methods include `NewCustomersRow()`, `AddCustomersRow()`, `FindByCustomerID()`, and `RemoveCustomersRow()`. You can also check if a column value contains a null value using methods such as `IsContactNameNull()`, and you can set a column to null using methods such as `SetContactNameNull()`. You'll learn how to use these methods in [Chapter 11](#).

## Creating a *DataAdapter* Object Using Visual Studio .NET

In this section, you'll learn how to create a *DataAdapter* using Visual Studio .NET.

**Note** You'll find a completed VS .NET project for this section in the *DataAdapter* directory. You can open this project in VS .NET by selecting File > Open > Project and opening the *WindowsApplication4.csproj* file. You'll need to change the *ConnectionString* property of the *sqlConnection1* object to connect to your database. You can also follow along with the instructions in this section by copying the *DataReader* directory to another directory and using that project as your starting point.

Open your form by double-clicking *Form1.cs* in the Solution Explorer window. Next, create a *SqlDataAdapter* object by dragging a *SqlDataAdapter* object from the Data tab of the Toolbox to your form. When you drag a *SqlDataAdapter* object to your form, you start the Data Adapter Configuration Wizard, as shown in [Figure 10.8](#).



**Figure 10.8:** The Data Adapter Configuration Wizard

Click the Next button to continue.

You now select the database connection you want to use, or you can create a new one. Pick your connection to the Northwind database (or create a new connection if you don't have an existing one), as shown in [Figure 10.9](#).

**Figure 10.9:** Choosing your data connection

Click the Next button to continue.

Next, you set your query type to "Use SQL statements." As you can see from [Figure 10.10](#), you set your query type to use SQL statements, create new stored procedures, or use existing stored procedures. The SQL statements or stored procedures are then used in the `SelectCommand`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties of your `SqlDataAdapter` object. You'll learn about the latter three properties in [Chapter 11](#); they are used to insert, update, and delete rows.

**Figure 10.10:** Choosing your query type

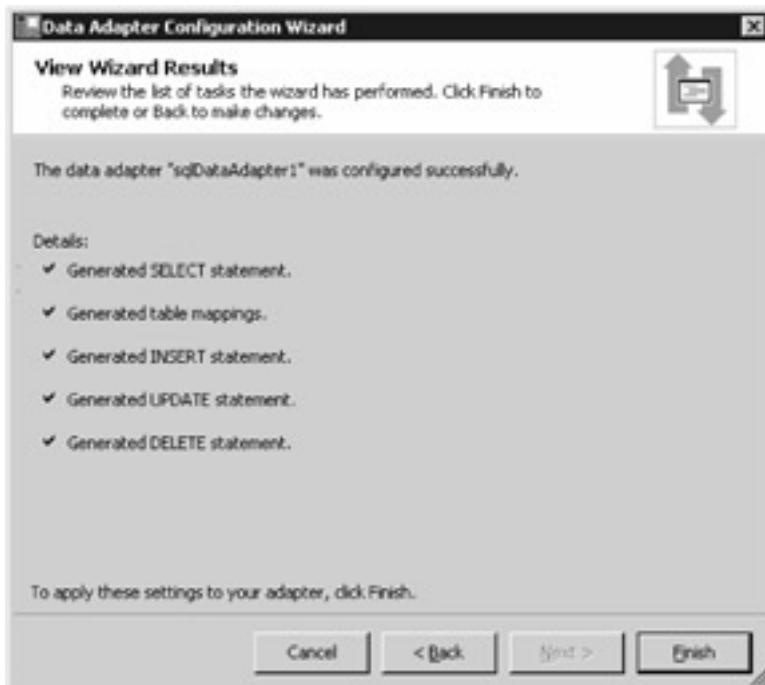
Make sure you've picked Use SQL statements, and click the Next button to continue.

You now generate the SQL statements for your `SqlDataAdapter`. You can either enter a `SELECT` statement directly by typing it or you can press the Query Builder button to build your `SELECT` statement visually. Enter the `SELECT` statement, as shown in [Figure 10.11](#), and click the Next button to continue.



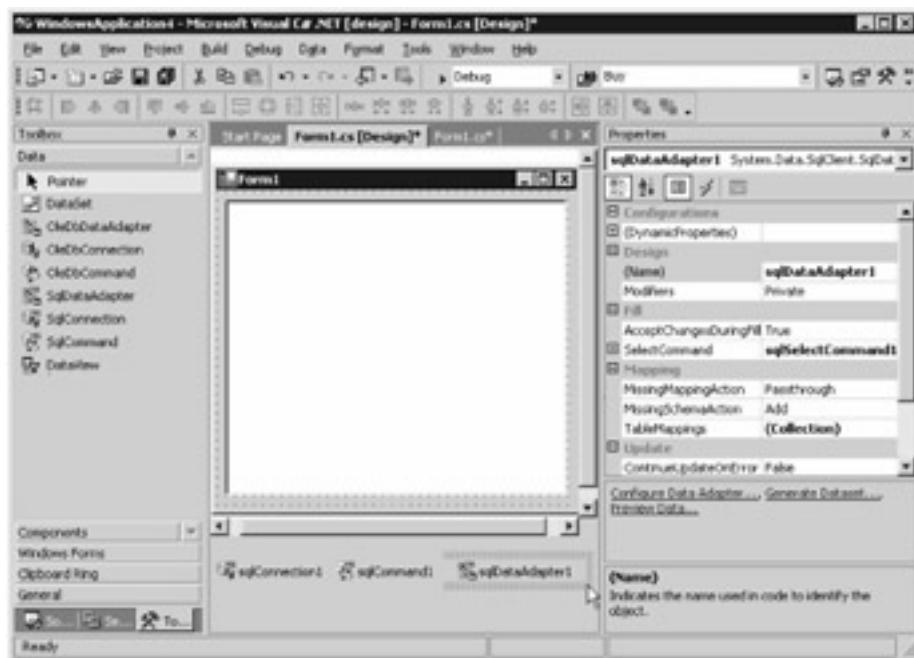
**Figure 10.11:** Generating the SQL statements

The SELECT statement you entered is now used to generate the INSERT, UPDATE, and DELETE statements along with the table mappings. [Figure 10.12](#) shows the final dialog box for the Data Adapter Configuration Wizard.



**Figure 10.12:** Final dialog box for the Data Adapter Configuration Wizard

Click the Finish button to complete the Wizard. A `SqlDataAdapter` object named `sqlDataAdapter1` is now added to the tray beneath your form, as shown in [Figure 10.13](#).



**Figure 10.13:** The new `SqlDataAdapter` object in the tray

### Warning

You need to set the `Connection` property of the `SelectCommand` in your `sqlDataAdapter1` object to your `Connection` object before the `DataAdapter` can access the database. You do this using the Properties window by drilling down from `SelectCommand` to `Connection`. You then click the drop-down list, select Existing, and select your `SqlConnection` object, which should be named `sqlConnection1`. Also check the `ConnectionString` property of your `SqlConnection` object to make sure it is set to connect to your Northwind database. If you don't do this step, you'll get an error stating that your `SqlDataAdapter` object hasn't been configured properly.

Notice the three links at the bottom of the Properties window for `sqlDataAdapter1`:

- **Configure Data Adapter** This link allows you to re-enter the Wizard to configure your `DataAdapter`.
- **Generate Dataset** This link allows you to generate a `DataSet` object using the information set for your `DataAdapter`. You'll use this link in the [next section](#) to generate a new `DataSet`.
- **Preview Data** This link allows you to preview the data returned by the `SelectCommand` of your `DataAdapter`.

Feel free to examine the code generated by the Wizard in your form for the `sqlDataAdapter1` object. When you're ready, select File ➤ Save All.

### Note

Don't bother running your project yet because you'll add a `DataSet` that will be populated using your `DataAdapter` in the [next section](#).

## Creating a *DataSet* Object Using Visual Studio .NET

In this section, you'll learn how to create a *DataSet* using Visual Studio .NET.

**Note** You'll find a completed VS .NET example project for this section in the *DataSet* directory. You can open this project in VS .NET by selecting File > Open > Project and opening the *WindowsApplication4.csproj* file. You can also follow along with the instructions in this section by continuing to modify the copy of the *DataReader* project you used in the [previous section](#).

If you're following along with these instructions, open your copy of the *DataReader* project you modified in the [previous section](#), and open *Form1.cs* by double-clicking it in the Solution Explorer window. To create a *DataSet* object, you can perform either one of the following:

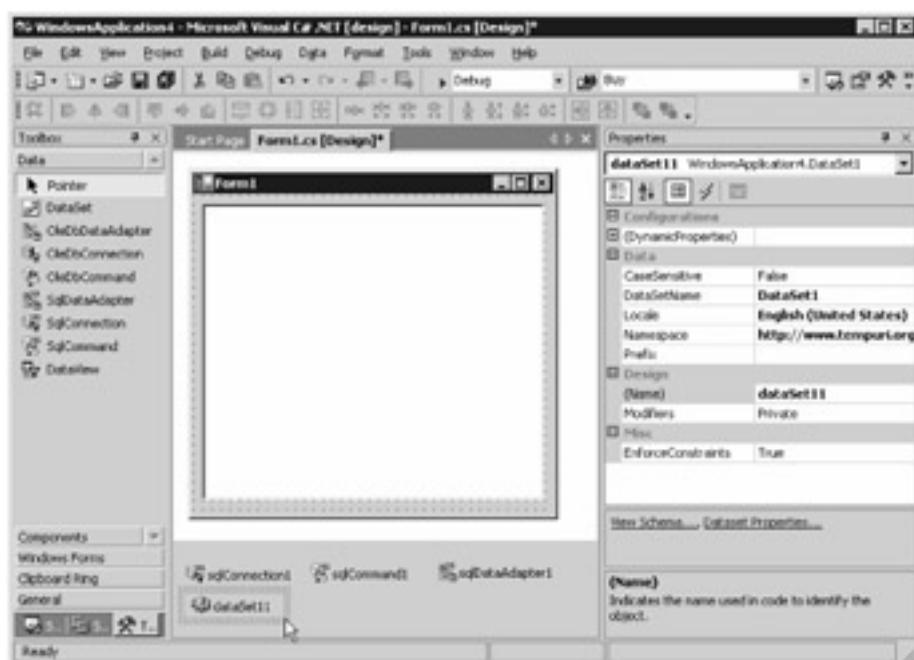
- Drag a *DataSet* object from the Data tab of the Toolbox to your form, and add code to your form to fill it using the `Fill()` method of a *DataAdapter* object.
- Click the Generate Dataset link at the bottom of the Properties window of your *DataAdapter*. You can see this link in [Figure 10.13](#).

You'll use the second step, so go ahead and click the Generate Dataset link. The Generate Dataset dialog box is then displayed, as shown in [Figure 10.14](#).



**Figure 10.14:** The Generate Dataset dialog box

Click the OK button to continue. The new *DataSet* object named *dataSet1* is added to the tray beneath your form, as shown in [Figure 10.15](#).



**Figure 10.15:** The new DataSet object in the tray

Your next step is to set the `Form1_Load()` method of your form as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    sqlConnection1.Open();
    sqlDataAdapter1.Fill(dataSet11, "Products");
    sqlConnection1.Close();
    System.Data.DataTable myDataTable =
        dataSet11.Tables["Products"];
    foreach (System.Data.DataRow myDataRow in myDataTable.Rows)
    {
        listView1.Items.Add(myDataRow[ "ProductID" ].ToString());
        listView1.Items.Add(myDataRow[ "ProductName" ].ToString());
        listView1.Items.Add(myDataRow[ "UnitPrice" ].ToString());
    }
}
```

**Note** Remember, to view the code of your form, you select View > Code. You then replace the `Form1_Load()` method with the previous code.

You can then compile and run your form. [Figure 10.16](#) shows the running form.

1	Chai	18	2	Chang	19
3	Aniseed Syrup	10	4	Chef Anton...	22
5	Chef Anto...	21.35			

**Figure 10.16:** The running form

Team LiB

◀ PREVIOUS    NEXT ▶

## Summary

In this chapter, you learned the details of using `DataSet` objects to store results returned from the database. `DataSet` objects allow you to store a copy of the tables and rows from the database, and you can work with that local copy while disconnected from the database. Unlike managed provider objects such as `SqlDataReader` objects, `DataSet` objects are generic and therefore work with any database. `DataSet` objects also allow you to read rows in any order and modify rows.

You also learned the details of using a `DataAdapter` object to read rows from the database into a `DataSet` object. The `DataAdapter` is part of the managed provider classes, and there are three `DataAdapter` classes: `SqlDataAdapter`, `OleDbDataAdapter`, and `OdbcDataAdapter`.

You use a `DataAdapter` object to move rows between your `DataSet` object and the database, and to synchronize any changes you make to your locally stored rows with the database. For example, you can read rows from the database into a `DataSet` through a `DataAdapter`, modify those rows in your `DataSet`, and then push those changes to the database through your `DataAdapter` object.

In [Chapter 11](#), you'll see how to make changes to the rows in a `DataSet` and then push those changes to the database.

## Chapter 11: Using *DataSet* Objects to Modify Data

### Overview

In [chapter 10](#), you saw how to use a `DataSet` to store a copy of the rows retrieved from the database. In this chapter, you'll learn how to modify the rows in a `DataSet`, and then push those changes to the database via a `DataAdapter`.

Featured in this chapter:

- The `DataTable`, `DataRow`, and  `DataColumn` classes
- Adding restrictions to `DataTable` and  `DataColumn` objects
- Finding, filtering, and sorting `DataRow` objects in a `DataTable`
- Modifying rows in a `DataTable` and pushing those changes to the database
- [Using stored procedures to add, modify, and remove rows from the database](#)
- Using a `CommandBuilder` object to automatically generate SQL statements
- Exploring the `DataAdapter` and `DataTable` events
- Dealing with update failures
- Using transactions with a `DataSet`
- Modifying data using a strongly typed `DataSet`

## The *DataTable* Class

You use an object of the `DataTable` class to represent a table. You can also store multiple `DataTable` objects in a `DataSet`. [Table 11.1](#) shows some of the `DataTable` properties.

**Table 11.1: `DataTable` PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
<code>CaseSensitive</code>	<code>bool</code>	Gets or sets a <code>bool</code> value that indicates whether string comparisons within <code>DataTable</code> objects are case-sensitive.
<code>ChildRelations</code>	<code>DataRelationCollection</code>	Gets the collection of relations ( <code>DataRelationCollection</code> ) that allows navigation from a parent table to a child table. A <code>DataRelationCollection</code> consists of <code>DataRelation</code> objects.
<code>Columns</code>	<code> DataColumnCollection</code>	Gets the collection of columns ( <code>DataColumnCollection</code> ) that contains <code>DataColumn</code> objects that represent the columns in the <code>DataTable</code> object.
<code>Constraints</code>	<code>ConstraintCollection</code>	Gets the collection of constraints ( <code>ConstraintCollection</code> ) that contains <code>Constraint</code> objects that represent primary key ( <code>UniqueConstraint</code> ) or foreign key constraints ( <code>ForeignKeyConstraint</code> ) in the <code>DataTable</code> object.
<code>DataSet</code>	<code>DataSet</code>	Gets the <code>DataSet</code> to which the <code>DataTable</code> belongs.
<code>HasErrors</code>	<code>bool</code>	Returns a <code>bool</code> value that indicates whether any of the rows in the <code>DataTable</code> have errors.
<code>PrimaryKey</code>	<code>DataColumn[ ]</code>	Gets or sets an array of <code>DataColumn</code> objects that are the primary keys for the <code>DataTable</code> .
<code>Rows</code>	<code>DataRowCollection</code>	Gets the collection of rows ( <code>DataRowCollection</code> ) that contains the <code>DataRow</code> objects stored in the <code>DataTable</code> .
<code>TableName</code>	<code>string</code>	Gets or sets the name of the <code>DataTable</code> object.

[Table 11.2](#) shows some of the `DataTable` methods.

**Table 11.2: `DataTable` METHODS**

METHOD	RETURN TYPE	DESCRIPTION
<code>AcceptChanges()</code>	<code>void</code>	Commits all the changes made to the <code>DataTable</code> object since it was loaded or since the last time the <code>AcceptChanges()</code> method was called.
<code>Clear()</code>	<code>void</code>	Removes all rows from the <code>DataTable</code> object.

Clone()	DataTable	Clones the structure of the DataTable object and returns that clone.
Compute()	object	Computes the given expression on the current rows that pass the filter criteria.
GetChanges()	DataTable	Overloaded. Returns a copy of the DataTable object since it was last loaded or since the last time the AcceptChanges() method was called.
GetErrors()	DataRow[]	Overloaded. Gets a copy of all the DataRow objects that have errors.
LoadDataRow()	DataRow	Finds and updates a specified DataRow object. If no matching object is found, a new row is created using the specified values.
NewRow()	DataRow	Creates a new DataRow object in the DataTable.
RejectChanges()	void	Undoes all the changes made to the DataTable object since it was created or since the last time the AcceptChanges() method was called.
Select()	DataRow[]	Overloaded. Returns the array of DataRow objects stored in the DataTable that match the specified filter string. You can also pass a string containing details on how to sort the DataRow objects.

[Table 11.3](#) shows some of the DataTable events.

### Table 11.3: DataTable EVENTS

EVENT	EVENT HANDLER	DESCRIPTION
ColumnChanging	DataColumnChangeEventHandler	Fires <i>before</i> a changed DataColumn value is committed in a DataRow.
ColumnChanged	DataColumnChangeEventHandler	Fires <i>after</i> a changed DataColumn value is committed in a DataRow.
RowChanging	DataRowChangeEventHandler	Fires <i>before</i> a changed DataRow is committed in a DataTable.
RowChanged	DataRowChangeEventHandler	Fires <i>after</i> a changed DataRow is committed in a DataTable.
RowDeleting	DataRowChangeEventHandler	Fires <i>before</i> a DataRow is deleted from a DataTable.
RowDeleted	DataRowChangeEventHandler	Fires <i>after</i> a DataRow is deleted from a DataTable.

## The *DataRow* Class

You use an object of the *DataRow* class to represent a row. You can also store multiple *DataRow* objects in a *DataTable*. [Table 11.4](#) shows some of the *DataRow* properties.

**Table 11.4: *DataRow* PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
HasErrors	bool	Returns a bool value that indicates whether any of the <i>DataTable</i> objects in the <i>DataRow</i> have errors.
ItemArray	object [ ]	Gets or sets all the <i>DataTable</i> objects in the <i>DataRow</i> .
RowState	DataRowState	Gets the current state of the <i>DataRow</i> . The state can be <i>Added</i> , <i>Deleted</i> , <i>Detached</i> , <i>Modified</i> , or <i>Unchanged</i> . The state depends in the operation performed on the <i>DataRow</i> and whether the <i>AcceptChanges ( )</i> method has been called to commit the changes.
Table	DataTable	Gets the <i>DataTable</i> object to which the <i>DataRow</i> belongs.

*The row has been created but isn't part of a *DataRowCollection* object; a *DataRow* is in this state immediately after it has been created and before it is added to a collection, or if it has been removed from a collection.*

[Table 11.5](#) shows some of the *DataRow* methods.

**Table 11.5: *DataRow* METHODS**

METHOD	RETURN TYPE	DESCRIPTION
AcceptChanges ( )	void	Commits all the changes made to the <i>DataRow</i> object since it was loaded or since the last time the <i>AcceptChanges ( )</i> method was called.
BeginEdit ( )	void	Starts an edit for the <i>DataRow</i> object.
CancelEdit ( )	void	Cancels an edit for the <i>DataRow</i> object and restores it to the original state.
ClearErrors ( )	void	Clears any errors for the <i>DataRow</i> object.
Delete ( )	void	Deletes the <i>DataRow</i> object.
EndEdit ( )	void	Stops an edit for the <i>DataRow</i> object and commits the change.
GetChildRows ( )	DataRow [ ]	Overloaded. Returns an array of <i>DataRow</i> objects that contain the child rows using the specified <i>DataRelation</i> object.
GetColumnError ( )	string	Overloaded. Returns the description of the error for the specified <i>DataTable</i> object.
GetColumnsInError ( )	DataTable [ ]	Returns an array of <i>DataTable</i> objects that have errors.
GetParentRow ( )	DataRow	Overloaded. Returns a <i>DataRow</i> object that contains the parent row using the specified <i>DataRelation</i> object.

<code>GetParentRows()</code>	<code>DataRow[ ]</code>	Overloaded. Returns an array of <code>DataRow</code> objects that contain the parent rows using the specified <code>DataRelation</code> object.
<code>IsNull()</code>	<code>bool</code>	Overloaded. Returns a <code>bool</code> value that indicates whether the specified <code> DataColumn</code> object contains a null value.
<code>RejectChanges()</code>	<code>void</code>	Undoes all changes made to the <code>DataRow</code> object since the <code>AcceptChanges()</code> method was called.
<code>SetNull()</code>	<code>void</code>	Sets the specified <code> DataColumn</code> object to a null value.
<code>SetParentRow()</code>	<code>void</code>	Overloaded. Sets the parent row to the specified <code>DataRow</code> object.

**Team LiB****◀ PREVIOUS****NEXT ▶**

## The *DataColumn* Class

You use an object of the *DataColumn* class to represent a column. You can also store multiple *DataColumn* objects in a *DataRow*. [Table 11.6](#) shows some of the *DataColumn* properties.

**Table 11.6: *DataColumn* PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
AllowDBNull	bool	Gets or sets a bool value that indicates whether null values are allowed in this <i>DataColumn</i> object. The default is <code>true</code> .
AutoIncrement	bool	Gets or sets a bool value that indicates whether the <i>DataColumn</i> object automatically increments the value of the column for new rows. The default is <code>false</code> .
AutoIncrementSeed	long	Gets or sets the starting value for the <i>DataColumn</i> object. Applies only when the <code>AutoIncrement</code> property is set to <code>true</code> . The default is 0.
AutoIncrementStep	long	Gets or sets the increment used. Applies only when the <code>AutoIncrement</code> property is set to <code>true</code> . The default is 1.
Caption	string	Gets or sets the caption for the column. The caption for the column is shown in Windows forms. The default is null.
ColumnName	string	Gets or sets the name of the <i>DataColumn</i> object.
ColumnMapping	MappingType	Gets or sets the <code>MappingType</code> of the <i>DataColumn</i> object. This determines how a <i>DataColumn</i> is saved in an XML document using the <code>WriteXml()</code> method.
DataType	Type	Gets or sets the .NET data type used to represent the column value stored in the <i>DataColumn</i> object. This can be <code>Boolean</code> , <code>Byte</code> , <code>Char</code> , <code>DateTime</code> , <code>Decimal</code> , <code>Double</code> , <code>Int16</code> , <code>Int32</code> , <code>Int64</code> , <code>SByte</code> , <code>Single</code> , <code>String</code> , <code> TimeSpan</code> , <code>UInt16</code> , or <code>UInt64</code> .
DefaultValue	object	Gets or sets the default value for the <i>DataColumn</i> when new rows are created. When <code>AutoIncrement</code> is set to <code>true</code> , <code>DefaultValue</code> is not used.
MaxLength	int	Gets or sets the maximum length of text that may be stored in a <i>DataColumn</i> object. The default is -1.
Ordinal	int	Gets the numeric position of the <i>DataColumn</i> object (0 is the first object).
ReadOnly	bool	Gets or sets a bool value that indicates whether the <i>DataColumn</i> object can be changed once it has been added to a <i>DataRow</i> . The default is <code>false</code> .
Table	DataTable	Gets the <i>DataTable</i> to which the <i>DataColumn</i> object belongs.
Unique	bool	Gets or sets a bool value that indicates whether the <i>DataColumn</i> values in each <i>DataRow</i> object must be unique. The default is <code>false</code> .

You'll see the use of some of these properties, methods, and events later in this chapter.



## Adding Restrictions to *DataTable* and *DataColumn* Objects

As you know, a *DataSet* object is used to store a copy of a subset of the database. For example, you can store a copy of the rows from database tables into a *DataSet*, with each table represented by a *DataTable* object. A *DataTable* stores columns in  *DataColumn* objects.

In addition to storing rows retrieved from a database table, you can also add restrictions to a *DataTable* and its *DataColumn* objects. This allows you to model the same restrictions placed on the database tables and columns in your *DataTable* and *DataColumn* objects. For example, you can add the following constraints to a *DataTable*:

- Unique
- Primary key
- Foreign key

In addition, you can add the following restrictions to a *DataColumn*:

- Whether the column can accept a null value—which you store in the *AllowDBNull* property of the *DataColumn*.
- Any auto-increment information—which you store in the *AutoIncrement*, *AutoIncrementSeed*, and *AutoIncrementStep* properties of the *DataColumn*. You set these properties when adding rows to a *DataTable* with a corresponding database table that contains an identity column. The *ProductID* column of the *Products* table is an example of an identity column.

**Note** ADO.NET will not automatically generate values for identity columns in a new row. Only the database can do that. You must read the generated identity value for the column from the database. You'll see how to do that later in the sections "[Retrieving New Identity Column Values](#)" and "[Using Stored Procedures to Add, Modify, and Remove Rows from the Database](#)." Also, if your database table contains columns that are assigned a default value, you should read that value from the database. This is better than setting the *DefaultValue* property of a *DataColumn* because if the default value set in the database table definition changes, you can pick up the new value from the database rather than having to change your code.

- The maximum length of a string or character column value—which you store in the *MaxLength* property of the *DataColumn*.
- Whether the column is read-only—which you store in the *ReadOnly* property of the *DataColumn*.
- Whether the column is unique—which you store in the *Unique* property of the *DataColumn*.

By adding these restrictions up front, you prevent bad data from being added to your *DataSet* to begin with. This helps reduce the errors when attempting to push changes in your *DataSet* to the database. If a user of your program attempts to add data that violates a restriction, they'll cause an exception to be thrown. You can then catch the exception in your program and display a message with the details. The user can then change the data they were trying to add and fix the problem.

You also need to define a primary key before you can find, filter, and sort *DataRow* objects in a *DataTable*. You'll learn how to do that later in the section "[Finding, Filtering, and Sorting Rows in a DataTable](#)."

**Tip** Adding constraints causes a performance degradation when you call the *Fill()* method of a *DataAdapter*. This is because the retrieved rows are checked against your constraints before they are added to your *DataSet*. You should therefore set the *EnforceConstraints* property of your *DataSet* to *false* before calling the *Fill()* method. You then set *EnforceConstraints* back to the default of *true* after the call to *Fill()*.

You can use one of following ways to add restrictions to *DataTable* and *DataColumn* objects:

- Add the restrictions yourself by setting the properties of your *DataTable* and *DataColumn* objects. This results in the fastest executing code.

- Call the `FillSchema()` method of your `DataAdapter` to copy the schema information from the database to your `DataSet`. This populates the properties of the `DataTable` objects and their `DataColumn` objects automatically. Although simple to call, the `FillSchema()` method takes a relatively long time to read the schema information from the database and you should avoid using it.

You'll learn the details of both these techniques in the following sections.

## Adding the Restrictions Yourself

You can add restrictions to your `DataTable` and `DataColumn` objects yourself using the properties of the `DataTable` and `DataColumn` objects.

For example, assume you have a `DataSet` object named `myDataSet` that contains three `DataTable` objects named `Products`, `Orders`, and `Order Details` that have been populated using the following code:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT ProductID, ProductName " +
    "FROM Products;" +
    "SELECT OrderID " +
    "FROM Orders;" +
    "SELECT OrderID, ProductID, UnitPrice " +
    "FROM [Order Details];";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet);
mySqlConnection.Close();
myDataSet.Tables["Table"].TableName = "Products";
myDataSet.Tables["Table1"].TableName = "Orders";
myDataSet.Tables["Table2"].TableName = "Order Details";
```

The primary key for the `Products` table is the `ProductID` column; the primary key for the `Orders` table is the `OrderID` column; and the primary key for the `Order Details` table is made up of both the `OrderID` and `ProductID` columns.

**Note** You must include all the columns of a database table's primary key in your query if you want to define a primary key on those columns in your `DataTable`.

In the following sections, you'll see how to

- Add constraints to the `Products`, `Orders`, and `Order Details` `DataTable` objects.
- Restrict the values placed in the `DataColumn` objects of the `Products` `DataTable`.

## Adding Constraints to `DataTable` Objects

In this section, you'll see how to add constraints to `DataTable` objects. Specifically, you'll see how to add primary key constraints to the `Products`, `Orders`, and `Order Details` `DataTable` objects. A primary key constraint is actually implemented as a unique constraint. You'll also see how to add foreign key constraints from the `Order Details` to the `Products` and `Orders` `DataTable` objects.

Constraints are stored in a `ConstraintCollection` object that stores `Constraint` objects. You access the `ConstraintCollection` using the `DataTable` object's `Constraints` property. To add a new `Constraint` object to `ConstraintCollection`, you call the `Add()` method through the `Constraints` property. The `Add()` method allows you to add unique constraints and foreign key constraints to a `DataTable`. Since a primary key constraint is implemented as a unique constraint, you can also use the `Add()` method to add a primary constraint to a `DataTable`. You'll see how to use the `Add()` method

shortly.

You can also add a primary key constraint to a `DataTable` object by setting its `PrimaryKey` property, which you set to an array of `DataColumn` objects that make up the primary key. An array is required because the primary key of a database table can be made up of multiple columns. As you'll see in the examples, this is simpler than using the `Add()` method to add a primary key constraint.

---

### **CALLING THE `Fill()` METHOD OF A `DataAdapter` MORE THAN ONCE**

The `Fill()` method retrieves *all* of the rows from the database table, as specified in your `DataAdapter` object's `SelectCommand` property. If you add a primary key to your `DataTable`, then calling the `Fill()` method more than once will put the retrieved rows in your `DataTable` and throw away any existing rows with matching primary key column values already in your `DataTable`.

If you don't add a primary key to your `DataTable`, then calling the `Fill()` method more than once will simply add all the retrieved rows to your `DataTable` again, duplicating the rows already there.

This is another reason for adding a primary key constraint to your `DataTable` because you don't want duplicate rows.

---

## **Adding a Primary Key to the Products `DataTable`**

Let's take a look at adding a primary key to the `Products` `DataTable`. First, the following example creates a `DataTable` object named `productsDataTable` and sets it to the `Products` `DataTable` retrieved from `myDataSet`:

```
DataTable productsDataTable = myDataSet.Tables["Products"];
```

Now, the primary key for the `Products` database table is the `ProductID` column; therefore, you need to set the `PrimaryKey` property of `productsDataTable` to an array containing the `ProductID`  `DataColumn` object. The following example shows how you do this. It creates an array of `DataColumn` objects named `productsPrimaryKey` and initializes it to the `ProductID` column of `productsDataTable`, then sets the `PrimaryKey` property of `productsDataTable` to the array:

```
DataTable[] productsPrimaryKey =
    new DataColumn[]
    {
        productsDataTable.Columns["ProductID"]
    };
productsDataTable.PrimaryKey = productsPrimaryKey;
```

When you set the `PrimaryKey` property of a `DataTable`, the `AllowDBNull` and `Unique` properties of the `DataColumn` object are automatically changed as follows:

- The `AllowDBNull` property is changed to `false` and indicates that the `DataColumn` cannot accept a null value.
- The `Unique` property is changed to `true` and indicates that the `DataColumn` value in each `DataRow` must be unique.

In the previous example, therefore, the `AllowDBNull` and `Unique` properties of the `ProductID`  `DataColumn` are automatically changed to `false` and `true`, respectively.

## **Adding a Primary Key to the Orders `DataTable`**

The following example sets the `PrimaryKey` property of the `Orders` `DataTable` to the `OrderID`  `DataColumn`:

```
myDataSet.Tables["Orders"].PrimaryKey =
    new DataColumn[]
    {
        myDataSet.Tables["Orders"].Columns["OrderID"]
    };
```

Notice I've used just one statement in this example to make it more concise than the previous example.

You can also use the `Add()` method to add a unique, primary key, or foreign key constraint to a `DataTable`. The `Add()` method is overloaded as follows:

```
void Add(Constraint myConstraint) // adds any constraint
void Add(string constraintName, DataColumn myDataColumn,
    bool isPrimaryKey) // adds a primary key or unique constraint
void Add(string constraintName, DataColumn parentColumn,
    DataColumn childColumn) // adds a foreign key constraint
void Add(string constraintName, DataColumn[] myDataColumn,
    bool isPrimaryKey) // adds a primary key or unique constraint
void Add(string constraintName, DataColumn[] parentColumns,
    DataColumn[] childColumns) // adds a foreign key constraint
```

where

- **`constraintName`** is the name you want to assign to your constraint.
- **`isPrimaryKey`** indicates whether the constraint is a primary key constraint or just a regular unique constraint.

The following example uses the `Add()` method to add a primary key constraint to the `Products` `DataTable`:

```
myDataSet.Tables["Orders"].Constraints.Add(
    "Primary key constraint",
    myDataSet.Tables["Orders"].Columns["OrderID"],
    true
);
```

This example does the same thing as the previous example that added the primary key constraint using the `PrimaryKey` property. Notice the last parameter to the `Add()` method is set to `true`, which indicates the constraint is for a primary key.

Just as an aside, if you have a column that isn't a primary key but is unique, you can add a `UniqueConstraint` object to the `ConstraintsCollection`. For example:

```
UniqueConstraint myUC =
    new UniqueConstraint(myDataTable.Columns["myColumn"]);
myDataTable.Constraints.Add(myUC);
```

## Adding a Primary Key to the OrderDetails DataTable

Let's consider an example of setting the `PrimaryKey` property for the `Order Details` `DataTable`. The primary for the `Order Details` table is made up of the `OrderID` and `ProductID` columns, and the following example sets the `PrimaryKey` property of the `Order Details` `DataTable` to these two columns:

```
myDataSet.Tables["Order Details"].PrimaryKey =
    new DataColumn[]
    {
        myDataSet.Tables["Order Details"].Columns["OrderID"],
        myDataSet.Tables["Order Details"].Columns["ProductID"]
    };
```

The following example uses the `Add()` method to do the same thing:

```
myDataSet.Tables["Order Details"].Constraints.Add(
    "Primary key constraint",
    new DataColumn[]
    {
        myDataSet.Tables["Order Details"].Columns["OrderID"],
        myDataSet.Tables["Order Details"].Columns["ProductID"]
    },
    true
);
```

```
    true
);
```

One thing to keep in mind when adding constraints to a `DataTable` is that it knows only about the rows you store in it; it doesn't know about any other rows stored in the actual database table. To see why this is an issue, consider the following scenario that involves primary keys:

1. You add a primary key constraint to a `DataTable`.
2. You retrieve a *subset* of the rows from a database table and store them in your `DataTable`.
3. You add a new `DataRow` to your `DataTable` with a primary key value not used in the subset of rows retrieved into your `DataTable` in the previous step—but that primary key value *is* already used in a row in the database table. Your new `DataRow` is added without any problem to the `DataTable` even though you added a primary key constraint to your `DataTable` in step 1. Your new `DataRow` is added successfully because the `DataTable` knows *only* about the rows stored in it, *not* the other rows stored in the database table that were not retrieved in step 2.
4. You attempt to push the new `DataRow` to the database, but you get a `SqlException` that states you've violated the primary key constraint in the database table. This is because a row in the database table already uses the primary key value.

You need to keep this issue in mind when adding rows to a `DataTable`, which you'll see how to do shortly.

That wraps up adding the primary key constraints to the `DataTable` objects. Next, you'll see how to add foreign key constraints.

## Adding Foreign Key Constraints to the Order Details `DataTable`

In this section, you'll see how to add a foreign key constraint to the `Order Details` `DataTable`. To do this, you use the `Add()` method through the `Constraints` property of the `DataTable`.

The following example adds a foreign key constraint from the `OrderID`  `DataColumn` of the `Order Details` `DataTable` to the `OrderID`  `DataColumn` of the `Orders` `DataTable`:

```
ForeignKeyConstraint myFKC = new ForeignKeyConstraint(
    myDataSet.Tables["Orders"].Columns["OrderID"],
    myDataSet.Tables["Order Details"].Columns["OrderID"]
);
myDataSet.Tables["Order Details"].Constraints.Add(myFKC);
```

**Note** Notice that the parent  `DataColumn` (`OrderID` of `Orders`) is specified before the child  `DataColumn` (`OrderID` of `Order Details`).

The next example adds a foreign key constraint from the `ProductID`  `DataColumn` of the `Order Details` `DataTable` to the `ProductID`  `DataColumn` of the `Products` `DataTable`:

```
myDataSet.Tables["Order Details"].Constraints.Add(
    "Foreign key constraint to ProductID DataColumn of the " +
    "Products DataTable",
    myDataSet.Tables["Order Details"].Columns["ProductID"],
    myDataSet.Tables["Products"].Columns["ProductID"]
);
```

That wraps up adding constraints to the `DataTable` objects. Next, you'll see how to add restrictions to `DataColumn` objects.

## Adding Restrictions to `DataColumn` Objects

In this section, you'll see how to add restrictions to the `DataColumn` objects stored in a `DataTable`. Specifically, you'll see how to set the `AllowDBNull`, `AutoIncrement`, `AutoIncrementSeed`, `AutoIncrementStep`, `ReadOnly`, and `Unique` properties of the `ProductID`  `DataColumn` of the `Products` `DataTable`. You'll also see how to set the `MaxLength`

property of the `ProductName` `DataColumn` of the `Products` `DataTable`.

The `ProductID` column of the `Products` database table is an identity column. The seed is the initial value and the step is the increment added to the last number and they are both set to 1 for `ProductID`. The `ProductID` identity values are therefore 1, 2, 3, and so on.

**Tip** When you set the `AutoIncrementSeed` and `AutoIncrementStep` properties for a `DataColumn` that corresponds to a database identity column, you should always set them both to -1. That way, when you call the `Fill()` method, ADO.NET will automatically figure out what values to set the `AutoIncrementSeed` and `AutoIncrementStep` to, based on the values retrieved from the database, and you don't have to figure out these values yourself.

The following code sets the properties of the `ProductID` `DataColumn`:

```
DataTable productIDDatatable = 
    myDataSet.Tables["Products"].Columns["ProductID"];
productIDDatatable.AllowDBNull = false;
productIDDatatable.AutoIncrement = true;
productIDDatatable.AutoIncrementSeed = -1;
productIDDatatable.AutoIncrementStep = -1;
productIDDatatable.ReadOnly = true;
productIDDatatable.Unique = true;
```

The next example sets the `MaxLength` property of the `ProductName` `DataColumn` to 40. This stops you from setting the column value for `ProductName` to a string greater than 40 characters in length:

```
myDataSet.Tables["Products"].Columns["ProductName"].MaxLength = 40;
```

[Listing 11.1](#) uses the code examples shown in this section and the previous one. Notice this program also displays the `ColumnName` and `DataType` properties of the `DataColumn` objects in each `DataTable`. The `ColumnName` property contains the name of the `DataColumn`, and the `DataType` contains the .NET data type used to represent the column value stored in the `DataColumn`.

## **Listing 11.1: ADDRESTRICTIONS.CS**

---

```
/*
 AddRestrictions.cs illustrates how to add constraints to
 DataTable objects and add restrictions to DataColumn objects
 */

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

class AddRestrictions
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT ProductID, ProductName " +
            "FROM Products;" +
            "SELECT OrderID " +
            "FROM Orders;" +
            "SELECT OrderID, ProductID, UnitPrice " +
            "FROM [Order Details];";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
```

```
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet);
mySqlConnection.Close();
myDataSet.Tables["Table"].TableName = "Products";
myDataSet.Tables["Table1"].TableName = "Orders";
myDataSet.Tables["Table2"].TableName = "Order Details";

// set the PrimaryKey property for the Products DataTable
// to the ProductID column
DataTable productsDataTable = myDataSet.Tables["Products"];
DataColumn[] productsPrimaryKey =
    new DataColumn[]
    {
        productsDataTable.Columns["ProductID"]
    };
productsDataTable.PrimaryKey = productsPrimaryKey;

// set the PrimaryKey property for the Orders DataTable
// to the OrderID column
myDataSet.Tables["Orders"].PrimaryKey =
    new DataColumn[]
    {
        myDataSet.Tables["Orders"].Columns["OrderID"]
    };

// set the PrimaryKey property for the Order Details DataTable
// to the OrderID and ProductID columns
myDataSet.Tables["Order Details"].Constraints.Add(
    "Primary key constraint on the OrderID and ProductID columns",
    new DataColumn[]
    {
        myDataSet.Tables["Order Details"].Columns["OrderID"],
        myDataSet.Tables["Order Details"].Columns["ProductID"]
    },
    true
);

// add a foreign key constraint on the OrderID column
// of Order Details to the OrderID column of Orders
ForeignKeyConstraint myFKC = new ForeignKeyConstraint(
    myDataSet.Tables["Orders"].Columns["OrderID"],
    myDataSet.Tables["Order Details"].Columns["OrderID"]
);
myDataSet.Tables["Order Details"].Constraints.Add(myFKC);
// add a foreign key constraint on the ProductID column
// of Order Details to the ProductID column of Products
myDataSet.Tables["Order Details"].Constraints.Add(
    "Foreign key constraint to ProductID DataColumn of the " +
    "Products DataTable",
    myDataSet.Tables["Products"].Columns["ProductID"],
    myDataSet.Tables["Order Details"].Columns["ProductID"]
);

// set the AllowDBNull, AutoIncrement, AutoIncrementSeed,
// AutoIncrementStep, ReadOnly, and Unique properties for
// the ProductID DataColumn of the Products DataTable
DataColumn productIDDataColumn =
    myDataSet.Tables["Products"].Columns["ProductID"];
```

```

productIDDataColumn.AllowDBNull = false;
productIDDataColumn.AutoIncrement = true;
productIDDataColumn.AutoIncrementSeed = -1;
productIDDataColumn.AutoIncrementStep = -1;
productIDDataColumn.ReadOnly = true;
productIDDataColumn.Unique = true;

// set the MaxLength property for the ProductName DataColumn
// of the Products DataTable
myDataSet.Tables["Products"].Columns["ProductName"].MaxLength = 40;

// display the details of the DataColumn objects for
// the DataTable objects
foreach (DataTable myDataTable in myDataSet.Tables)
{
    Console.WriteLine("\n\nReading from the " +
        myDataTable + "DataTable:\n");

    // display the primary key
    foreach (DataColumn myPrimaryKey in myDataTable.PrimaryKey)
    {
        Console.WriteLine("myPrimaryKey = " + myPrimaryKey);
    }

    // display some of the details for each column
    foreach (DataColumn myDataColumn in myDataTable.Columns)
    {
        Console.WriteLine("\nmyDataColumn.ColumnName = " +
            myDataColumn.ColumnName);
        Console.WriteLine("myDataColumn.DataType = " +
            myDataColumn.DataType);

        Console.WriteLine("myDataColumn.AllowDBNull = " +
            myDataColumn.AllowDBNull);
        Console.WriteLine("myDataColumn.AutoIncrement = " +
            myDataColumn.AutoIncrement);
        Console.WriteLine("myDataColumn.AutoIncrementSeed = " +
            myDataColumn.AutoIncrementSeed);
        Console.WriteLine("myDataColumn.AutoIncrementStep = " +
            myDataColumn.AutoIncrementStep);
        Console.WriteLine("myDataColumn.MaxLength = " +
            myDataColumn.MaxLength);
        Console.WriteLine("myDataColumn.ReadOnly = " +
            myDataColumn.ReadOnly);
        Console.WriteLine("myDataColumn.Unique = " +
            myDataColumn.Unique);
    }
}
}
}
}

```

---

The output from this program is as follows:

Reading from the Products DataTable:

myPrimaryKey = ProductID

myDataColumn.ColumnName = ProductID  
 myDataColumn.DataType = System.Int32

## Adding Restrictions to DataTable and DataColumn Objects

```
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = True
myDataColumn.AutoIncrementSeed = -1
myDataColumn.AutoIncrementStep = -1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = True
myDataColumn.Unique = True

myDataColumn.ColumnName = ProductName
myDataColumn.DataType = System.String
myDataColumn.AllowDBNull = True
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = 40
myDataColumn.ReadOnly = False
myDataColumn.Unique = False
```

Reading from the Orders DataTable:

```
myPrimaryKey = OrderID
myDataColumn.ColumnName = OrderID
myDataColumn.DataType = System.Int32
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = False
myDataColumn.Unique = True
```

Reading from the Order Details DataTable:

```
myPrimaryKey = OrderID
myPrimaryKey = ProductID

myDataColumn.ColumnName = OrderID
myDataColumn.DataType = System.Int32
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = False
myDataColumn.Unique = False

myDataColumn.ColumnName = ProductID
myDataColumn.DataType = System.Int32
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = False
myDataColumn.Unique = False

myDataColumn.ColumnName = UnitPrice
myDataColumn.DataType = System.Decimal
```

```

myDataColumn.AllowDBNull = True
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = False
myDataColumn.Unique = False

```

## Adding Restrictions by Calling the *DataAdapter* Object's *FillSchema()* Method

Instead of adding restrictions yourself, you can add them by calling the `FillSchema()` method of your `DataAdapter`. The `FillSchema()` method does the following:

- Copies the schema information from the database.
- Creates `DataTable` objects in your `DataSet` if they don't already exist.
- Adds the constraints to the `DataTable` objects.
- Sets the properties of the  `DataColumn` objects appropriately.

The properties of the  `DataColumn` objects set by `FillSchema()` include the following:

- The  `DataColumn` name—which is stored in the `ColumnName` property.
- The  `DataColumn` .NET data type—which is stored in the `DataType` property.
- The maximum length of a variable length data type—which is stored in the `MaxLength` property.
- Whether the  `DataColumn` can accept a null value—which is stored in the `AllowDBNull` property.
- Whether the  `DataColumn` value must be unique—which is stored in the `Unique` property.
- Any auto-increment information—which is stored in the `AutoIncrement`, `AutoIncrementSeed`, and `AutoIncrementStep` properties.

The `FillSchema()` method will also determine whether the  `DataColumn` is part of a primary key and store that information in the `PrimaryKey` property of the `DataTable`.

**Warning** `FillSchema()` does not automatically add `ForeignKeyConstraint` objects to the `DataTable` objects. Neither does it retrieve the actual rows from the database; it retrieves only the schema information.

The `FillSchema()` method is overloaded, with the most commonly used version of this method being the following:

```
DataTable[] FillSchema(DataSet myDataSet, SchemaType mySchemaType)
```

where `mySchemaType` specifies how you want to handle any existing schema mappings.

You set `mySchemaType` to one of the constants defined in the `System.Data.SchemaType` enumeration. [Table 11.7](#) shows the constants defined in the `SchemaType` enumeration.

**Table 11.7: SchemaType ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
Mapped	Apply any existing table mappings to the incoming schema and configure the <code>DataSet</code> with the transformed schema. This is the constant you should typically use.

Source

Ignore any table mappings and configure the DataSet without any transformations.

Let's take a look at an example that contains a call to the `FillSchema()` method. Notice the call uses the `SchemaType.Mapped` constant to apply any existing table mappings:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT ProductID, ProductName " +
    "FROM Products;" +
    "SELECT OrderID " +
    "FROM Orders;" +
    "SELECT OrderID, ProductID, UnitPrice " +
    "FROM [Order Details];";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.FillSchema(myDataSet, SchemaType.Mapped);
mySqlConnection.Close();
myDataSet.Tables["Table"].TableName = "Products";
myDataSet.Tables["Table1"].TableName = "Orders";
myDataSet.Tables["Table2"].TableName = "Order Details";
```

The call to `FillSchema()` copies the schema information from the `Products`, `Orders`, and `Order Details` tables to `myDataSet`, setting the `PrimaryKey` property of each `DataTable` and the properties of the `DataColumn` objects appropriately.

[Listing 11.2](#) shows the use of the `FillSchema()` method.

## **Listing 11.2: FILLSHEMA.CS**

---

```
/*
 FillSchema.cs illustrates how to read schema information
 using the FillSchema() method of a DataAdapter object
 */

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

class FillSchema
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT ProductID, ProductName " +
            "FROM Products;" +
            "SELECT OrderID " +
            "FROM Orders;" +
            "SELECT OrderID, ProductID, UnitPrice " +
            "FROM [Order Details];";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
```

```
mySqlDataAdapter.FillSchema(myDataSet, SchemaType.Mapped);
mySqlConnection.Close();
myDataSet.Tables["Table"].TableName = "Products";
myDataSet.Tables["Table1"].TableName = "Orders";
myDataSet.Tables["Table2"].TableName = "Order Details";

// display the details of the DataColumn objects for
// the DataTable objects
foreach (DataTable myDataTable in myDataSet.Tables)
{
    Console.WriteLine("\n\nReading from the " +
        myDataTable + "DataTable:\n");

    // display the primary key
    foreach (DataColumn myPrimaryKey in myDataTable.PrimaryKey)
    {
        Console.WriteLine("myPrimaryKey = " + myPrimaryKey);
    }

    // display the constraints
    foreach (Constraint myConstraint in myDataTable.Constraints)
    {
        Console.WriteLine("myConstraint.IsPrimaryKey = " + ((UniqueConstraint)
myConstraint).IsPrimaryKey);
        foreach (DataColumn myDataColumn in ((UniqueConstraint)
myConstraint).Columns)
        {
            Console.WriteLine("myDataColumn.ColumnName = " +
myDataColumn.ColumnName);
        }
    }

    // display some of the details for each column
    foreach (DataColumn myDataColumn in myDataTable.Columns)
    {
        Console.WriteLine("\nmyDataColumn.ColumnName = " +
            myDataColumn.ColumnName);
        Console.WriteLine("myDataColumn.DataType = " +
            myDataColumn.DataType);

        Console.WriteLine("myDataColumn.AllowDBNull = " +
            myDataColumn.AllowDBNull);
        Console.WriteLine("myDataColumn.AutoIncrement = " +
            myDataColumn.AutoIncrement);
        Console.WriteLine("myDataColumn.AutoIncrementSeed = " +
            myDataColumn.AutoIncrementSeed);
        Console.WriteLine("myDataColumn.AutoIncrementStep = " +
            myDataColumn.AutoIncrementStep);
        Console.WriteLine("myDataColumn.MaxLength = " +
            myDataColumn.MaxLength);
        Console.WriteLine("myDataColumn.ReadOnly = " +
            myDataColumn.ReadOnly);
        Console.WriteLine("myDataColumn.Unique = " +
            myDataColumn.Unique);
    }
}
}
```

The output from this program is as follows:

Reading from the Products DataTable:

```
myPrimaryKey = ProductID
myConstraint.IsPrimaryKey = True
myDataColumn.ColumnName = ProductID

myDataColumn.ColumnName = ProductID
myDataColumn.DataType = System.Int32
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = True
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = True
myDataColumn.Unique = True

myDataColumn.ColumnName = ProductName
myDataColumn.DataType = System.String
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = 40
myDataColumn.ReadOnly = False
myDataColumn.Unique = False
```

Reading from the Orders DataTable:

```
myPrimaryKey = OrderID
myConstraint.IsPrimaryKey = True
myDataColumn.ColumnName = OrderID

myDataColumn.ColumnName = OrderID
myDataColumn.DataType = System.Int32
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = True
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = True
myDataColumn.Unique = True
```

Reading from the Order Details DataTable:

```
myPrimaryKey = OrderID
myPrimaryKey = ProductID
myConstraint.IsPrimaryKey = True
myDataColumn.ColumnName = OrderID
myDataColumn.ColumnName = ProductID

myDataColumn.ColumnName = OrderID
myDataColumn.DataType = System.Int32
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
```

```
myDataColumn.ReadOnly = False
myDataColumn.Unique = False

myDataColumn.ColumnName = ProductID
myDataColumn.DataType = System.Int32
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = False
myDataColumn.Unique = False
myDataColumn.ColumnName = UnitPrice
myDataColumn.DataType = System.Decimal
myDataColumn.AllowDBNull = False
myDataColumn.AutoIncrement = False
myDataColumn.AutoIncrementSeed = 0
myDataColumn.AutoIncrementStep = 1
myDataColumn.MaxLength = -1
myDataColumn.ReadOnly = False
myDataColumn.Unique = False
```

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Finding, Filtering, and Sorting Rows in a *DataTable*

Each row in a *DataTable* is stored in a *DataRow* object, and in this section you'll learn how to find, filter, and sort the *DataRow* objects in a *DataTable*.

### Finding a *DataRow* in a *DataTable*

To find a *DataRow* in a *DataTable*, you follow these steps:

1. Retrieve the rows from the database into your *DataTable*.
2. Set the *PrimaryKey* property of your *DataTable*.
3. Call the *Find()* method of your *DataTable*, passing the primary key column value of the *DataRow* you want.

For example, the following code performs steps 1 and 2 in this list, retrieving the top 10 rows from the *Products* table and setting the *PrimaryKey* property to the *ProductID*  *DataColumn*:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 10 ProductID, ProductName " +
    "FROM Products " +
    "ORDER BY ProductID";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet, "Products");
mySqlConnection.Close();
DataTable productsDataTable = myDataSet.Tables["Products"];
productsDataTable.PrimaryKey =
    new DataColumn[]
    {
        productsDataTable.Columns["ProductID"]
    };
```

Next, the following example performs step 3, calling the *Find()* method to retrieve the *DataRow* from *productsDataTable* that has a *ProductID* of 3:

```
DataRow productDataRow = productsDataTable.Rows.Find("3");
```

Notice that the *Find()* method is called through the *Rows* property of *productsDataTable*. The *Rows* property returns an object of the *DataRowCollection* class.

If the primary key for the database table consists of more than one column, then you can pass an array of objects to the *Find()* method. For example, the *Order Details* table's primary key is made up of the *OrderID* and *ProductID* columns. Assuming you've already performed steps 1 and 2 and retrieved the rows from the *Order Details* table into a *DataTable* object named *orderDetailsDataTable*, then the following example retrieves the *DataRow* with an *OrderID* and *ProductID* of 10248 and 11, respectively:

```
object[] orderDetails =
    new object[]
    {
        10248,
        11
    };
DataRow orderDetailDataRow = orderDetailsDataTable.Rows.Find(orderDetails);
```

## Filtering and Sorting *DataRow* Objects in a *DataTable*

To filter and sort the *DataRow* objects in a *DataTable*, you use the `Select()` method of your *DataTable*. The `Select()` method is overloaded as follows:

```
DataRow[] Select()
DataRow[] Select(string filterExpression)
DataRow[] Select(string filterExpression, string sortExpression)
DataRow[] Select(string filterExpression, string sortExpression,
    DataViewRowState myDataGridViewRowState)
```

where

- ***filterExpression*** specifies the rows to select.
- ***sortExpression*** specifies how the selected rows are to be ordered.
- ***myDataGridViewRowState*** specifies the state of the rows to select. You set *myDataGridViewRowState* to one of the constants defined in the `System.Data.DataViewRowState` enumeration. [Table 11.8](#) shows these constants.

**Table 11.8: *DataViewRowState* ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
Added	A new row.
CurrentRows	The current rows, which include Unchanged, Added, and ModifiedCurrent rows.
Deleted	A deleted row.
ModifiedCurrent	A current row that has been modified.
ModifiedOriginal	The original row before it was modified.
None	Doesn't match any of the rows in the <i>DataTable</i> .
OriginalRows	The original rows, which include Unchanged and Deleted rows.
Unchanged	A row that hasn't been changed.

Let's take a look at some examples that use the `Select()` method.

The following example calls the `Select()` method with no parameters, which returns all rows in the *DataTable* without any filtering or sorting:

```
DataRow[] productDataRows = productsDataTable.Select();
```

The next example supplies a filter expression to `Select()`, which returns only the *DataRow* objects with `ProductID`  *DataColumn* values that are less than or equal to 5:

```
DataRow[] productDataRows = productsDataTable.Select("ProductID <= 5");
```

The following example supplies both a filter expression and a sort expression that orders the *DataRow* objects by descending `ProductID` values:

```
DataRow[] productDataRows = productsDataTable.Select("ProductID <= 5", "ProductID DESC");
```

The next example supplies a `DataViewRowState` of `OriginalRows` to the previous `Select()` call:

```
DataRow[] productDataRows =
    productsDataTable.Select("ProductID <= 5", "ProductID DESC",
        DataViewRowState.OriginalRows);
```

```
    DataViewRowState.OriginalRows);
```

As you can see from the previous examples, the filter and sort expressions are similar to WHERE and ORDER BY clauses in a SELECT statement. You can therefore use very powerful expressions in your calls to the Sort( ) method. For example, you can use AND, OR, NOT, IN, LIKE, comparison operators, arithmetic operators, wildcard characters, and aggregate functions in your filter expressions.

**Note** For full details on how to use such filter expressions, refer to the *DataTable.Expression* property in the .NET online documentation.

The following example that uses the LIKE operator and the percent wildcard character (%)—which matches any number of characters—to filter rows with a ProductName that start with Cha. The example also sorts the rows by descending Product ID and ascending ProductName values:

```
productDataRows =
    productsDataTable.Select("ProductName LIKE 'Cha%'",
    "ProductID DESC, ProductName ASC");
```

Notice that the string Cha% is placed in single quotes, which you must do for all string literals.

**Note** You can also use a *DataGridView* object to filter and sort rows, and you'll learn how to do that in [Chapter 13](#), "Using *DataGridView* Objects."

[Listing 11.3](#) shows a program that finds, filters, and sorts *DataRow* objects.

### **Listing 11.3: FINDFILTERANDSORTDATAROWS.CS**

---

```
/*
 FindFilterAndSortDataRows.cs illustrates how to find, filter,
 and sort DataRow objects
 */

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

class FindFilterAndSortDataRows
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 10 ProductID, ProductName " +
            "FROM Products " +
            "ORDER BY ProductID;" +
            "SELECT TOP 10 OrderID, ProductID, UnitPrice, Quantity " +
            "FROM [Order Details] " +
            "ORDER BY OrderID";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        mySqlDataAdapter.Fill(myDataSet);
        mySqlConnection.Close();
        myDataSet.Tables["Table"].TableName = "Products";
        myDataSet.Tables["Table1"].TableName = "Order Details";
```

```

// set the PrimaryKey property for the Products DataTable
// to the ProductID column
DataTable productsDataTable = myDataSet.Tables["Products"];
productsDataTable.PrimaryKey =
    new DataColumn[]
    {
        productsDataTable.Columns["ProductID"]
    };

// set the PrimaryKey property for the Order Details DataTable
// to the OrderID and ProductID columns
DataTable orderDetailsDataTable = myDataSet.Tables["Order Details"];
orderDetailsDataTable.Constraints.Add(
    "Primary key constraint on the OrderID and ProductID columns",
    new DataColumn[]
    {
        orderDetailsDataTable.Columns["OrderID"],
        orderDetailsDataTable.Columns["ProductID"]
    },
    true
);

// find product with ProductID of 3 using the Find() method
// to locate the DataRow using its primary key value
Console.WriteLine("Using the Find() method to locate DataRow object " +
    "with a ProductID of 3");
DataRow productDataRow = productsDataTable.Rows.Find("3");
foreach (DataColumn my DataColumn in productsDataTable.Columns)
{
    Console.WriteLine(my DataColumn + " = " + productDataRow[my DataColumn]);
}

// find order with OrderID of 10248 and ProductID of 11 using
// the Find() method
Console.WriteLine("Using the Find() method to locate DataRow object " +
    "with an OrderID of 10248 and a ProductID of 11");
object[] orderDetails =
    new object[]
    {
        10248,
        11
    };
DataRow orderDetailDataRow = orderDetailsDataTable.Rows.Find(orderDetails);
foreach (DataColumn my DataColumn in orderDetailsDataTable.Columns)
{
    Console.WriteLine(my DataColumn + " = " + orderDetailDataRow[my DataColumn]);
}

// filter and sort the DataRow objects in productsDataTable
// using the Select() method
Console.WriteLine("Using the Select() method to filter and sort DataRow
objects");
DataRow[] productDataRows =
    productsDataTable.Select("ProductID <= 5", "ProductID DESC",
        DataViewRowState.OriginalRows);
foreach (DataRow my DataRow in productDataRows)
{
    foreach (DataColumn my DataColumn in productsDataTable.Columns)
    {

```

```
        Console.WriteLine(myDataColumn + " = " + myDataRow[myDataColumn]);
    }

// filter and sort the DataRow objects in productsDataTable
// using the Select() method
Console.WriteLine("Using the Select() method to filter and sort DataRow
objects");
productDataRows =
    productsDataTable.Select("ProductName LIKE 'Cha*'",
        "ProductID ASC, ProductName DESC");
foreach (DataRow myDataRow in productDataRows)
{
    foreach (DataColumn myDataColumn in productsDataTable.Columns)
    {
        Console.WriteLine(myDataColumn + " = " + myDataRow[myDataColumn]);
    }
}
}
```

---

The output from this program is as follows:

```
Using the Find() method to locate DataRow object with a ProductID of 3
ProductID = 3
ProductName = Aniseed Syrup
Using the Find() method to locate DataRow object with an OrderID of 10248 and a
ProductID of 11
OrderID = 10248
ProductID = 11
UnitPrice = 14
Quantity = 12
Using the Select() method to filter and sort DataRow objects
ProductID = 5
ProductName = Chef Anton's Gumbo Mix
ProductID = 4
ProductName = Chef Anton's Cajun Seasoning
ProductID = 3
ProductName = Aniseed Syrup
ProductID = 2
ProductName = Chang
ProductID = 1
ProductName = Chai
Using the Select() method to filter and sort DataRow objects
ProductID = 1
ProductName = Chai
ProductID = 2
ProductName = Chang
```

## Modifying Rows in a *DataTable*

In this section, you'll see the steps required to add, modify, and remove *DataRow* objects from a *DataTable* and then push those changes to the database. The examples in this section show how to add, modify, and delete rows in the *Customers* database table.

**Note** You'll find a complete program named *AddModifyAndRemoveDataRows.cs* in the *ch11* directory that illustrates the use of the methods shown in this section. This program listing is omitted from this book for brevity.

### Setting up a *DataAdapter* to Push Changes to the Database

In [Chapter 10](#), you saw that before you call the *Fill()* method of your *DataAdapter* to read rows from the database, you first need to set the *SelectCommand* property of your *DataAdapter*. For example:

```
SqlCommand mySelectCommand = mySqlConnection.CreateCommand();
mySelectCommand.CommandText =
    "SELECT CustomerID, CompanyName, Address " +
    "FROM Customers " +
    "ORDER BY CustomerID";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySelectCommand;
```

The *SELECT* statement is then run when you call the *mySqlDataAdapter* object's *Fill()* method to retrieve rows from the *Customers* table into a *DataSet*.

Similarly, before you can push changes to the database, you must first set up your *DataAdapter* with *Command* objects containing appropriate SQL *INSERT*, *UPDATE*, and *DELETE* statements. You store these *Command* objects in your *DataAdapter* object's *InsertCommand*, *UpdateCommand*, and *DeleteCommand* properties.

You push changes from your *DataSet* to the database using the *Update()* method of your *DataAdapter*. When you add, modify, or remove *DataRow* objects from your *DataSet* and then call the *Update()* method of your *DataAdapter*, the appropriate *InsertCommand*, *UpdateCommand*, or *DeleteCommand* is run to push your changes to the database.

Let's take a look at how to set the *InsertCommand*, *UpdateCommand*, and *DeleteCommand* properties of a *DataAdapter*.

### Setting the *InsertCommand* Property of a *DataAdapter*

The following example creates a *SqlCommand* object named *myInsertCommand* that contains an *INSERT* statement:

```
SqlCommand myInsertCommand = mySqlConnection.CreateCommand();
myInsertCommand.CommandText =
    "INSERT INTO Customers (" +
    "CustomerID, CompanyName, Address" +
    ") VALUES (" +
    "@CustomerID, @CompanyName, @Address" +
    ")";
myInsertCommand.Parameters.Add("@CustomerID", SqlDbType.NChar,
    5, "CustomerID");
myInsertCommand.Parameters.Add("@CompanyName", SqlDbType.NVarChar,
    40, "CompanyName");
myInsertCommand.Parameters.Add("@Address", SqlDbType.NVarChar,
    60, "Address");
```

The four parameters to the *Add()* method are as follows:

- The name of the parameter
- The .NET type of the parameter

- The maximum length of the string that may be assigned to the parameter's value
- The name of the corresponding database column that the parameter is bound to

**Note** Commands and parameters are covered in [Chapter 8](#), "Executing Database Commands."

As you can see from the previous code, the @CustomerID, @CompanyName, and @Address parameters are bound to the CustomerID, CompanyName, and Address columns in the database.

Next, the following example sets the InsertCommand property of mySqlDataAdapter to myInsertCommand:

```
mySqlDataAdapter.InsertCommand = myInsertCommand;
```

## Setting the *UpdateCommand* Property of a *DataAdapter*

The following example creates a SqlCommand object named myUpdateCommand that contains an UPDATE statement and sets the UpdateCommand property of mySqlDataAdapter to myUpdateCommand:

```
myUpdateCommand.CommandText =
    "UPDATE Customers " +
    "SET " +
    "    CompanyName = @NewCompanyName, " +
    "    Address = @NewAddress " +
    "WHERE CustomerID = @OldCustomerID " +
    "AND CompanyName = @OldCompanyName " +
    "AND Address = @OldAddress";
myUpdateCommand.Parameters.Add("@NewCompanyName", SqlDbType.NVarChar,
    40, "CompanyName");
myUpdateCommand.Parameters.Add("@NewAddress", SqlDbType.NVarChar,
    60, "Address");
myUpdateCommand.Parameters.Add("@OldCustomerID", SqlDbType.NChar,
    5, "CustomerID");
myUpdateCommand.Parameters.Add("@OldCompanyName", SqlDbType.NVarChar,
    40, "CompanyName");
myUpdateCommand.Parameters.Add("@OldAddress", SqlDbType.NVarChar,
    60, "Address");
myUpdateCommand.Parameters["@OldCustomerID"].SourceVersion =
    DataRowVersion.Original;
myUpdateCommand.Parameters["@OldCompanyName"].SourceVersion =
    DataRowVersion.Original;
myUpdateCommand.Parameters["@OldAddress"].SourceVersion =
    DataRowVersion.Original;
mySqlDataAdapter.UpdateCommand = myUpdateCommand;
```

There are two things to notice about this code:

- The UPDATE statement's WHERE clause specifies parameters for CompanyID, CompanyName, and Address columns. This uses optimistic concurrency, which you'll learn about shortly.
- A property named SourceVersion for the @OldCustomerID, @OldCompanyName and @OldAddress parameters is set to DataRowVersion.Original. This causes the values for these parameters to be set to the original DataRow column values before you change them.

These items determine the concurrency of the UPDATE, which you'll now learn about.

## Concurrency

Concurrency determines how multiple users' modifications to the same row are handled. There are two types of concurrency that apply to a DataSet:

- **Optimistic Concurrency** With *optimistic concurrency*, you can modify a row in a database table only if no one else has modified that same row since you loaded it into your `DataSet`. This is typically the best type of concurrency to use because you don't want to overwrite someone else's changes.
- **"Last One Wins" Concurrency** With "*last one wins*" concurrency, you can always modify a row-and your changes overwrite anyone else's changes. You typically want to avoid using "last one wins" concurrency.

To use optimistic concurrency, you have to do the following in your `UPDATE` or `DELETE` statement's `WHERE` clause:

1. Include all the columns used in the original `SELECT`.
2. Set these column values to original values retrieved from the row in the table before you changed the values.

When you do these two things in your `UPDATE` or `DELETE` statement's `WHERE` clause, your statement first checks that the original row still exists before updating or deleting the row. That way, you can be sure your changes don't overwrite anyone else's changes. Of course, if the original row has been deleted by another user, then your `UPDATE` or `DELETE` statement will fail.

To use "last one wins" concurrency, you just include the primary key and its value in the `WHERE` clause of your `UPDATE` or `DELETE` statement. Since your `UPDATE` statement doesn't check the original values, it simply overwrites anyone else's changes if the row still exists. Also, a `DELETE` statement simply deletes the row-even if another user has modified the row.

Returning to the previous code example that set the `UpdateCommand` property of `mySqlDataAdapter`, you can see that all the columns are included in the `WHERE` clause of the `UPDATE`. That satisfies the first requirement of using optimistic concurrency shown earlier.

The second requirement is that you set the column in the `WHERE` clause to the original row values. You do this by setting the `SourceVersion` property of the `@OldCustomerID`, `@OldCompanyName`, and `@OldAddress` parameters to `DataRowVersion.Original`. At runtime, this pulls the original values from the `DataColumn` objects in the `DataRow` before you changed them and puts them in the `UPDATE` statement's `WHERE` clause.

`Original` is just one of the members of the `System.Data.DataRowVersion` enumeration; the others are shown in [Table 11.9](#).

**Table 11.9: `DataRowVersion` ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
Current	The current column value.
Default	The default column value.
Original	The original column value.
Proposed	The proposed column value, which is set when you edit a <code>DataRow</code> using the <code>BeginEdit()</code> method.

## Setting the `DeleteCommand` Property of a `DataAdapter`

The following example creates a `SqlCommand` object named `myDeleteCommand` that contains a `DELETE` statement and sets the `DeleteCommand` property of `mySqlDataAdapter` to `myDeleteCommand`:

```
SqlCommand myDeleteCommand = mySqlConnection.CreateCommand();
myDeleteCommand.CommandText =
    "DELETE FROM Customers " +
    "WHERE CustomerID = @OldCustomerID " +
    "AND CompanyName = @OldCompanyName " +
    "AND Address = @OldAddress";
myDeleteCommand.Parameters.Add("@OldCustomerID", SqlDbType.NChar,
    5, "CustomerID");
myDeleteCommand.Parameters.Add("@OldCompanyName", SqlDbType.NVarChar,
    40, "CompanyName");
myDeleteCommand.Parameters.Add("@OldAddress", SqlDbType.NVarChar,
```

## Modifying Rows in a DataTable

```
60, "Address");
myDeleteCommand.Parameters["@OldCustomerID"].SourceVersion =
    DataRowVersion.Original;
myDeleteCommand.Parameters["@OldCompanyName"].SourceVersion =
    DataRowVersion.Original;
myDeleteCommand.Parameters["@OldAddress"].SourceVersion =
    DataRowVersion.Original;
mySqlDataAdapter.DeleteCommand = myDeleteCommand;
```

Notice that the `DELETE` statement also uses optimistic concurrency.

This completes the setup of the `DataAdapter` object.

## Adding a *DataRow* to a *DataTable*

In this section, you'll learn how to add a `DataRow` to a `DataTable`. Before you see this, let's populate a `DataSet` with the rows from the `Customers` table. The following code creates a `DataSet` object named `myDataSet` and populates it by calling `mySqlDataAdapter.Fill()`:

```
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
int numRows =
    mySqlDataAdapter.Fill(myDataSet, "Customers");
mySqlConnection.Close();
```

The `int` returned by the `Fill()` method is the number of rows retrieved from the database. The `myDataSet` object now contains a `DataTable` named `Customers`, which contains the rows retrieved by the following `SELECT` statement set earlier in the `SelectCommand` property of `mySqlDataAdapter`:

```
SELECT CustomerID, CompanyName, Address
FROM Customers
ORDER BY CustomerID
```

To add a new row to a `DataTable` object, you use the following steps:

1. Use the `NewRow()` method of your `DataTable` to create a new `DataRow`.
2. Set the values for the `DataColumn` objects of your new `DataRow`. Note: you can set a `DataColumn` value to null using the `SetNull()` method of a `DataRow`. You can also check if a `DataColumn` contains null using the `IsNull()` method of a `DataRow`.
3. Use the `Add()` method through the `Rows` property of your `DataTable` to add your new `DataRow` to the `DataTable`.
4. Use the `Update()` method of your `DataAdapter` to push the new row to the database.

The following method, named `AddDataRow()`, uses these steps to add a new row to a `DataTable`:

```
public static void AddDataRow(
    DataTable myDataTable,
    SqlDataAdapter mySqlDataAdapter,
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn AddDataRow()");

    // step 1: use the NewRow() method of the DataTable to
    // create a new DataRow
    Console.WriteLine("Calling myDataTable.NewRow()");
    DataRow myNewDataRow = myDataTable.NewRow();
    Console.WriteLine("myNewDataRow.RowState = " +
```

## Modifying Rows in a DataTable

```
myNewDataRow.RowState);

// step 2: set the values for the DataColumn objects of
// the new DataRow
myNewDataRow[ "CustomerID" ] = "J5COM";
myNewDataRow[ "CompanyName" ] = "J5 Company";
myNewDataRow[ "Address" ] = "1 Main Street";

// step 3: use the Add() method through the Rows property
// to add the new DataRow to the DataTable
Console.WriteLine("Calling myDataTable.Rows.Add()");
myDataTable.Rows.Add(myNewDataRow);
Console.WriteLine("myNewDataRow.RowState = " +
    myNewDataRow.RowState);

// step 4: use the Update() method to push the new
// row to the database
Console.WriteLine("Calling mySqlDataAdapter.Update()");
mySqlConnection.Open();
int numRows = mySqlDataAdapter.Update(myDataTable);
mySqlConnection.Close();
Console.WriteLine("numRows = " + numRows);
Console.WriteLine("myNewDataRow.RowState = " +
    myNewDataRow.RowState);

DisplayDataRow(myNewDataRow, myDataTable);
}
```

You'll notice I call the `Open()` and `Close()` methods of `mySqlConnection` around the call to the `Update()` method. You don't have to do this because the `Update()` method-like the `Fill()` method-will automatically open and then close `mySqlConnection` if it is currently closed. It is good programming practice, however, to explicitly include the `Open()` and `Close()` calls so that you can see exactly what is going on.

**Note** In the ADO.NET disconnected model of data access, you should typically keep the connection to the database open for as short a period as possible. Of course, if you're making a lot of calls to the `Update()` or the `Fill()` method over a short time, you could keep the connection open and then close it when you're finished. That way, your code will have better performance. You might need to experiment with your own programs to find the right balance.

The `Update()` method is overloaded as follows:

```
int Update(DataRow[] myDataRows)
int Update(DataSet myDataSet)
int Update(DataTable myDataTable)
int Update(DataRow[] myDataRows, DataTableMapping myDataTableMapping)
int Update(DataSet myDataSet, string dataTableName)
```

where `dataTable` is a string containing the name of the `DataTable` to update. The `int` returned by the `Update()` method is the number of rows successfully updated in the database.

Going back to the previous `AddDataRow()` method, you'll also notice the inclusion of `Console.WriteLine()` calls that display the `RowState` property of `myNewDataRow`. The `RowState` property is set to one of the constants defined in the `System.Data.DataViewRowState` enumeration. [Table 11.10](#) shows the constants defined in the `DataRowState` enumeration.

**Table 11.10: DataRowState ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
Added	The <code>DataRow</code> has been added to the <code>DataRowCollection</code> of the <code>DataTable</code> .

Deleted	The DataRow has been removed from the DataTable.
Detached	The DataRow isn't part of the DataTable.
Modified	The DataRow has been modified.
Unchanged	The DataRow hasn't been modified.

AddDataRow( ) calls a method named `DisplayDataRow()`, which displays the `DataColumn` values for the `DataRow` passed as the first parameter. `DisplayDataRow()` is defined as follows:

```
public static void DisplayDataRow(
    DataRow myDataRow,
    DataTable myDataTable)
{
    Console.WriteLine("\nIn DisplayDataRow()");
    foreach (DataColumn my DataColumn in myDataTable.Columns)
    {
        Console.WriteLine(my DataColumn + " = " +
            myDataRow[my DataColumn]);
    }
}
```

In the previous `AddDataRow()` method, you saw that it displays the `RowState` property of `myNewDataRow` at various points. The output from `AddDataRow()` and its call to `DisplayDataRow()` is as follows:

```
In AddDataRow()
Calling myDataTable.NewRow()
myNewDataRow.RowState = Detached
Calling myDataTable.Rows.Add()
myNewDataRow.RowState = Added
Calling mySqlDataAdapter.Update()
numOfRows = 1
myNewDataRow.RowState = Unchanged
In DisplayDataRow()
CustomerID = J5COM
CompanyName = J5 Company
Address = 1 Main Street
```

Let's examine this run in detail:

- After `myDataTable.NewRow()` is called to create `myNewDataRow`, its `RowState` is `Detached`, which indicates `myNewDataRow` isn't yet part of `myDataTable`.
- Next, `myDataTable.Rows.Add()` is called to add `myNewDataRow` to `myDataTable`. This causes the `RowState` of `myNewDataRow` to change to `Added`, which indicates `myNewDataRow` is now part of `myDataTable`.
- Finally, `mySqlDataAdapter.Update()` is called to push the new row to the database. This causes the `RowState` of `myNewDataRow` to change to `Unchanged`.

Behind the scenes, the `Update()` method runs the `INSERT` statement in the `mySqlDataAdapter.InsertCommand` property to add the new row to the `Customers` table. The `int` returned by the `Update()` statement is the number of rows affected by the method call. In this example, one is returned since one row was added.

## Modifying a `DataRow` in a `DataTable`

To modify a `DataRow` in a `DataTable`, you use the following steps:

1. Set the `PrimaryKey` property of your `DataTable`. You need to set this to find the `DataRow` in the next step.

## Modifying Rows in a DataTable

2. Use the `Find()` method to locate the `DataRow` that you want to modify in your `DataTable`. You locate the `DataRow` using the value of its primary key column.
3. Change the `DataColumn` values for your `DataRow`.
4. Use the `Update()` method of your `DataAdapter` object to push the modified row to the database.

The following method, named `ModifyDataRow()`, uses these steps to modify the row that was previously added by the `AddDataRow()` method:

```
public static void ModifyDataRow(
    DataTable myDataTable,
    SqlDataAdapter mySqlDataAdapter,
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn ModifyDataRow()");

    // step 1: set the PrimaryKey property of the DataTable
    myDataTable.PrimaryKey =
        new DataColumn[]
    {
        myDataTable.Columns["CustomerID"]
    };

    // step 2: use the Find() method to locate the DataRow
    // in the DataTable using the primary key value
    DataRow myEditDataRow = myDataTable.Rows.Find("J5COM");

    // step 3: change the DataColumn values of the DataRow
    myEditDataRow["CompanyName"] = "Widgets Inc.";
    myEditDataRow["Address"] = "1 Any Street";
    Console.WriteLine("myEditDataRow.RowState = " +
        myEditDataRow.RowState);
    Console.WriteLine("myEditDataRow[\"CustomerID\", " +
        "DataRowVersion.Original] = " +
        myEditDataRow["CustomerID", DataRowVersion.Original]);
    Console.WriteLine("myEditDataRow[\"CompanyName\", " +
        "DataRowVersion.Original] = " +
        myEditDataRow["CompanyName", DataRowVersion.Original]);
    Console.WriteLine("myEditDataRow[\"Address\", " +
        "DataRowVersion.Original] = " +
        myEditDataRow["Address", DataRowVersion.Original]);
    Console.WriteLine("myEditDataRow[\"CompanyName\", " +
        "DataRowVersion.Current] = " +
        myEditDataRow["CompanyName", DataRowVersion.Current]);
    Console.WriteLine("myEditDataRow[\"Address\", " +
        "DataRowVersion.Current] = " +
        myEditDataRow["Address", DataRowVersion.Current]);

    // step 4: use the Update() method to push the modified
    // row to the database
    Console.WriteLine("Calling mySqlDataAdapter.Update()");
    mySqlConnection.Open();
    int numRows = mySqlDataAdapter.Update(myDataTable);
    mySqlConnection.Close();
    Console.WriteLine("numRows = " + numRows);
    Console.WriteLine("myEditDataRow.RowState = " +
        myEditDataRow.RowState);
```

## Modifying Rows in a DataTable

```
    DisplayDataRow(myEditDataRow, myDataTable);  
}
```

Setting the primary key in step 1 doesn't have to be done inside the `ModifyDataRow()` method. You could, for example, set the primary key immediately after calling the `Fill()` method in the `Main()` method of the `AddModifyAndRemoveDataRows.cs` program. The reason I set the primary key in `ModifyDataRow()` is that you can see all the steps together in this method.

Notice in step 3 of this method the original values for the `CustomerID`, `CompanyName`, and `Address`  `DataColumn` objects are displayed using the `DataRowVersion.Original` constant. These are the  `DataColumn` values before they are changed. The current values for the `CompanyName` and `Address`  `DataColumn` objects are also displayed using the `DataRowVersion.Current` constant. These are the  `DataColumn` values after they are changed.

The output from `ModifyDataRow()` and its call to `DisplayDataRow()` is as follows:

```
In ModifyDataRow()  
myEditDataRow.RowState = Modified  
myEditDataRow[ "CustomerID", DataRowVersion.Original] = J5COM  
myEditDataRow[ "CompanyName", DataRowVersion.Original] = J5 Company  
myEditDataRow[ "Address", DataRowVersion.Original] = 1 Main Street  
myEditDataRow[ "CompanyName", DataRowVersion.Current] = Widgets Inc.  
myEditDataRow[ "Address", DataRowVersion.Current] = 1 Any Street  
Calling mySqlDataAdapter.Update()  
numOfRows = 1  
myEditDataRow.RowState = Unchanged  
  
In DisplayDataRow()  
CustomerID = J5COM  
CompanyName = Widgets Inc.  
Address = 1 Any Street
```

Notice that the `CompanyName` and `Address`  `DataColumn` objects of `myEditDataRow` are changed. The `RowState` property of `myEditDataRow` changes to `Modified` after its `CompanyName` and `Address` are changed, and then to `Unchanged` after `mySqlDataAdapter.Update()` is called.

## Marking Your Modifications

You can use the `BeginEdit()` method to mark the beginning of a modification to a `DataRow`. For example:

```
myEditDataRow.BeginEdit();  
myEditDataRow[ "CompanyName" ] = "Widgets Inc.";  
myEditDataRow[ "Address" ] = "1 Any Street";
```

You then use either the `EndEdit()` or `CancelEdit()` methods to mark the end of the modification to the `DataRow`. `EndEdit()` commits the modification; `CancelEdit()` rejects the modification and restores the `DataRow` to its original state before the edit began.

The following example calls the `EndEdit()` method of `myEditDataRow` to commit the changes made in the previous example:

```
myEditDataRow.EndEdit();
```

## Removing a *DataRow* from a *DataTable*

To remove a `DataRow` from a `DataTable`, you use the following steps:

1. Set the `PrimaryKey` property for your `DataTable` object.
2. Use the `Find()` method to locate your `DataRow`.
3. Use the `Delete()` method to remove your `DataRow`.

4. Use the `Update()` method to push the delete to the database.

The following method, named `RemoveDataRow()`, uses these steps to remove the `DataRow` that was previously modified by the `ModifyDataRow()` method:

```
public static void RemoveDataRow(
    DataTable myDataTable,
    SqlDataAdapter mySqlDataAdapter,
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn RemoveDataRow()");

    // step 1: set the PrimaryKey property of the DataTable
    myDataTable.PrimaryKey =
        new DataColumn[]
    {
        myDataTable.Columns["CustomerID"]
    };

    // step 2: use the Find() method to locate the DataRow
    DataRow myRemoveDataRow = myDataTable.Rows.Find("J5COM");

    // step 3: use the Delete() method to remove the DataRow
    Console.WriteLine("Calling myRemoveDataRow.Delete()");
    myRemoveDataRow.Delete();
    Console.WriteLine("myRemoveDataRow.RowState = " +
        myRemoveDataRow.RowState);

    // step 4: use the Update() method to remove the deleted
    // row from the database
    Console.WriteLine("Calling mySqlDataAdapter.Update()");
    mySqlConnection.Open();
    int numRows = mySqlDataAdapter.Update(myDataTable);
    mySqlConnection.Close();
    Console.WriteLine("numRows = " + numRows);
    Console.WriteLine("myRemoveDataRow.RowState = " +
        myRemoveDataRow.RowState);
}
```

The output from `RemoveDataRow()` is as follows:

```
In RemoveDataRow()
Calling myRemoveDataRow.Delete()
myRemoveDataRow.RowState = Deleted
Calling mySqlDataAdapter.Update()
numRows = 1
myRemoveDataRow.RowState = Detached
```

Notice that the `RowState` property of `myRemoveDataRow` is set to `Deleted` after `myRemoveData .Delete()` is called, and then to `Detached` after `mySqlDataAdapter.Update()` is called-meaning that `myRemoveDataRow` is no longer part of the `DataTable`.

**Note** You'll find a complete program named `AddModifyAndRemoveDataRows.cs` in the `ch11` directory that illustrates the use of the `AddDataRow()`, `ModifyDataRow()`, and `RemoveDataRow()` methods. This program listing is omitted from this book for brevity.

## Retrieving New Identity Column Values

The `ProductID` column of the `Products` table is an identity column. In this section, you'll see how to insert a new row into to `Products` table and retrieve the new value generated by the database for the `ProductID` identity column.

**Note** You'll find a complete program named `UsingIdentityColumn.cs` in the `ch11` directory that illustrates the use of the methods shown in this section. This program listing is omitted from this book for brevity.

In the examples, assume you have a `DataTable` named `productsDataTable` that is populated with the rows retrieved by the following `SELECT` statement:

```
SELECT
    ProductID, ProductName, UnitPrice
FROM Products
ORDER BY ProductID
```

The following example sets the `PrimaryKey` property of `productsDataTable`:

```
productsDataTable.PrimaryKey =
    new DataColumn[]
    {
        productsDataTable.Columns["ProductID"]
    };
```

The next example sets the `AllowDBNull`, `AutoIncrement`, `AutoIncrementSeed`, `AutoIncrementStep`, `ReadOnly`, and `Unique` properties for the `ProductID` `DataColumn` of `productsDataTable`:

```
DataTable productIDDataColumn =
    productsDataTable.Columns["ProductID"];
productIDDataColumn.AllowDBNull = false;
productIDDataColumn.AutoIncrement = true;
productIDDataColumn.AutoIncrementSeed = -1;
productIDDataColumn.AutoIncrementStep = -1;
productIDDataColumn.ReadOnly = true;
productIDDataColumn.Unique = true;
```

Because of these settings, when you add a new `DataRow` to `productsDataTable` the `ProductID` `DataColumn` of your new `DataRow` will initially have the value -1.

As in the earlier section, "[Modifying Rows in a DataTable](#)," you need to set your `DataAdapter` object's `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties with appropriate `Command` objects. The `CommandText` property of the `Command` object used in the `UpdateCommand` property is as follows:

```
myUpdateCommand.CommandText =
    "UPDATE Products " +
    "SET " +
    "  ProductName = @NewProductName, " +
    "  UnitPrice = @NewUnitPrice " +
    "WHERE ProductID = @OldProductID " +
    "AND ProductName = @OldProductName " +
    "AND UnitPrice = @OldUnitPrice";
```

The `CommandText` property of the `Command` object used in the `DeleteCommand` property is as follows:

```
myDeleteCommand.CommandText =
    "DELETE FROM Products " +
    "WHERE ProductID = @OldProductID " +
    "AND ProductName = @OldProductName " +
    "AND UnitPrice = @OldUnitPrice";
```

Notice the `CommandText` of these two `Command` objects isn't substantially different from those shown in the [previous section](#), except that it goes against the `Products` table rather than the `Customers` table.

The real difference is in the `CommandText` of the `Command` object used in the `InsertCommand` property, which must retrieve the `ProductID` value generated by the database for the new row. To do this, you can use the following code that contains an `INSERT` statement that adds a new row, along with a `SELECT` statement that retrieves the `ProductID` value using a call to the SQL Server `SCOPE_IDENTITY()` function:

```
myInsertCommand.CommandText =
    "INSERT INTO Products ( " +
    " ProductName, UnitPrice " +
    " ) VALUES ( " +
    " @MyProductName, @MyUnitPrice " +
    " ); " +
    "SELECT @MyProductID = SCOPE_IDENTITY();";
myInsertCommand.Parameters.Add(
    "@MyProductName", SqlDbType.NVarChar, 40, "ProductName");
myInsertCommand.Parameters.Add(
    "@MyUnitPrice", SqlDbType.Money, 0, "UnitPrice");
myInsertCommand.Parameters.Add("@MyProductID", SqlDbType.Int,
    0, "ProductID");
myInsertCommand.Parameters["@MyProductID"].Direction =
    ParameterDirection.Output;
```

The `SCOPE_IDENTITY()` function returns the last inserted identity value into any table performed within the current database session *and* stored procedure, trigger, function, or batch. For example, calling `SCOPE_IDENTITY()` in the previous example returns the last identity value inserted into the `Products` table, which is the `ProductID` of the new row.

**Note** For details on the `SCOPE_IDENTITY()` function, refer to [Chapter 4](#), "Introduction to Transact-SQL Programming."

When you add a new `DataRow` to `productsDataTable`, the `ProductID`  `DataColumn` of your new `DataRow` will initially have the value -1. When you call the `Update()` method of your `SqlDataAdapter` to push the new row to the database, the following steps occur:

1. Your new `DataRow` is pushed to the database using the `INSERT` statement set in `myInsertCommand`, with the `ProductID` column of the `Products` table being set to a new identity value generated by the database.
2. The `ProductID` identity value is retrieved by the `SELECT` statement set in `myInsertCommand`.
3. The `ProductID`  `DataColumn` in your `DataRow` is set to the retrieved identity value.

Feel free to examine, compile, and run the `UsingIdentityColumn.cs` program located in the `ch11` directory. This program performs the following high-level actions:

1. Retrieves rows from the `Products` table into a `DataTable` named `productsDataTable`.
2. Adds a `DataRow` to `productsDataTable`.
3. Modifies the new `DataRow`.
4. Deletes the new `DataRow`.

When you run this program you'll notice the change in the `ProductID`  `DataColumn` value for a newly added `DataRow` from -1 to an actual value retrieved from the database.

## Using Stored Procedures to Add, Modify, and Remove Rows from the Database

You can get a `DataAdapter` object to call stored procedures to add, modify, and remove rows from the database. These procedures are called instead of the `INSERT`, `UPDATE`, and `DELETE` statements you've seen how to set in a `DataAdapter` object's `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties.

The ability to call stored procedures using a `DataAdapter` is a very powerful addition to ADO.NET. For example, you can use a stored procedure to add a row to a table containing an identity column, and then retrieve the new value for that column generated by the database. You can also do additional work in a stored procedure such as inserting a row into an audit table when a row is modified. You'll see examples of both these scenarios in this section.

**Tip** Using stored procedures instead of `INSERT`, `UPDATE`, and `DELETE` statements can also improve performance. You should use stored procedures if your database supports them. SQL Server and Oracle support stored procedures. Oracle stored-procedures are written in PL/SQL.

The `ProductID` column of the `Products` table is an identity column, and you saw a number of stored procedures in [Chapter 4](#), "Introduction to Transact-SQL Programming," that added a row to the `Products` table and returned the `ProductID`.

In this section, you'll see how to

- Create the required stored procedures in the Northwind database.
- Set up a `DataAdapter` to call the stored procedures.
- Add, modify, and remove a `DataRow` to from a `DataTable`.

The C# methods shown in this section follow the same steps as shown in the earlier section, "[Modifying Rows in a DataTable](#)."

**Note** You'll find a complete program named `PushChangesUsingProcedures.cs` in the `ch11` directory that illustrates the use of the methods shown in this section. The listing for this program is omitted from this book for brevity.

## Creating the Stored Procedures in the Database

You'll create the following three stored procedures in the Northwind database:

- `AddProduct4()`, which adds a row to the `Products` table.
- `UpdateProduct()`, which updates a row in the `Products` table.
- `DeleteProduct()`, which deletes a row from the `Products` table.

Let's take a look at these procedures.

### The `AddProduct4()` Procedure

`AddProduct4()` adds a row to the `Products` table. It uses the number 4 because previous chapters featured procedures named `AddProduct()`, `AddProduct2()`, and `AddProduct3()`.

[Listing 11.4](#) shows the `AddProduct4.sql` file that you use to create the `AddProduct4()` procedure. Refer to [Chapter 4](#) if you need a refresher on the Transact-SQL language or if you need to find out how to run this script to create the procedure in the database.

### Listing 11.4: ADDPRODUCT4.SQL

---

```
/*
AddProduct4.sql creates a procedure that adds a row to the
Products table using values passed as parameters to the
procedure. The procedure returns the ProductID of the new row
```

```

        using a RETURN statement
*/
CREATE PROCEDURE AddProduct4
    @MyProductName nvarchar(40),
    @MyUnitPrice money
AS
    -- declare the @MyProductID variable
    DECLARE @MyProductID int

    -- insert a row into the Products table
    INSERT INTO Products (
        ProductName, UnitPrice
    ) VALUES (
        @MyProductName, @MyUnitPrice
    )

    -- use the SCOPE_IDENTITY() function to get the last
    -- identity value inserted into a table performed within
    -- the current database session and stored procedure,
    -- so SCOPE_IDENTITY returns the ProductID for the new row
    -- in the Products table in this case
    SET @MyProductID = SCOPE_IDENTITY()

    RETURN @MyProductID

```

---

**Note** You'll find the *AddProduct4.sql* file in the *ch11* directory.

## The *UpdateProduct()* Procedure

*UpdateProduct()* updates a row in the *Products* table. [Listing 11.5](#) shows the *UpdateProduct.sql* file that you use to create the *UpdateProduct()* procedure.

### **Listing 11.5: UPDATEPRODUCT.SQL**

---

```

/*
UpdateProduct.sql creates a procedure that modifies a row
in the Products table using values passed as parameters
to the procedure
*/
CREATE PROCEDURE UpdateProduct
    @OldProductID int,
    @NewProductName nvarchar(40),
    @NewUnitPrice money,
    @OldProductName nvarchar(40),
    @OldUnitPrice money
AS
    -- update the row in the Products table
    UPDATE Products
    SET
        ProductName = @NewProductName,
        UnitPrice = @NewUnitPrice
    WHERE ProductID = @OldProductID
    AND ProductName = @OldProductName
    AND UnitPrice = @OldUnitPrice

```

---

Because the WHERE clause contains the old column values in the UPDATE statement of this procedure, the UPDATE uses optimistic concurrency described earlier. This means that one user doesn't overwrite another user's changes.

## The *DeleteProduct()* Procedure

`DeleteProduct()` deletes a row from the `Products` table. [Listing 11.6](#) shows the `DeleteProduct.sql` file that you use to create the `DeleteProduct()` procedure.

### **Listing 11.6: DELETEPRODUCT.SQL**

---

```
/*
DeleteProduct.sql creates a procedure that removes a row
from the Products table
*/
CREATE PROCEDURE DeleteProduct
    @OldProductID int,
    @OldProductName nvarchar(40),
    @OldUnitPrice money
AS
    -- delete the row from the Products table
    DELETE FROM Products
    WHERE ProductID = @OldProductID
    AND ProductName = @OldProductName
    AND UnitPrice = @OldUnitPrice
```

---

## Using *SET NOCOUNT ON* in Stored Procedures

In [Chapter 4](#), "Introduction to Transact-SQL Programming," you saw that you use the `SET NOCOUNT ON` command to prevent Transact-SQL from returning the number of rows affected. Typically, you must avoid using this command in your stored procedures because the `DataAdapter` uses the returned number of rows affected to know whether the update succeeded.

There is one situation when you must use `SET NOCOUNT ON`: when your stored procedure performs an `INSERT`, `UPDATE`, or `DELETE` statement that affects another table besides the main one you are pushing a change to. For example, say the `DeleteProduct()` procedure also performed an `INSERT` statement to add a row to the `ProductAudit` table (described in [Chapter 4](#)) to record the attempt to delete the row from the `Products` table. In this example, you must use `SET NOCOUNT ON` before performing the `INSERT` into the `ProductAudit` table, as shown in [Listing 11.7](#).

### **Listing 11.7: DELETEPRODUCT2.SQL**

---

```
/*
DeleteProduct2.sql creates a procedure that removes a row
from the Products table
*/
CREATE PROCEDURE DeleteProduct2
    @OldProductID int,
    @OldProductName nvarchar(40),
    @OldUnitPrice money
AS
    -- delete the row from the Products table
    DELETE FROM Products
    WHERE ProductID = @OldProductID
    AND ProductName = @OldProductName
    AND UnitPrice = @OldUnitPrice
```

```
-- use SET NOCOUNT ON to suppress the return of the
-- number of rows affected by the INSERT statement
SET NOCOUNT ON

-- add a row to the Audit table
IF @@ROWCOUNT = 1
    INSERT INTO ProductAudit (
        Action
    ) VALUES (
        'Product deleted with ProductID of ' +
        CONVERT(nvarchar, @OldProductID)
    )
ELSE
    INSERT INTO ProductAudit (
        Action
    ) VALUES (
        'Product with ProductID of ' +
        CONVERT(nvarchar, @OldProductID) +
        ' was not deleted'
    )

```

---

By using `SET NOCOUNT ON` before the `INSERT`, only the number of rows affected by the `DELETE` statement is returned, and the `DataAdapter` therefore gets the correct value.

Transact-SQL also has a `SET NOCOUNT ON` command to turn on the number of rows affected. You can use a combination of `SET NOCOUNT OFF` and `SET NOCOUNT ON` if you need to perform an `INSERT`, `UPDATE`, or `DELETE` statement before the main SQL statement in your stored procedure.

## Setting Up a *DataAdapter* to Call Stored Procedures

As mentioned in the earlier section "[Modifying Rows in a \*DataTable\*](#)," you need to create a `DataAdapter` object and set its `SelectCommand`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties with appropriate `Command` objects. This time, however, the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties will contain `Command` objects that call the stored procedures shown earlier.

First, the following example creates a `SqlCommand` object containing a `SELECT` statement and sets the `SelectCommand` property of a `SqlDataAdapter` to that `SqlCommand`:

```
SqlCommand mySelectCommand = mySqlConnection.CreateCommand();
mySelectCommand.CommandText =
    "SELECT " +
    " ProductID, ProductName, UnitPrice " +
    "FROM Products " +
    "ORDER BY ProductID";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySelectCommand;
```

The `SELECT` statement is then run when you call the `mySqlDataAdapter` object's `Fill()` method to retrieve rows from the `Products` table into a `DataSet`.

Before you can push changes to the database, you must set the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties of your `DataAdapter` with `Command` objects. These `Command` objects will contain calls to the `AddProduct4()`, `UpdateProduct()`, and `DeleteProduct()` stored procedures that you created earlier. When you then add, modify, or remove `DataRow` objects from your `DataSet`, and then call the `Update()` method of your `DataAdapter`, the appropriate stored procedure is run to push your changes to the database.

Let's take a look at how to set the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties of your `DataAdapter`.

## Setting the `InsertCommand` Property of a `DataAdapter`

The following example creates a `SqlCommand` object named `myInsertCommand` that contains a call to the `AddProduct4()` stored procedure:

```
SqlCommand myInsertCommand = mySqlConnection.CreateCommand();
myInsertCommand.CommandText =
    "EXECUTE @MyProductID = AddProduct4 @MyProductName, @MyUnitPrice";
myInsertCommand.Parameters.Add(
    "@MyProductID", SqlDbType.Int, 0, "ProductID");
myInsertCommand.Parameters["@MyProductID"].Direction =
    ParameterDirection.Output;
myInsertCommand.Parameters.Add(
    "@MyProductName", SqlDbType.NVarChar, 40, "ProductName");
myInsertCommand.Parameters.Add(
    "@MyUnitPrice", SqlDbType.Money, 0, "UnitPrice");
```

As you can see from the previous code, the direction of the `@MyProductID` parameter is set to `ParameterDirection.Output`, which indicates that this parameter is an output parameter. Also, the maximum length of the `@MyProductID` and `@MyUnitPrice` parameters is set to 0 in the third parameter to the `Add()` method. Setting them to 0 is fine because the maximum length doesn't apply to fixed length types such as numbers, only to types such as strings.

Next, the following example sets the `InsertCommand` property of `mySqlDataAdapter` to `myInsertCommand`:

```
mySqlDataAdapter.InsertCommand = myInsertCommand;
```

## Setting the `UpdateCommand` Property of a `DataAdapter`

The following example creates a `SqlCommand` object named `myUpdateCommand` that contains a call to the `UpdateProduct()` stored procedure and sets the `UpdateCommand` property of `mySqlDataAdapter` to `myUpdateCommand`:

```
SqlCommand myUpdateCommand = mySqlConnection.CreateCommand();
myUpdateCommand.CommandText =
    "EXECUTE UpdateProduct @OldProductID, @NewProductName, " +
    "@NewUnitPrice, @OldProductName, @OldUnitPrice";
myUpdateCommand.Parameters.Add(
    "@OldProductID", SqlDbType.Int, 0, "ProductID");
myUpdateCommand.Parameters.Add(
    "@NewProductName", SqlDbType.NVarChar, 40, "ProductName");
myUpdateCommand.Parameters.Add(
    "@NewUnitPrice", SqlDbType.Money, 0, "UnitPrice");
myUpdateCommand.Parameters.Add(
    "@OldProductName", SqlDbType.NVarChar, 40, "ProductName");
myUpdateCommand.Parameters.Add(
    "@OldUnitPrice", SqlDbType.Money, 0, "UnitPrice");
myUpdateCommand.Parameters["@OldProductID"].SourceVersion =
    DataRowVersion.Original;
myUpdateCommand.Parameters["@OldProductName"].SourceVersion =
    DataRowVersion.Original;
myUpdateCommand.Parameters["@OldUnitPrice"].SourceVersion =
    DataRowVersion.Original;
mySqlDataAdapter.UpdateCommand = myUpdateCommand;
```

## Setting the `DeleteCommand` Property of a `DataAdapter`

The following example creates a `SqlCommand` object named `myDeleteCommand` that contains a call to the `DeleteProduct()` stored procedure and sets the `DeleteCommand` property of `mySqlDataAdapter` to `myDeleteCommand`:

```

SqlCommand myDeleteCommand = mySqlConnection.CreateCommand();
myDeleteCommand.CommandText =
    "EXECUTE DeleteProduct @OldProductID, @OldProductName, @OldUnitPrice";
myDeleteCommand.Parameters.Add(
    "@OldProductID", SqlDbType.Int, 0, "ProductID");
myDeleteCommand.Parameters.Add(
    "@OldProductName", SqlDbType.NVarChar, 40, "ProductName");
myDeleteCommand.Parameters.Add(
    "@OldUnitPrice", SqlDbType.Money, 0, "UnitPrice");
myDeleteCommand.Parameters["@OldProductID"].SourceVersion =
    DataRowVersion.Original;
myDeleteCommand.Parameters["@OldProductName"].SourceVersion =
    DataRowVersion.Original;
myDeleteCommand.Parameters["@OldUnitPrice"].SourceVersion =
    DataRowVersion.Original;
mySqlDataAdapter.DeleteCommand = myDeleteCommand;

```

This completes the setup of the `DataAdapter` object.

## Adding a *DataRow* to a *DataTable*

In this section, you'll learn how to add a `DataRow` to a `DataTable`. First, the following code creates a `DataSet` object named `myDataSet` and populates it by calling `mySqlDataAdapter.Fill()`:

```

DataSet myDataSet = new DataSet();
mySqlConnection.Open();
int numOfRows =
    mySqlDataAdapter.Fill(myDataSet, "Products");
mySqlConnection.Close();

```

The `int` returned by the `Fill()` method is the number of rows retrieved from the database and copied to `myDataSet`. The `myDataSet` object now contains a `DataTable` named `Products`, which contains the rows retrieved by the following `SELECT` statement set earlier in the `SelectCommand` property of `mySqlDataAdapter`:

```

SELECT ProductID, ProductName, UnitPrice
FROM Products
ORDER BY ProductID

```

To add a new row to a `DataTable` object, you use the same four steps as shown earlier in the section "[Modifying a \*DataRow\* in a \*DataTable\*](#)." The following method, named `AddDataRow()`, uses those steps to add a new row to a `DataTable`:

```

public static int AddDataRow(
    DataTable myDataTable,
    SqlDataAdapter mySqlDataAdapter,
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn AddDataRow()");

    // step 1: use the NewRow() method of the DataTable to
    // create a new DataRow
    Console.WriteLine("Calling myDataTable.NewRow()");
    DataRow myNewDataRow = myDataTable.NewRow();
    Console.WriteLine("myNewDataRow.RowState = " +
        myNewDataRow.RowState);

    // step 2: set the values for the DataColumn objects of
    // the new DataRow
    myNewDataRow["ProductName"] = "Widget";
    myNewDataRow["UnitPrice"] = 10.99;
}

```

```

// step 3: use the Add() method through the Rows property
// to add the new DataRow to the DataTable
Console.WriteLine("Calling myDataTable.Rows.Add( )");
myDataTable.Rows.Add(myNewDataRow);
Console.WriteLine("myNewDataRow.RowState = " +
    myNewDataRow.RowState);
// step 4: use the Update() method to push the new
// row to the database
Console.WriteLine("Calling mySqlDataAdapter.Update( )");
mySqlConnection.Open();
int numRows = mySqlDataAdapter.Update(myDataTable);
mySqlConnection.Close();
Console.WriteLine("numRows = " + numRows);
Console.WriteLine("myNewDataRow.RowState = " +
    myNewDataRow.RowState);

DisplayDataRow(myNewDataRow, myDataTable);

// return the ProductID of the new DataRow
return (int) myNewDataRow["ProductID"];
}

```

Notice that no value for the `ProductID`  `DataColumn` is set in step 2. This is because the `ProductID` is automatically generated by the database when the new row is pushed to the database by the `Update( )` method in step 4.

When the `Update( )` method is called, the `AddProduct4( )` stored procedure is run to add the new row to the `Products` table. The database then generates a new `ProductID` for the row, which is then returned by the `AddProduct4( )` stored procedure. You can then read the new `ProductID` using `myNewDataRow[ "ProductID" ]`, which now contains the new `ProductID`. This `ProductID` is then returned at the end of the `AddDataRow( )` method.

The output from `AddDataRow( )` and its call to `DisplayDataRow( )` are as follows:

```

In AddDataRow()
Calling myDataTable.NewRow()
myNewDataRow.RowState = Detached
Calling myDataTable.Rows.Add()
myNewDataRow.RowState = Added
Calling mySqlDataAdapter.Update()
numRows = 1
myNewDataRow.RowState = Unchanged

```

```

In DisplayDataRow()
ProductID = 180
ProductName = Widget
UnitPrice = 10.99

```

As you can see, after `myDataTable.NewRow( )` is called to create `myNewDataRow` its `RowState` is `Detached`, which indicates `myNewDataRow` isn't yet part of `myDataTable`.

Next, `myDataTable.Rows.Add( )` is called to add `myNewDataRow` to `myDataTable`. This causes the `RowState` of `myNewDataRow` to change to `Added`, which indicates `myNewDataRow` has been added to `myDataTable`.

Finally, `mySqlDataAdapter.Update( )` is called to push the new row to the database. The `AddProduct4( )` stored procedure is run to add the new row to the `Products` table, and the `RowState` of `myNewDataRow` changes to `Unchanged`.

## Modifying a *DataRow* in a *DataTable*

The following method, named `ModifyDataRow()`, uses four steps to modify a `DataRow` in a `DataTable` object. Notice that the `ProductID` to modify is passed as a parameter:

```

public static void ModifyDataRow(
    DataTable myDataTable,
    int productID,
    SqlDataAdapter mySqlDataAdapter,
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn ModifyDataRow()");

    // step 1: set the PrimaryKey property of the DataTable
    myDataTable.PrimaryKey =
        new DataColumn[]
    {
        myDataTable.Columns["ProductID"]
    };

    // step 2: use the Find() method to locate the DataRow
    // in the DataTable using the primary key value
    DataRow myEditDataRow = myDataTable.Rows.Find(productID);

    // step 3: change the DataColumn values of the DataRow
    myEditDataRow["ProductName"] = "Advanced Widget";
    myEditDataRow["UnitPrice"] = 24.99;
    Console.WriteLine("myEditDataRow.RowState = " +
        myEditDataRow.RowState);
    Console.WriteLine("myEditDataRow[\" ProductID\", " +
        "DataRowVersion.Original] = " +
        myEditDataRow["ProductID", DataRowVersion.Original]);
    Console.WriteLine("myEditDataRow[\" ProductName\", " +
        "DataRowVersion.Original] = " +
        myEditDataRow["ProductName", DataRowVersion.Original]);
    Console.WriteLine("myEditDataRow[\" UnitPrice\", " +
        "DataRowVersion.Original] = " +
        myEditDataRow["UnitPrice", DataRowVersion.Original]);
    Console.WriteLine("myEditDataRow[\" ProductName\", " +
        "DataRowVersion.Current] = " +
        myEditDataRow["ProductName", DataRowVersion.Current]);
    Console.WriteLine("myEditDataRow[\" UnitPrice\", " +
        "DataRowVersion.Current] = " +
        myEditDataRow["UnitPrice", DataRowVersion.Current]);

    // step 4: use the Update() method to push the update
    // to the database
    Console.WriteLine("Calling mySqlDataAdapter.Update()");
    mySqlConnection.Open();
    int numRows = mySqlDataAdapter.Update(myDataTable);
    mySqlConnection.Close();
    Console.WriteLine("numOfRows = " + numRows);
    Console.WriteLine("myEditDataRow.RowState = " +
        myEditDataRow.RowState);

    DisplayDataRow(myEditDataRow, myDataTable);
}

```

Notice this method displays the original values for the `ProductID`, `ProductName`, and `UnitPrice` `DataColumn` objects using the `DataRowVersion.Original` constant. These are the `DataColumn` values before they are changed. The method also displays the current values for the `ProductName` and `UnitPrice` `DataColumn` objects using the `DataRowVersion.`

Current constant. These are the `DataTable` values after they are changed. When the `Update()` method is called in step 4, the `UpdateProduct()` stored procedure is run behind the scenes to perform the update.

The output from `ModifyDataRow()` and its call to `DisplayDataRow()` is as follows:

```
In ModifyDataRow()
myEditDataRow.RowState = Modified
myEditDataRow[ "ProductID", DataRowVersion.Original] = 180
myEditDataRow[ "ProductName", DataRowVersion.Original] = Widget
myEditDataRow[ "UnitPrice", DataRowVersion.Original] = 10.99
myEditDataRow[ "ProductName", DataRowVersion.Current] = Advanced Widget
myEditDataRow[ "UnitPrice", DataRowVersion.Current] = 24.99
Calling mySqlDataAdapter.Update()
numOfRows = 1
myEditDataRow.RowState = Unchanged

In DisplayDataRow()
ProductID = 180
ProductName = Advanced Widget
UnitPrice = 24.99
```

Notice that the `RowState` property of `myEditDataRow` changes to `Modified` after it is changed, and then to `Unchanged` after `mySqlDataAdapter.Update()` is called.

## Removing a *DataRow* from a *DataTable*

The following method, named `RemoveDataRow()`, uses four steps to remove a `DataRow` from a `DataTable`. Notice that the `ProductID` to modify is passed as a parameter:

```
public static void RemoveDataRow(
    DataTable myDataTable,
    int productID,
    SqlDataAdapter mySqlDataAdapter,
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn RemoveDataRow()");

    // step 1: set the PrimaryKey property of the DataTable
    myDataTable.PrimaryKey =
        new DataColumn[]
    {
        myDataTable.Columns[ "ProductID" ]
    };

    // step 2: use the Find() method to locate the DataRow
    DataRow myRemoveDataRow = myDataTable.Rows.Find(productID);

    // step 3: use the Delete() method to remove the DataRow
    Console.WriteLine("Calling myRemoveDataRow.Delete()");
    myRemoveDataRow.Delete();
    Console.WriteLine("myRemoveDataRow.RowState = " +
        myRemoveDataRow.RowState);

    // step 4: use the Update() method to push the delete
    // to the database
    Console.WriteLine("Calling mySqlDataAdapter.Update()");
    mySqlConnection.Open();
    int numOfRows = mySqlDataAdapter.Update(myDataTable);
    mySqlConnection.Close();
```

```
Console.WriteLine("numOfRows = " + numOfRows);
Console.WriteLine("myRemoveDataRow.RowState = " +
    myRemoveDataRow.RowState);
}
```

The output from `RemoveDataRow()` is as follows:

```
In RemoveDataRow()
Calling myRemoveDataRow.Delete()
myRemoveDataRow.RowState = Deleted
Calling mySqlDataAdapter.Update()
numOfRows = 1
myRemoveDataRow.RowState = Detached
```

Notice that the `RowState` property of `myRemoveDataRow` is set to `Deleted` after `myRemoveData .Delete()` is called, and then to `Detached` after `mySqlDataAdapter.Update()` is called. When the `Update()` method is called in step 4, the `DeleteProduct()` stored procedure is run behind the scenes to perform the delete.

**Note** You'll find a complete program named `PushChangesUsingProcedures.cs` in the `ch11` directory that illustrates the use of the `AddDataRow()`, `ModifyDataRow()`, and `RemoveDataRow()` methods. This listing is omitted from this book for brevity.

**Team LiB**

[◀ PREVIOUS](#)

[NEXT ▶](#)

## Automatically Generating SQL Statements

As you've seen in the previous sections, supplying your own `INSERT`, `UPDATE`, and `DELETE` statements or stored procedures to push changes from your `DataSet` to the database means you have to write a lot of code. You can avoid writing this code by using a `CommandBuilder` object, which can automatically generate single-table `INSERT`, `UPDATE`, and `DELETE` commands that push the changes you make to a `DataSet` object to the database. These commands are then set in the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties of your `DataAdapter` object. When you then make changes to your `DataSet` and call the `Update()` method of your `DataAdapter`, the automatically generated command is run to push the changes to the database.

Although you can save writing some code using a `CommandBuilder`, you must remember the following limitations when using a `CommandBuilder`:

- The `SelectCommand` property of your `DataAdapter` can retrieve rows from only a single table.
- The database table used in your `SelectCommand` must contain a primary key.
- The primary key of the table must be included in your `SelectCommand`.
- The `CommandBuilder` takes a certain amount of time to generate the commands because it has to examine the database.

**Warning** Because a `CommandBuilder` lowers the performance of your program, you should try to avoid using them. They are intended for use by developers who aren't familiar with SQL or stored procedures. For best performance, use stored procedures.

There are three `CommandBuilder` managed provider classes: `SqlCommandBuilder`, `OleDbCommandBuilder`, and `OdbcCommandBuilder`. You'll see the use of a `SqlCommandBuilder` object in this section that works with a SQL Server database. The other types of objects work in the same way.

First, you need to set the `SelectCommand` property of a `SqlDataAdapter` object. The `SELECT` statement used in this command can retrieve rows from only a single table, and in the following example the `Customers` table is used:

```
SqlCommand mySelectCommand = mySqlConnection.CreateCommand();
mySelectCommand.CommandText =
    "SELECT CustomerID, CompanyName, Address " +
    "FROM Customers " +
    "ORDER BY CustomerID";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySelectCommand;
```

Next, the following example creates a `SqlCommandBuilder` object, passing `mySqlDataAdapter` to the constructor:

```
SqlCommandBuilder mySqlCommandBuilder =
    new SqlCommandBuilder(mySqlDataAdapter);
```

The `SqlCommandBuilder` will generate the commands containing `INSERT`, `UPDATE`, and `DELETE` statements based on the `SELECT` statement previously set in the `mySqlDataAdapter` object's `SelectCommand` property.

You can obtain the generated commands using the `GetInsertCommand()`, `GetUpdateCommand()`, and `GetDeleteCommand()` methods of `mySqlCommandBuilder`. For example:

```
Console.WriteLine(
    "mySqlCommandBuilder.GetInsertCommand().CommandText =\n" +
    mySqlCommandBuilder.GetInsertCommand().CommandText
);
Console.WriteLine(
    "mySqlCommandBuilder.GetUpdateCommand().CommandText =\n" +
    mySqlCommandBuilder.GetUpdateCommand().CommandText
);
```

```
) ;
Console.WriteLine(
    "mySqlCommandBuilder.GetDeleteCommand().CommandText =\\n" +
    mySqlCommandBuilder.GetDeleteCommand().CommandText
);
```

This code displays the following output (I've added some white space to make it easier to read):

```
mySqlCommandBuilder.GetInsertCommand().CommandText =
    INSERT INTO Customers(CustomerID , CompanyName , Address)
    VALUES (@p1 , @p2 , @p3)
mySqlCommandBuilder.GetUpdateCommand().CommandText =
    UPDATE Customers
    SET CustomerID = @p1 , CompanyName = @p2 , Address = @p3
    WHERE (CustomerID = @p4 AND CompanyName = @p5 AND Address = @p6)
mySqlCommandBuilder.GetDeleteCommand().CommandText =
    DELETE FROM Customers
    WHERE (CustomerID = @p1 AND CompanyName = @p2 AND Address = @p3)
```

As you can see, these commands are similar to those shown earlier in the section "[Modifying Rows in a DataTable](#)." The SQL statements used in these commands use optimistic concurrency.

You can now populate and make changes to a `DataTable` containing rows from the `Customers` table, and then push those changes to the database using the `Update()` method. You can use the same `AddDataRow()`, `ModifyDataRow()`, and `RemoveDataRow()` methods as shown in the earlier section, "[Modifying Rows in a DataTable](#)."

**Note** You'll find a complete program named `UsingCommandBuilder.cs` in the `ch11` directory that illustrates the use of the `CommandBuilder` object shown in this section. This listing is omitted from this book for brevity.

## Exploring the *DataAdapter* and *DataTable* Events

You'll find all the code examples shown in this section in the program `UsingEvents.cs` located in the `ch11` directory. The listing for this program is omitted from this book for brevity. If you compile and run that program, you'll see the order in which the events fire when you add, modify, and remove a row from a `DataTable` that contains rows retrieved from the `Customers` table.

### The *DataAdapter* Events

The events exposed by a `SqlDataAdapter` object are shown in [Table 11.11](#).

**Table 11.11: `SqlDataAdapter` EVENTS**

EVENT	EVENT HANDLER	DESCRIPTION
FillError	FillErrorEventHandler	Fires when an error occurs during a call to the <code>Fill()</code> method.
RowUpdating	RowUpdatingEventHandler	Fires <i>before</i> a row is added, modified, or deleted in the database as a result of calling the <code>Update()</code> method.
RowUpdated	RowUpdatedEventHandler	Fires <i>after</i> a row is added, modified, or deleted in the database as a result of calling the <code>Update()</code> method.

### The *FillError* Event

The `FillError` event fires when you call the `Fill()` method and an error occurred. The following are a couple of scenarios that would cause an error:

- An attempt is made to add a number from the database to a  `DataColumn` that couldn't be converted to the .NET type of that  `DataColumn` without losing precision.
- An attempt is made to add a row from the database to a  `DataTable` that violates a constraint in that  `DataTable`.

The following example event handler, named `FillErrorEventHandler()`, checks for the precision conversion error:

```
public static void FillErrorEventHandler(
    object sender, FillEventArgs myFEEA
)
{
    if (myFEEA.Errors.GetType() == typeof(System.OverflowException))
    {
        Console.WriteLine("A loss of precision occurred");
        myFEEA.Continue = true;
    }
}
```

The first parameter is an object of the `System.Object` class, and it represents the object that raises the event. The second parameter is an object of the `FillEventArgs` class, which like all the `EventArgs` classes, is derived from the `System.EventArgs` class. The `EventArgs` class is the base class for event data and represents the details of the event. [Table 11.12](#) shows the `FillEventArgs` properties.

**Table 11.12: `FillEventArgs` PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
Continue	bool	Gets or sets a <code>bool</code> that indicates whether you want to continue filling your <code>DataSet</code> even though an error has occurred. The default is <code>false</code> .

DataTable	DataTable	Gets the DataTable that was being filled when the error occurred.
Errors	Exception	Gets the Exception containing the error that occurred.
Values	object[]	Gets the DataColumn values of the DataRow in which the error occurred. These values are returned in an object array.

You indicate that `mySqlDataAdapter` is to call the `FillErrorEventHandler()` method when the `FillError` event fires using the following code:

```
mySqlDataAdapter.FillError +=  
    new FillErrorEventHandler(FillErrorEventHandler);
```

## The RowUpdating Event

The `RowUpdating` event fires before a row is updated in the database as a result of you calling the `Update()` method of your `DataAdapter`. This event fires once for each `DataRow` you've added, modified, or deleted in a `DataTable`.

The following actions are performed behind the scenes for each `DataRow` when you call the `Update()` method of your `DataAdapter`:

1. The values in your `DataRow` are copied to the parameter values of the appropriate Command in the `InsertCommand`, `UpdateCommand`, or `DeleteCommand` property of your `DataAdapter`.
2. The `RowUpdating` event of your `DataAdapter` fires.
3. The Command is run to push the change to the database.
4. Any output parameters from the Command are returned.
5. The `RowUpdated` event of your `DataAdapter` fires.
6. The `AcceptChanges()` method of your `DataRow` is called.

The second parameter to any event handler method you write to handle the `RowUpdating` event of a `SqlDataAdapter` object is of the `SqlRowUpdatingEventArgs` class, and [Table 11.13](#) shows the properties of this class.

**Table 11.13: SqlRowUpdatingEventArgs PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
Command	SqlCommand	Gets or sets the SqlCommand that is run when the <code>Update()</code> method is called.
Errors	Exception	Gets the <code>Exception</code> for any error that occurred.
Row	DataRow	Gets the <code>DataRow</code> to send to the database through the <code>Update()</code> method.

StatementType	StatementType	<p>Gets the type of the SQL statement that is to be run. <code>StatementType</code> is an enumeration in the <code>System.Data</code> namespace that contains the following members:</p> <ul style="list-style-type: none"> <li>• <code>Delete</code></li> <li>• <code>Insert</code></li> <li>• <code>Select</code></li> <li>• <code>Update</code></li> </ul>
Status	UpdateStatus	<p>Gets or sets the <code>UpdateStatus</code> of the <code>Command</code> object. <code>UpdateStatus</code> is an enumeration in the <code>System.Data</code> namespace that contains the following members:</p> <ul style="list-style-type: none"> <li>• <code>Continue</code>, which indicates that the <code>DataAdapter</code> is to continue processing rows.</li> <li>• <code>ErrorsOccurred</code>, which indicates that the update is to be treated as an error.</li> <li>• <code>SkipAllRemainingRows</code>, which indicates that the current row and <i>all</i> remaining rows are to be skipped and not updated.</li> <li>• <code>SkipCurrentRow</code>, which indicates that the current row is to be skipped and not updated. The default is <code>Continue</code>.</li> </ul>
TableMapping	DataTableMapping	<p>Gets the <code>DataTableMapping</code> object that is sent to the <code>Update()</code> method. A <code>DataTableMapping</code> object contains a description of a mapped relationship between a source table and a <code>DataTable</code> (see <a href="#">Chapter 10</a>, "Using <code>DataSet</code> Objects to Store Data").</p>

The following example event handler, named `RowUpdatingEventHandler()`, prevents any new rows from being added to the database with a `CustomerID` of J5COM:

```
public static void RowUpdatingEventHandler(
    object sender, SqlRowUpdatingEventArgs mySRUEA
)
{
    Console.WriteLine("\nIn RowUpdatingEventHandler() ");
    if ((mySRUEA.StatementType == StatementType.Insert) &&
        (mySRUEA.Row["CustomerID"] == "J5COM"))
    {
        Console.WriteLine("Skipping current row");
        mySRUEA.Status = UpdateStatus.SkipCurrentRow;
    }
}
```

You indicate that `mySqlDataAdapter` is to call the `RowUpdatingEventHandler()` method when the `RowUpdating` event fires using the following code:

```
mySqlDataAdapter.RowUpdating +=
    new SqlRowUpdatingEventHandler(RowUpdatingEventHandler);
```

If you then call the `AddDataRow()` method shown earlier to attempt to add a row to the `Customers` table, the `RowUpdating` event will fire and call the `RowUpdatingEventHandler()` method. This method causes the new row to be skipped and prevents it

from being added to the `Customers` database table.

## The `RowUpdated` Event

The `RowUpdated` event fires after a row is updated in the database as a result of you calling the `Update()` method of your `DataAdapter`. This event fires once for each `DataRow` you've added, modified, or deleted in a `DataTable`.

The second parameter to any method you write to handle the `RowUpdated` event of a `SqlDataAdapter` object is of the `SqlRowEventArgs` class. The properties of this class are the same as those shown earlier in [Table 11.13](#), plus one additional property shown in [Table 11.14](#).

**Table 11.14: ADDITIONAL `SqlRowEventArgs` PROPERTY**

PROPERTY	TYPE	DESCRIPTION
<code>RecordsAffected</code>	<code>int</code>	Gets an <code>int</code> containing the number of rows added, modified, or removed when the appropriate <code>Command</code> is run by the <code>Update()</code> method.

The following example event handler, named `RowUpdatedEventHandler()`, displays the number of records affected by the following Command:

```
public static void RowUpdatedEventHandler(
    object sender, SqlRowEventArgs mySRUEA
)
{
    Console.WriteLine("\nIn RowUpdatedEventHandler() ");
    Console.WriteLine("mySRUEA.RecordsAffected = " +
        mySRUEA.RecordsAffected);
}
```

You indicate that `mySqlDataAdapter` is to call the `RowUpdatedEventHandler()` method when the `RowUpdated` event fires using the following code:

```
mySqlDataAdapter.RowUpdated += 
    new SqlRowUpdatedEventHandler(RowUpdatedEventHandler);
```

## The `DataTable` Events

The events exposed by a `DataTable` object are shown in [Table 11.15](#).

**Table 11.15: `DataTable` EVENTS**

EVENT	EVENT HANDLER	DESCRIPTION
<code>ColumnChanging</code>	<code> DataColumnChangeEventHandler</code>	Fires <i>before</i> a changed <code>DataColumn</code> value is committed in a <code>DataRow</code> .
<code>ColumnChanged</code>	<code> DataColumnChangeEventHandler</code>	Fires <i>after</i> a changed <code>DataColumn</code> value is committed in a <code>DataRow</code> .
<code>RowChanging</code>	<code> DataRowChangeEventHandler</code>	Fires <i>before</i> a changed <code>DataRow</code> is committed in a <code>DataTable</code> .
<code>RowChanged</code>	<code> DataRowChangeEventHandler</code>	Fires <i>after</i> a changed <code>DataRow</code> is committed in a <code>DataTable</code> .
<code>RowDeleting</code>	<code> DataRowChangeEventHandler</code>	Fires <i>before</i> a <code>DataRow</code> is deleted from a <code>DataTable</code> .

RowDeleted	DataRowChangeEventHandler	Fires <b>after</b> a DataRow is deleted from a DataTable.
------------	---------------------------	---

## The **ColumnChanging** and **ColumnChanged** Events

The **ColumnChanging** event fires before a change to a  **DataColumn** value is committed in a  **DataRow**. Similarly, the **ColumnChanged** event fires after a change to a  **DataColumn** value is committed in a  **DataRow**. These two events are always fired before the **RowChanging** and **RowChanged** events.

What is meant by "commit" in this context? If you simply set a new value for a  **DataColumn**, then the change is automatically committed in the **DataRow**. However, if you start the change to the **DataRow** using the **BeginEdit()** method, then the change is committed only when you call the **EndEdit()** method of that **DataRow**. You can also reject the change to the **DataRow** using the **CancelEdit()** method.

The second parameter to any event handler you write to handle the **ColumnChanging** or **ColumnChanged** events of a **DataTable** object is of the  **DataColumnChangeEventArgs** class. [Table 11.16](#) shows the properties of this class.

**Table 11.16: DataColumnChangeEventArgs PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
Column	<b>DataColumn</b>	Gets the <b> DataColumn</b> with the value that is changing.
ProposedValue	<b>object</b>	Gets or sets the new value for the <b> DataColumn</b> .
Row	<b>DataRow</b>	Gets the <b> DataRow</b> that contains the <b> DataColumn</b> with the value that is changing.

The following example method handlers, named **ColumnChangingEventHandler()** and **ColumnChangedEventHandler()**, display the **Column** and **ProposedValue** properties:

```
public static void ColumnChangingEventHandler(
    object sender,
    DataColumnChangeEventArgs myDCCEA
)
{
    Console.WriteLine("\nIn ColumnChangingEventHandler()");
    Console.WriteLine("myDCCEA.Column = " + myDCCEA.Column);
    Console.WriteLine("myDCCEA.ProposedValue = " + myDCCEA.ProposedValue);
}

public static void ColumnChangedEventHandler(
    object sender,
    DataColumnChangeEventArgs myDCCEA
)
{
    Console.WriteLine("\nIn ColumnChangedEventHandler()");
    Console.WriteLine("myDCCEA.Column = " + myDCCEA.Column);
    Console.WriteLine("myDCCEA.ProposedValue = " + myDCCEA.ProposedValue);
}
```

The next example creates a **DataTable** named **customersDataTable** and adds the previous two methods to the **ColumnChanging** and **ColumnChanged** events of **customersDataTable**:

```
DataTable customersDataTable = myDataSet.Tables["Customers"];

customersDataTable.ColumnChanging +=
    new DataColumnChangeEventHandler(ColumnChangingEventHandler);
```

```
customersDataTable.ColumnChanged +=  
    new DataColumnChangeEventHandler(ColumnChangedEventHandler);
```

## The **RowChanging** and **RowChanged** Events

The RowChanging event fires before a change to a DataRow is committed in a DataTable. Similarly, the RowChanged event fires after a change to a DataRow value is committed in a DataTable.

The second parameter to any event handler you write to handle the RowChanging or RowChanged events of a DataTable object is of the DataRowChangeEventArgs class. [Table 11.17](#) shows the properties of this class.

**Table 11.17: DataRowChangeEventArgs PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
Action	DataRowAction	<p>Gets the DataRowAction that has occurred for the DataRow. The <code>DataRowAction</code> enumeration is defined in the <code>System.Data</code> namespace and contains the following members:</p> <ul style="list-style-type: none"> <li>• <code>Add</code>, which indicates the <code>DataRow</code> has been added to the <code>DataTable</code>.</li> <li>• <code>Change</code>, which indicates the <code>DataRow</code> has been modified.</li> <li>• <code>Commit</code>, which indicates the <code>DataRow</code> has been committed in the <code>DataTable</code>.</li> <li>• <code>Delete</code>, which indicates the <code>DataRow</code> has been removed from the <code>DataTable</code>.</li> <li>• <code>Nothing</code>, which indicates the <code>DataRow</code> has not changed.</li> <li>• <code>Rollback</code>, which indicates the change to the <code>DataRow</code> has been rolled back.</li> </ul>
Row	DataRow	Gets the <code>DataRow</code> that contains the <code> DataColumn</code> with the value that is changing.

The following example event handlers, named RowChangingEventHandler() and RowChangedEventHandler(), display the Action property:

```
public static void RowChangingEventHandler(  
    object sender,  
    DataRowChangeEventArgs myDRCEA  
)  
{  
    Console.WriteLine("\nIn RowChangingEventHandler()");  
    Console.WriteLine("myDRCEA.Action = " + myDRCEA.Action);  
}  
  
public static void RowChangedEventHandler(  
    object sender,  
    DataRowChangeEventArgs myDRCEA  
)  
{  
    Console.WriteLine("\nIn RowChangedEventHandler()");  
    Console.WriteLine("myDRCEA.Action = " + myDRCEA.Action);  
}
```

The next example adds the previous two methods to the RowChanging and RowChanged events of customersDataTable:

```
customersDataTable.RowChanging +=  
    new DataRowChangeEventHandler(RowChangingEventHandler);  
  
customersDataTable.RowChanged +=  
    new DataRowChangeEventHandler(RowChangedEventHandler);
```

## The **RowDeleting** and **RowDeleted** Events

The RowDeleting event fires before a DataRow is deleted from a DataTable. Similarly, the RowDeleted event fires after a DataRow is deleted from a DataTable.

The second parameter to any event handler you write to handle the RowDeleting or RowDeleted events of a DataTable is of the `DataRowChangeEventArgs` class, and [Table 11.17](#) shown earlier shows the properties of this class.

The following example method handlers, named RowDeletingEventHandler() and RowDeleted-EventHandler(), display the Action property:

```
public static void RowDeletingEventHandler(  
    object sender,  
    DataRowChangeEventArgs myDRCEA  
)  
{  
    Console.WriteLine("\nIn RowDeletingEventHandler()");  
    Console.WriteLine("myDRCEA.Action = " + myDRCEA.Action);  
}  
  
public static void RowDeletedEventHandler(  
    object sender,  
    DataRowChangeEventArgs myDRCEA  
)  
{  
    Console.WriteLine("\nIn RowDeletedEventHandler()");  
    Console.WriteLine("myDRCEA.Action = " + myDRCEA.Action);  
}
```

The next example adds the previous two methods to the RowDeleting and RowDeleted events of customersDataTable:

```
customersDataTable.RowDeleting +=  
    new DataRowChangeEventHandler(RowDeletingEventHandler);  
  
customersDataTable.RowDeleted +=  
    new DataRowChangeEventHandler(RowDeletedEventHandler);
```

**Note** You'll find all the code examples shown in this section in the program `UsingEvents.cs` located in the `ch11` directory. This program listing is omitted from this book for brevity.

## Dealing with Update Failures

So far, the examples you've seen have assumed that the updates pushed to the database by the `Update()` have succeeded. In this section, you see what happens when updates fail-and what you can do about it.

**Note** You'll find all the code examples shown in this section in the `HandlingUpdateFailures.cs` file located in the `ch11` directory. This program listing is omitted from this book for brevity.

In the examples in this section, assume that the `CommandText` property of a `SqlDataAdapter` object's `UpdateCommand` is set as follows:

```
UPDATE Customers
SET
    CompanyName = @NewCompanyName,
    Address = @NewAddress
WHERE CustomerID = @OldCustomerID
AND CompanyName = @OldCompanyName
AND Address = @OldAddress
```

This UPDATE statement uses optimistic concurrency because the updated columns are included in the WHERE clause.

## An Update Failure Scenario

Consider the following scenario that shows an update failure:

1. User 1 retrieves the rows from the `Customers` table into a `DataTable` named `customersDataTable`.
2. User 2 retrieves the same rows.
3. User 1 updates the `CompanyName`  `DataColumn` of the `DataRow` with the `CustomerID`  `DataColumn` of J5COM and pushes the change to the database. Let's say User 1 changes the `CompanyName` from J5 Company to Updated Company.
4. User 2 updates the same `DataRow` and changes the `CompanyName` from J5 Company to Widgets Inc. and attempts to push the change to the database. User 2 then causes a `DBConcurrencyException` object to be thrown and their update fails. (The same exception occurs if User 2 tries to update or delete a row that has already been deleted by User 1.)

Why does the update fail in step 4? The reason is that with optimistic concurrency, the `CompanyName` column is used in the `WHERE` clause of the `UPDATE` statement. Because of this, the original row loaded by User 2 cannot be found anymore-and therefore the `UPDATE` statement fails. The row cannot be found because User 1 has already changed the `CompanyName` column from J5 Company to Updated Company in step 2.

That's the problem with optimistic concurrency, but what can you as a developer do about it? You can report the problem to User 2, refresh their rows using the `Fill()` method, and they can make their change again-however, if User 2 has already made a large number of changes and they can't save any of them, they'll probably be very annoyed at your program.

Fortunately, you can set the `ContinueUpdateOnError` property of your `DataAdapter` to `true` to continue updating any `DataRow` objects even if an error occurs. That way, when User 2 saves their changes they can at least save the rows that don't cause any errors. Let's take a look at how to set the `ContinueUpdateOnError` property.

## Setting the `ContinueUpdateOnError` Property

The following example sets the `ContinueUpdateOnError` property to `true` for `mySqlDataAdapter`:

```
mySqlDataAdapter.ContinueUpdateOnError = true;
```

When you call `mySqlDataAdapter.Update()`, it will push all the changes that don't cause errors to the database. You can then

check for errors afterward using the `HasErrors` property of a `DataSet` or the `HasErrors` property of individual `DataTable` objects, which you'll see how to shortly in the section "[Checking for Errors](#)."

## Programming a Failed Update Example

Let's program an example of a failed update. This example will simulate the updates made by User 1 and User 2 described earlier. I'll use the following method, named `ModifyRowsUsingUPDATE()`, to simulate the update made by User 1 in step 3 described earlier:

```
public static void ModifyRowUsingUPDATE(
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn ModifyDataRowUsingUPDATE()");
    Console.WriteLine("Updating CompanyName to 'Updated Company' for J5COM");

    SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
    mySqlCommand.CommandText =
        "UPDATE Customers " +
        "SET CompanyName = 'Updated Company' " +
        "WHERE CustomerID = 'J5COM';

    mySqlConnection.Open();
    int numberOfRows = mySqlCommand.ExecuteNonQuery();
    Console.WriteLine("Number of rows updated = " +
        numberOfRows);
    mySqlConnection.Close();
}
```

Notice that the `CompanyName` is set to `Updated Company` for the row with the `CustomerID` of `J5COM`.

I'll use the following `ModifyDataRow()` method to simulate the update made by User 2 in step 4. This is similar to the other `ModifyDataRow()` methods you've seen earlier in this chapter. Notice that the `CompanyName` is set to `Widgets Inc.` for the row with the `CustomerID` of `J5COM`:

```
public static void ModifyDataRow(
    DataTable myDataTable,
    SqlDataAdapter mySqlDataAdapter,
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn ModifyDataRow()");

    // step 1: set the PrimaryKey property of the DataTable
    myDataTable.PrimaryKey =
        new DataColumn[]
    {
        myDataTable.Columns["CustomerID"]
    };

    // step 2: use the Find() method to locate the DataRow
    // in the DataTable using the primary key value
    DataRow myDataRow = myDataTable.Rows.Find("J5COM");

    // step 3: change the DataColumn values of the DataRow
    myDataRow["CompanyName"] = "Widgets Inc.";
    Console.WriteLine("myDataRow.RowState = " +
        myDataRow.RowState);

    // step 4: use the Update() method to push the modified
    // row to the database
```

```

Console.WriteLine("Calling mySqlDataAdapter.Update()");
mySqlConnection.Open();
int numRows = mySqlDataAdapter.Update(myDataTable);
mySqlConnection.Close();
Console.WriteLine("numOfRows = " + numRows);
Console.WriteLine("myDataRow.RowState = " +
    myDataRow.RowState);

DisplayDataRow(myDataRow, myDataTable);
}

```

The next example calls `ModifyRowUsingUPDATE()` to perform the first update of the row with the `CustomerID` of J5COM:

```
ModifyRowUsingUPDATE(mySqlConnection);
```

Next, the `ContinueUpdateOnError` property of `mySqlDataAdapter` is set to `true` to continue updating any `DataRow` objects even if an error occurs:

```
mySqlDataAdapter.ContinueUpdateOnError = true;
```

Next, `ModifyDataRow()` is called to attempt to modify the same row as `ModifyRowsUsingUPDATE()`:

```
ModifyDataRow(customersDataTable, mySqlDataAdapter, mySqlConnection);
```

Normally, this will throw an exception since the row cannot be found, but since the `ContinueUpdateOnError` property of `mySqlDataAdapter` has been set to `true`, no exception will be thrown. That way, if `myDataTable` had other updated rows, they would still be pushed to the database by the call to the `Update()` method.

## Checking for Errors

Having set the `ContinueUpdateOnError` property of `mySqlDataAdapter` to `true`, no exception will be thrown when an error occurs. Instead, you can check for errors in a `DataSet` or individual `DataTable` or `DataRow` using the `HasErrors` property. You can then show the user the details of the error using the `RowError` property of the `DataRow`, along with the original and current values for the `DataColumn` objects in that `DataRow`. For example:

```

if (myDataSet.HasErrors)
{
    Console.WriteLine("\nDataSet has errors!");
    foreach (DataTable myDataTable in myDataSet.Tables)
    {
        // check the HasErrors property of myDataTable
        if (myDataTable.HasErrors)
        {
            foreach (DataRow myDataRow in myDataTable.Rows)
            {
                // check the HasErrors property of myDataRow
                if (myDataRow.HasErrors)
                {
                    Console.WriteLine("Here is the row error:");
                    Console.WriteLine(myDataRow.RowError);
                    Console.WriteLine("Here are the column details in the DataSet:");
                    foreach (DataColumn my DataColumn in myDataTable.Columns)
                    {
                        Console.WriteLine(my DataColumn + "original value = " +
                            myDataRow[my DataColumn, DataRowVersion.Original]);
                        Console.WriteLine(my DataColumn + "current value = " +
                            myDataRow[my DataColumn, DataRowVersion.Current]);
                    }
                }
            }
        }
    }
}

```

```
        }  
    }  
}
```

## Fixing the Error

Showing the user the error is only half the story. What can you do to fix the problem? One solution is to call the `Fill()` method again to synchronize the rows in your `DataSet` with the database. That way, the row updated by `ModifyRowsUsingUPDATE()` in the `Customers` table will be pulled from the `Customers` table and replace the original J5COM row in the `DataSet`. For example:

```
mySqlConnection.Open();  
numOfRows =  
    mySqlDataAdapter.Fill(myDataSet, "Customers");  
mySqlConnection.Close();
```

You might expect `numOfRows` to be 1 because you're replacing only one row, right? Wrong: `numOfRows` will contain the *total* number of rows in the `Customers` table. The reason for this is that the `Fill()` method actually pulls all of the rows from the `Customers` table and puts them in the `Customers` `DataTable` of `myDataSet`, throwing away any existing rows with matching primary key column values already in the `Customers` `DataTable`.

**Warning** If you didn't add a primary key to your `Customers` `DataTable`, then the call to the `Fill()` method would simply add all the rows from the `Customers` table to the `Customers` `DataTable` again-duplicating the rows already there.

Next, you call `ModifyDataRow()` again to modify the J5COM row:

```
ModifyDataRow(customersDataTable, mySqlDataAdapter, mySqlConnection);
```

This time the update succeeds because `ModifyDataRow()` finds the J5COM row in `customersDataTable` that was just pulled from the `Customers` table.

**Note** If a user tries to update a row that has already been deleted from the database table, the only thing you can do is refresh the rows and ask the user to enter their row again.

## Using Transactions with a *DataSet* (SQL)

In [Chapter 3](#), "Introduction to Structured Query Language," you saw how you can group SQL statements together into transactions. The transaction is then committed or rolled back as one unit. For example, in the case of a banking transaction, you might want to withdraw money from one account and deposit it into another account. You would then commit both of these changes as one unit, or if there's a problem, rollback both changes. In [Chapter 8](#), "Executing Database Commands," you saw how to use a *Transaction* object to represent a transaction.

As you know, a *DataSet* doesn't have a direct connection to the database. Instead, you use the `Fill()` and `Update()` methods of a *DataAdapter* to pull and push rows from and to the database to your *DataSet* respectively. In fact, a *DataSet* has no "knowledge" of the database at all. A *DataSet* simply stores a disconnected copy of the data. Because of this, a *DataSet* doesn't have any built-in functionality to handle transactions.

How then do you use transactions with a *DataSet*? The answer is you must use the `Transaction` property of the *Command* objects stored in a *DataAdapter*.

### Using the *DataAdapter Command Object's Transaction Property*

A *DataAdapter* stores four *Command* objects that you access using the `SelectCommand`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties. When you call the `Update()` method of a *DataAdapter*, it runs the appropriate `InsertCommand`, `UpdateCommand`, or `DeleteCommand`.

You can create a *Transaction* object and set the `Transaction` property of the *Command* objects in your *DataAdapter* to this *Transaction* object. When you then modify your *DataSet* and push the changes to the database using the `Update()` method of your *DataAdapter*, the changes will use the same *Transaction*.

The following example creates a *SqlTransaction* object named `mySqlTransaction` and sets the `Transaction` property of each of the *Command* objects in `mySqlDataAdapter` to `mySqlTransaction`:

```
SqlTransaction mySqlTransaction =
    mySqlConnection.BeginTransaction();
mySqlDataAdapter.SelectCommand.Transaction = mySqlTransaction;
mySqlDataAdapter.InsertCommand.Transaction = mySqlTransaction;
mySqlDataAdapter.UpdateCommand.Transaction = mySqlTransaction;
mySqlDataAdapter.DeleteCommand.Transaction = mySqlTransaction;
```

Each of the *Command* objects in `mySqlDataAdapter` will now use `mySqlTransaction`.

Let's say you added, modified, and removed some rows from a *DataTable* contained in a *DataSet* named `myDataSet`. You can push these changes to the database using the following example:

```
mySqlDataAdapter.Update(myDataSet);
```

All your changes to `myDataSet` are pushed to the database as part of the transaction in `mySqlTransaction`. You can commit those changes using the `Commit()` method of `mySqlTransaction`:

```
mySqlTransaction.Commit();
```

You could also roll back those changes using the `Rollback()` method of `mySqlTransaction`.

**Note** A transaction is rolled back by default; therefore, you should always explicitly commit or roll back your transaction using `Commit()` or `Rollback()` to make it clear what your program is intended to do.

## Modifying Data Using a Strongly Typed *DataSet*

In [Chapter 10](#), you saw how to create and use a strongly typed *DataSet* class named *MyDataSet*. You can use objects of this class to represent the *Customers* table and rows from that table. In this section, you'll see how to modify data using a strongly typed object of the *MyDataSet* class.

**Note** One of the features of a strongly typed *DataSet* object allows you to read a column value using a property with the same name as the column. For example, to read the *CustomerID* of a column you can use *myDataRow.CustomerID* rather than *myDataRow["CustomerID"]*. See [Chapter 10](#) for more details on reading column values.

The following methods in the *MyDataSet* class allow you to modify the rows stored in a *MyDataSet* object: *NewCustomersRow()*, *AddCustomersRow()*, and *RemoveCustomersRow()*. You can find a row using the *FindByCustomerID()* method. You can check if a column value contains a null value using methods such as *IsContactNameNull()*, and you can set a column to null using methods such as *SetContactNameNull()*. You'll see these methods used shortly.

**Note** You'll find a completed VS .NET example project for this section in the *StronglyTypedDataSet2* directory. You can open this project in VS .NET by selecting File > Open > Project and opening the *WindowsApplication4.csproj* file. You'll need to change the *ConnectionString* property of the *sqlConnection1* object to connect to your Northwind database.

The *Form1\_Load()* method of the form in the example project shows how to add, modify, and remove a row to a strongly typed *DataSet* object named *myDataSet1*. You can see the steps that accomplish these tasks in the following *Form1\_Load()* method:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // populate the DataSet with the CustomerID, CompanyName,
    // and Address columns from the Customers table
    sqlConnection1.Open();
    sqlDataAdapter1.Fill(myDataSet1, "Customers");

    // get the Customers DataTable
    MyDataSet.CustomersDataTable myDataTable =
        myDataSet1.Customers;

    // create a new DataRow in myDataTable using the
    // NewCustomersRow() method of myDataTable
    MyDataSet.CustomersRow myDataRow =
        myDataTable.NewCustomersRow();

    // set the CustomerID, CompanyName, and Address of myDataRow
    myDataRow.CustomerID = "J5COM";
    myDataRow.CompanyName = "J5 Company";
    myDataRow.Address = "1 Main Street";

    // add the new row to myDataTable using the
    // AddCustomersRow() method
    myDataTable.AddCustomersRow(myDataRow);

    // push the new row to the database using
    // the Update() method of sqlDataAdapter1
    sqlDataAdapter1.Update(myDataTable);

    // find the row using the FindByCustomerID()
    // method of myDataTable
    myDataRow = myDataTable.FindByCustomerID("J5COM");
    // modify the CompanyName and Address of myDataRow
    myDataRow.CompanyName = "Widgets Inc.";
    myDataRow.Address = "1 Any Street";
}
```

```
// push the modification to the database
sqlDataAdapter1.Update(myDataTable);

// display the DataRow objects in myDataTable
// in the listView1 object
foreach (MyDataSet.CustomersRow myDataRow2 in myDataTable.Rows)
{
    listView1.Items.Add(myDataRow2.CustomerID);
    listView1.Items.Add(myDataRow2.CompanyName);

    // if the Address is null, set Address to "Unknown"
    if (myDataRow2.IsAddressNull() == true)
    {
        myDataRow2.Address = "Unknown";
    }
    listView1.Items.Add(myDataRow2.Address);
}

// find and remove the new row using the
// FindByCustomerID() and RemoveCustomersRow() methods
// of myDataTable
myDataRow = myDataTable.FindByCustomerID("J5COM");
myDataTable.RemoveCustomersRow(myDataRow);

// push the delete to the database
sqlDataAdapter1.Update(myDataTable);

sqlConnection1.Close();
}
```

Feel free to compile and run the example form.

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

## Summary

In this chapter, you learned how to modify the rows in a `DataSet` and then push those changes to the database via a `DataAdapter`.

You saw how to add restrictions to a `DataTable` and its  `DataColumn` objects. This allows you to model the same restrictions placed on the database tables and columns in your `DataTable` and  `DataColumn` objects. By adding restrictions up front, you prevent bad data from being added to your `DataSet` to begin with, and this helps reduce the errors when attempting to push changes in your `DataSet` to the database.

Each row in a `DataTable` is stored in a `DataRow` object, and you saw how to find, filter, and sort the `DataRow` objects in a `DataTable` using the `Find()` method of a `DataTable`. You also learned how to filter and sort the `DataRow` objects in a `DataTable` using the `Select()` method.

You saw the steps required to add, modify, and remove `DataRow` objects from a `DataTable` and then push those changes to the database. To do this you must first set up your `DataAdapter` with `Command` objects containing appropriate SQL `INSERT`, `UPDATE`, and `DELETE` statements. You store these `Command` objects in your `DataAdapter` object's `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties. You push changes from your `DataSet` to the database using the `Update()` method of your `DataAdapter`. When you add, modify, or remove `DataRow` objects from your `DataSet` and then call the `Update()` method of your `DataAdapter`, the appropriate `InsertCommand`, `UpdateCommand`, or `DeleteCommand` is run to push your changes to the database.

**Concurrency** determines how multiple users' modifications to the same row are handled. With *optimistic concurrency*, you can modify a row in a database table only if no one else has modified that same row since you loaded it into your `DataSet`. This is typically the best type of concurrency to use because you don't want to overwrite someone else's changes. With "*last one wins*" concurrency, you can always modify a row-and your changes overwrite anyone else's changes. You typically want to avoid using "last one wins" concurrency.

You can get a `DataAdapter` object to call stored procedures to add, modify, and remove rows from the database. These procedures are called instead of the `INSERT`, `UPDATE`, and `DELETE` statements you've seen how to set in a `DataAdapter` object's `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties.

Supplying your own `INSERT`, `UPDATE`, and `DELETE` statements or stored procedures to push changes from your `DataSet` to the database means you have to write a lot of code. You can avoid writing this code by using a `CommandBuilder` object, which can automatically generate single-table `INSERT`, `UPDATE`, and `DELETE` commands that push the changes you make to a `DataSet` object to the database. These commands are then set in the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` properties of your `DataAdapter` object. When you then make changes to your `DataSet` and call the `Update()` method of your `DataAdapter`, the automatically generated command is run to push the changes to the database.

You also saw how to handle update failures and use transactions with a `DataSet`, and in the final section of this chapter you saw how to update the rows stored in a strongly typed data set.

In [Chapter 12](#), you'll learn how to navigate and modify related data.

## Chapter 12: Navigating and Modifying Related Data

### Overview

In [chapter 2](#), "Introduction to Databases," you saw how database tables can be related to each other through foreign keys. For example, the `OrderID` column of the `Orders` table is related to the `CustomerID` column of the `Customers` table through a foreign key. Because the `Orders` table depends on the `Customers` table, the `Orders` table is known as the *child table* and the `Customers` table as the *parent table*. The foreign key is said to define a *parent-child relationship* between the two tables.

Similarly, you can relate parent and child `DataTable` objects using a `ForeignKeyConstraint` or a `DataRelation` object. By default, when you add a `DataRelation` object to a `DataSet`, it actually adds a `UniqueConstraint` object to your parent `DataTable` along with a `ForeignKeyConstraint` to your child `DataTable`. The `ForeignKeyConstraint` ensures that each `DataRow` in your child `DataTable` has a matching `DataRow` in your parent `DataTable`.

In this chapter, you'll delve into the details of `UniqueConstraint`, `ForeignKeyConstraint`, and `DataRelation` objects. You'll also see how to navigate rows in related `DataTable` objects, make changes in related `DataTable` objects, and finally push those changes to the database.

**Tip** To improve performance when loading a `DataTable` with large numbers of `DataRow` objects, you should set the `EnforceConstraints` property of your `DataSet` to `false` for the duration of the load. This stops the constraints from being enforced. Remember to set `EnforceConstraints` back to the default of `true` at the end of the load.

Featured in this chapter:

- The `UniqueConstraint` class
- [Creating a `UniqueConstraint` object](#)
- The `ForeignKeyConstraint` class
- [Creating a `ForeignKeyConstraint` object](#)
- The `DataRelation` class
- Creating and using a `DataRelation` object
- Adding, updating, and deleting related rows
- Issues when updating the primary key of a parent row
- Nested XML
- Defining a relationship using Visual Studio .NET

## The *UniqueConstraint* Class

You use an object of the `UniqueConstraint` class to ensure that a `DataTable` value, or combination of `DataTable` values, is unique for each `DataRow` in a `DataTable`. The `UniqueConstraint` class is derived from the `System.Data.Constraint` class. [Table 12.1](#) shows the `UniqueConstraint` properties.

**Table 12.1: UniqueConstraint PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
Columns	<code>DataColumn[]</code>	Gets the array of <code>DataColumn</code> objects for the <code>UniqueConstraint</code> .
ConstraintName	<code>string</code>	Gets the name of the <code>UniqueConstraint</code> .
ExtendedProperties	<code>PropertyCollection</code>	Gets the <code>PropertyCollection</code> object that you can use to store strings of additional information.
IsPrimaryKey	<code>bool</code>	Gets a <code>bool</code> that indicates whether the <code>UniqueConstraint</code> is a primary key.
Table	<code>DataTable</code>	Gets the <code>DataTable</code> on which the <code>UniqueConstraint</code> was created.

## Creating a *UniqueConstraint* Object

In this section, you'll learn how to create a *UniqueConstraint* object. The *UniqueConstraint* constructor is overloaded as follows:

```
UniqueConstraint(DataColumn myDataColumn)
UniqueConstraint(DataColumn[] myDataColumns)
UniqueConstraint(DataColumn myDataColumn, bool isPrimaryKey)
UniqueConstraint(DataColumn[] myDataColumns, bool isPrimaryKey)
UniqueConstraint(string constraintName, DataColumn myDataColumn)
UniqueConstraint(string constraintName, DataColumn[] myDataColumns)
UniqueConstraint(string constraintName, DataColumn myDataColumn, bool isPrimaryKey)
UniqueConstraint(string constraintName, DataColumn[] myDataColumns,
    bool isPrimaryKey)
UniqueConstraint(string constraintName, string[] columnNames,
    bool isPrimaryKey)
```

where

- **myDataColumn** and **myDataColumns** are the *DataColumn* objects that you want to ensure are unique.
- **constraintName** is the name you want to assign to the *ConstraintName* property of your *UniqueConstraint*.
- **isPrimaryKey** indicates whether your *UniqueConstraint* is for a primary key.

Before creating and adding a *UniqueConstraint* to a *DataTable*, you first need a *DataTable*. The following example creates and populates two *DataTable* objects named *customersDT* and *ordersDT* (the *ordersDT* object will be used later in the section "[Creating a ForeignKeyConstraint Object](#)"):

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT CustomerID, CompanyName " +
    "FROM Customers " +
    "WHERE CustomerID = 'ALFKI' " +
    "SELECT OrderID, CustomerID " +
    "FROM Orders " +
    "WHERE CustomerID = 'ALFKI';";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet);
mySqlConnection.Close();
myDataSet.Tables["Table"].TableName = "Customers";
myDataSet.Tables["Table1"].TableName = "Orders";
DataTable customersDT = myDataSet.Tables["Customers"];
DataTable ordersDT = myDataSet.Tables["Orders"];
```

The following example creates a *UniqueConstraint* object on the *CustomerID* *DataColumn* of the *customersDT* *DataTable*, and then adds this *UniqueConstraint* to *customersDT*; notice that the third parameter to the *UniqueConstraint* constructor is set to *true*, indicating that the constraint is for a primary key:

```
UniqueConstraint myUC =
    new UniqueConstraint(
        "UniqueConstraintCustomerID",
        customersDT.Columns["CustomerID"],
        true
    );
customersDT.Constraints.Add(myUC);
```

**Note** To successfully add a *UniqueConstraint* to the *DataColumn* of a *DataTable*, the *DataColumn* value in each *DataRow* object in the *DataTable* must be unique.

The final example retrieves the constraint just added to *customersDT* and displays its properties:

```
myUC =
    (UniqueConstraint) customersDT.Constraints["UniqueConstraintCustomerID"]; Console.WriteLine("Columns:");
foreach (DataColumn myDataColumn in myUC.Columns)
{
    Console.WriteLine(" " + myDataColumn);
}
Console.WriteLine("myUC.ConstraintName = " + myUC.ConstraintName);
Console.WriteLine("myUC.IsPrimaryKey = " + myUC.IsPrimaryKey);
```

```
Console.WriteLine("myUC.Table = " + myUC.Table);
```

This example displays the following output:

Columns:

```
CustomerID  
myUC.ConstraintName = UniqueConstraintCustomerID  
myUC.IsPrimaryKey = True  
myUC.Table = Customers
```

The `IsPrimaryKey` property is `true` because the constraint is for a primary key; this was specified in the constructor when the constraint was created earlier.

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## The *ForeignKeyConstraint* Class

You use an object of the `ForeignKeyConstraint` class to represent a foreign key constraint between two `DataTable` objects. This ensures that each `DataRow` in your child `DataTable` has a matching `DataRow` in your parent `DataTable`. The `ForeignKeyConstraint` class is derived from the `System.Data.Constraint` class. [Table 12.2](#) shows the `ForeignKeyConstraint` properties.

**Table 12.2: `ForeignKeyConstraint` PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
<code>AcceptRejectRule</code>	<code>AcceptRejectRule</code>	<p>Gets or sets the <code>AcceptRejectRule</code> that indicates the action that is to take place when the <code>AcceptChanges()</code> method of the <code>DataTable</code> is called.</p> <p>The members of the <code>System.Data.AcceptRejectRule</code> enumeration are</p> <ul style="list-style-type: none"> <li>• <code>Cascade</code>, which indicates that the changes to the <code>DataRow</code> objects in the parent <code>DataTable</code> are also made in the child <code>DataTable</code>.</li> <li>• <code>None</code>, which indicates that no action takes place.</li> </ul> <p>The default is <code>None</code>.</p>
<code>Columns</code>	<code>DataColumn[]</code>	Gets the array of <code>DataColumn</code> objects from the child <code>DataTable</code> .
<code>ConstraintName</code>	<code>string</code>	Gets the name of the <code>UniqueConstraint</code> object.
<code>DeleteRule</code>	<code>Rule</code>	<p>Gets or sets the <code>Rule</code> that indicates the action that is to take place when a <code>DataRow</code> in the parent <code>DataTable</code> is deleted.</p> <p>The members of the <code>System.Data.Rule</code> enumeration are</p> <ul style="list-style-type: none"> <li>• <code>Cascade</code>, which indicates that the delete or update to the <code>DataRow</code> objects in the parent <code>DataTable</code> is also made in the child <code>DataTable</code>.</li> <li>• <code>None</code>, which indicates that no action takes place.</li> <li>• <code>SetDefault</code>, which indicates that the <code>DataColumn</code> values in the child <code>DataTable</code> are to be set to the value in the <code>DefaultValue</code> property of the <code>DataColumn</code>.</li> <li>• <code>SetNull</code>, which indicates that the <code>DataColumn</code> values in the child <code>DataTable</code> are to be set to <code>DBNull</code>.</li> </ul> <p>The default is <code>Cascade</code>.</p>

ExtendedProperties	PropertyCollection	Gets the <code>PropertyCollection</code> object that you can use to store strings of additional information.
RelatedColumns	DataColumn[ ]	Gets the array of <code>DataColumn</code> objects in the parent <code>DataTable</code> for the <code>UniqueConstraint</code> .
RelatedTable	DataTable	Gets the parent <code>DataTable</code> for the <code>UniqueConstraint</code> .
Table	DataTable	Gets the child <code>DataTable</code> to which the <code>UniqueConstraint</code> belongs.
UpdateRule	Rule	Gets or sets the <code>Rule</code> that indicates the action that is to take place when a <code>DataRow</code> in the parent <code>DataTable</code> is updated. See the <code>DeleteRule</code> property for the members of the <code>Rule</code> enumeration. The default is <code>Cascade</code> .

**Team LiB****◀ PREVIOUS****NEXT ▶**

## Creating a *ForeignKeyConstraint* Object

The *ForeignKeyConstraint* constructor is overloaded as follows:

```
ForeignKeyConstraint(DataColumn parentDataColumn, DataColumn childDataColumn)
ForeignKeyConstraint(DataColumn[] parentDataColumns,
    DataColumn[] childDataColumns)
ForeignKeyConstraint(string constraintName, DataColumn parentDataColumn,
    DataColumn childDataColumn)
ForeignKeyConstraint(string constraintName,
    DataColumn[] parentDataColumns, DataColumn[] childDataColumns)
ForeignKeyConstraint(string constraintName, string parentDataTableName,
    string[] parentDataColumnNames, string[] childDataColumnNames,
    AcceptRejectRule acceptRejectRule, Rule deleteRule, Rule updateRule)
```

where

- **parentDataColumn and parentDataColumns** are the *DataTable* objects in the parent *DataTable*.
- **childDataColumn and childDataColumns** are the *DataTable* objects in the child *DataTable*.
- **constraintName** is the name you want to assign to the *ConstraintName* property of your *ForeignKeyConstraint*.
- **parentDataTableName** is the name of the parent *DataTable*.
- **parentDataColumnNames and childDataColumnNames** contain the names of the *DataTable* objects in the parent and child *DataTable* objects.
- **acceptRejectRule, deleteRule, and updateRule** are the various rules for the *ForeignKeyConstraint*.

Earlier in the section "[Creating a UniqueConstraint Object](#)," you saw a code example that created two *DataTable* objects named *customersDT* and *ordersDT*. The following example creates a *ForeignKeyConstraint* object on the *CustomerID* *DataTable* of *ordersDT* to the *CustomerID* *DataTable* of *customersDT*:

```
ForeignKeyConstraint myFKC =
    new ForeignKeyConstraint(
        "ForeignKeyConstraintCustomersOrders",
        customersDT.Columns[ "CustomerID" ],
        ordersDT.Columns[ "CustomerID" ]
    );
ordersDT.Constraints.Add(myFKC);
```

Notice that the *ForeignKeyConstraint* is added to *ordersDT* using the *Add()* method.

**Note** To successfully add a *ForeignKeyConstraint* to a *DataTable*, each *DataTable* must have a matching *DataTable* value in the child *DataTable*.

The next example retrieves the constraint just added to *ordersDT* and displays its properties:

```
myFKC =
    (ForeignKeyConstraint)
        ordersDT.Constraints[ "ForeignKeyConstraintCustomersOrders" ];
Console.WriteLine( "myFKC.AcceptRejectRule = " + myFKC.AcceptRejectRule );
Console.WriteLine( "Columns:" );
foreach (DataColumn myDataColumn in myFKC.Columns)
{
    Console.WriteLine( "" + myDataColumn );
}
```

```
Console.WriteLine("myFKC.ConstraintName = " + myFKC.ConstraintName);
Console.WriteLine("myFKC.DeleteRule = " + myFKC.DeleteRule);
Console.WriteLine("RelatedColumns:");
foreach (DataColumn related DataColumn in myFKC.RelatedColumns)
{
    Console.WriteLine(" "+ related DataColumn);
}
Console.WriteLine("myFKC.RelatedTable = " + myFKC.RelatedTable);
Console.WriteLine("myFKC.Table = " + myFKC.Table);
Console.WriteLine("myFKC.UpdateRule = " + myFKC.UpdateRule);
```

This example displays the following output:

```
myFKC.AcceptRejectRule = None
Columns:
    CustomerID
myFKC.ConstraintName = ForeignKeyConstraintCustomersOrders
myFKC.DeleteRule = Cascade
RelatedColumns:
    CustomerID
myFKC.RelatedTable = Customers
myFKC.Table = Orders
myFKC.UpdateRule = Cascade
```

**Note** You'll find all the code examples shown in this section and the [previous section](#), "Creating a *UniqueConstraint* Object," in the *AddConstraints.cs* program. The listing is omitted from this book for brevity.

Team LiB

◀ PREVIOUS

NEXT ▶

## The *DataRelation* Class

You use an object of the `DataRelation` class to represent a relationship between two `DataTable` objects. You use a `DataRelation` object to model parent-child relationships between two database tables. By default, when you create a `DataRelation`, a `UniqueConstraint` and `ForeignKeyConstraint` are automatically added to your parent and child `DataTable` objects. [Table 12.3](#) shows the `DataRelation` properties.

**Table 12.3: DataRelation PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
ChildColumns	<code> DataColumn[]</code>	Gets the array of child <code> DataColumn</code> objects.
ChildKeyConstraint	<code> ForeignKeyConstraint</code>	Gets the <code> ForeignKeyConstraint</code> object for the <code>DataRelation</code> .
ChildTable	<code> DataTable</code>	Gets the child <code> DataTable</code> object.
DataSet	<code> DataSet</code>	Gets the <code> DataSet</code> to which the <code>DataRelation</code> belongs.
ExtendedProperties	<code> PropertyCollection</code>	Gets the <code> PropertyCollection</code> object that you can use to store strings of additional information.
Nested	<code> bool</code>	Gets or sets a <code> bool</code> value that indicates whether the <code>DataRelation</code> objects are nested. This is useful when defining hierarchical relationships in XML. The default is false.
ParentColumns	<code> DataColumn[]</code>	Gets the array of parent <code> DataColumn</code> objects.
ParentKeyConstraint	<code> UniqueConstraint</code>	Gets the <code> UniqueConstraint</code> object that ensures that <code> DataColumn</code> values in the parent <code> DataTable</code> are unique.
ParentTable	<code> DataTable</code>	Gets the parent <code> DataTable</code> object.
RelationName	<code> string</code>	Gets the name of the <code>DataRelation</code> object.

## Creating and Using a *DataRelation* Object

In this section, you'll learn how to create a *DataRelation* object to define a relationship between two *DataTable* objects that hold some rows from the *Customers* and *Orders* tables. As you know, the *CustomerID* column of the child *Orders* table is a foreign key that links to the *CustomerID* column of the parent *Customers* table.

Once you've created a *DataRelation*, you can use the *GetChildRows()* method of a *DataRow* object in the parent *DataTable* to obtain the corresponding *DataRow* objects from the child *DataTable*. By "corresponding," I mean the rows that have matching values in the foreign key  *DataColumn* objects. You can also use the *GetParentRow()* method of a *DataRow* in the child *DataTable* to obtain the corresponding *DataRow* in the parent *DataTable*.

Before creating and adding a *DataRelation* to a *DataSet*, you first need a *DataSet*. The following example creates and populates a *DataSet* with two *DataTable* objects named *customersDT* and *ordersDT*; notice that the top two rows from the *Customers* table along with the corresponding rows from the *Orders* table are retrieved:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT TOP 2 CustomerID, CompanyName " +
    "FROM Customers " +
    "ORDER BY CustomerID;" +
    "SELECT OrderID, CustomerID " +
    "FROM Orders " +
    "WHERE CustomerID IN (" +
    "    SELECT TOP 2 CustomerID " +
    "    FROM Customers " +
    "    ORDER BY CustomerID" +
    ")";
```

```
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet);
mySqlConnection.Close();
myDataSet.Tables["Table"].TableName = "Customers";
myDataSet.Tables["Table1"].TableName = "Orders";
DataTable customersDT = myDataSet.Tables["Customers"];
DataTable ordersDT = myDataSet.Tables["Orders"];
```

You'll see how to create a *DataRelation* that defines a relationship between the *customersDT* and *ordersDT* *DataTable* objects next.

**Note** You'll find all the code examples shown in this section in the *CreateDataRelation.cs* program.

## Creating the *DataRelation*

The *DataRelation* constructor is overloaded as follows:

```
DataRelation(string dataRelationName,
    DataColumn parent DataColumn, DataColumn child DataColumn)
DataRelation(string dataRelationName,
    DataColumn[] parent DataColumns, DataColumn[] child DataColumns)
DataRelation(string dataRelationName,
    DataColumn parent DataColumn, DataColumn child DataColumn, bool createConstraints)
DataRelation(string dataRelationName,
    DataColumn[] parent DataColumns, DataColumn[] child DataColumns,
    bool createConstraints)
```

```
DataRelation(string dataRelationName,
    string parentDataTableName, string childDataTableName,
    string[] parentDataColumnNames, string[] childDataColumnNames,
    bool nested)
```

where

- **dataRelationName** is the name you want to assign to the `RelationName` property of your `DataRelation`.
- **parentDataColumn** and **parentDataColumns** are the  `DataColumn` objects in the parent `DataTable`.
- **childDataColumn** and **childDataColumns** are the  `DataColumn` objects in the child `DataTable`.
- **createConstraints** indicates whether you want a `UniqueConstraint` added to the parent `DataTable` and a `ForeignKeyConstraint` added to the child `DataTable` automatically (the default is `true`).
- **parentDataTableName** and **childDataTableName** are the names of the parent and child `DataTable` objects.
- **parentDataColumnNames** and **childDataColumnNames** contain the names of the  `DataColumn` objects in the parent and child `DataTable` objects.
- **nested** indicates whether the relationships are nested.

The following example creates a `DataRelation` object named `customersOrdersDataRel`:

```
DataRelation customersOrdersDataRel =
    new DataRelation(
        "CustomersOrders",
        customersDT.Columns[ "CustomerID" ],
        ordersDT.Columns[ "CustomerID" ]
    );
```

The name assigned to the `RelationName` property of `customersOrdersDataRel` is `CustomersOrders`, the parent  `DataColumn` is `customersDT.Columns[ "CustomerID" ]`, and the child  `DataColumn` is `ordersDT.Columns[ "CustomerID" ]`.

Next, `customersOrdersDataRel` must be added to `myDataSet`. You access the `DataRelation` objects in a `DataSet` object through its `Relationships` property. The `Relationships` property returns an object of the `DataRelationCollection` class, which is a collection of `DataRelation` objects. To add a `DataRelation` object to the `DataRelationCollection` object of a `DataSet`, you call the `Add( )` method through the `Relationships` property of your `DataSet`.

The following example uses the `Add( )` method to add `customersOrdersDataRel` to `myDataSet`:

```
myDataSet.Relations.Add(
    customersOrdersDataRel
);
```

The `Add( )` method is overloaded, and you can also use the following version of the `Add( )` method to create and add a `DataRelation` object to `myDataSet`:

```
myDataSet.Relations.Add(
    "CustomersOrders",
    customersDT.Columns[ "CustomerID" ],
    ordersDT.Columns[ "CustomerID" ]
);
```

This example does the same thing as the two earlier examples. The first parameter to the `Add( )` method is a string containing the name you want to assign to the `RelationName` property of the `DataRelation`. The second and third parameters of the relationship are

the DataColumn objects from the parent and child DataTable objects.

## Examining the Constraints Created by the *DataRelation*

By default, when you create a DataRelation, a UniqueConstraint and ForeignKeyConstraint are automatically added to your parent and child DataTable objects. You can get the UniqueConstraint from a DataRelation using its ParentKeyConstraint property. For example:

```
UniqueConstraint myUC =
    customersOrdersDataRel.ParentKeyConstraint;
```

You can view the properties of the myUC UniqueConstraint object using the following code:

```
Console.WriteLine("Columns:");
foreach (DataColumn myDataColumn in myUC.Columns)
{
    Console.WriteLine(" " + myDataColumn);
}
Console.WriteLine("myUC.ConstraintName = " + myUC.ConstraintName);
Console.WriteLine("myUC.IsPrimaryKey = " + myUC.IsPrimaryKey);
Console.WriteLine("myUC.Table = " + myUC.Table);
```

The output from this code is as follows:

```
Columns:
CustomerID
myUC.ConstraintName = Constraint1
myUC.IsPrimaryKey = False
myUC.Table = Customers
```

You can get the ForeignKeyConstraint from a DataRelation using its ChildKeyConstraint property. For example:

```
ForeignKeyConstraint myFKC =
    customersOrdersDataRel.ChildKeyConstraint;
```

You can view the properties of myFKC using the following code:

```
Console.WriteLine("myFKC.AcceptRejectRule = " + myFKC.AcceptRejectRule);
Console.WriteLine("Columns:");
foreach (DataColumn myDataColumn in myFKC.Columns)
{
    Console.WriteLine(" " + myDataColumn);
}
Console.WriteLine("myFKC.ConstraintName = " + myFKC.ConstraintName);
Console.WriteLine("myFKC.DeleteRule = " + myFKC.DeleteRule);
Console.WriteLine("RelatedColumns:");
foreach (DataColumn relatedDataColumn in myFKC.RelatedColumns)
{
    Console.WriteLine(" " + relatedDataColumn);
}
Console.WriteLine("myFKC.RelatedTable = " + myFKC.RelatedTable);
Console.WriteLine("myFKC.Table = " + myFKC.Table);
Console.WriteLine("myFKC.UpdateRule = " + myFKC.UpdateRule);
```

The output from this code is as follows:

```
myFKC.AcceptRejectRule = None
Columns:
CustomerID
myFKC.ConstraintName = CustomersOrders
myFKC.DeleteRule = Cascade
```

**RelatedColumns:**

```

CustomerID
myFKC.RelatedTable = Customers
myFKC.Table = Orders
myFKC.UpdateRule = Cascade

```

The DeleteRule and UpdateRule properties are set to Cascade by default. Because DeleteRule is set to Cascade, when you delete a DataRow in the parent DataTable, then any corresponding DataRow objects in the child DataTable are also deleted. Because UpdateRule is set to Cascade, when you change the DataColumn in the parent DataTable on which the ForeignKeyConstraint was created, then the same change is also made in any corresponding DataRow objects in the child DataTable. You'll learn more about this later in the section "Issues When Updating the Primary Key of a Parent Row."

## Navigating *DataRow* Objects in the Parent and Child *DataTable* Objects

To navigate the DataRow objects in related DataTable objects, you use the GetChildRows() and GetParentRow() methods of a DataRow.

### Using the *GetChildRows()* Method

You use the GetChildRows() method to get the related child DataRow objects from the parent DataRow. For example, the following code displays the parent DataRow objects from the customersDT DataTable and their related child DataRow objects from the ordersDT DataTable:

```

foreach (DataRow customerDR in customersDT.Rows)
{
    Console.WriteLine("\nCustomerID = " + customerDR[ "CustomerID" ]);
    Console.WriteLine("CompanyName = " + customerDR[ "CompanyName" ]);

    DataRow[] ordersDRs = customerDR.GetChildRows("CustomersOrders");
    Console.WriteLine("This customer placed the following orders:");
    foreach (DataRow orderDR in ordersDRs)
    {
        Console.WriteLine("OrderID = " + orderDR[ "OrderID" ]);
    }
}

```

The output from this code is as follows:

```

CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
This customer placed the following orders:
    OrderID = 10643
    OrderID = 10692
    OrderID = 10702
    OrderID = 10835
    OrderID = 10952
    OrderID = 11011

CustomerID = ANATR
CompanyName = Ana Trujillo Emparedados y helados
This customer placed the following orders:
    OrderID = 10308
    OrderID = 10625
    OrderID = 10759
    OrderID = 10926

```

### Using the *GetParentRow()* Method

You use the GetParentRow() method to get the parent DataRow from the child DataRow. For example, the following code

displays the first child DataRow from ordersDT and its related parent DataRow from customersDT:

```
DataRow parentCustomerDR = ordersDT.Rows[0].GetParentRow("CustomersOrders");
Console.WriteLine("\nOrder with OrderID of " + ordersDT.Rows[0]["OrderID"] +
    " was placed by the following customer:");
Console.WriteLine("CustomerID = " + parentCustomerDR["CustomerID"]);
```

The output from this code is as follows:

```
Order with OrderID of 10643 was placed by the following customer:
CustomerID = ALFKI
```

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Adding, Updating, and Deleting Related Rows

In this section, you'll learn how to make changes in `DataTable` objects that store rows from the `Customers` and `Orders` tables. These tables are related through the `CustomerID` foreign key. As you'll see, you must push changes to the underlying database tables in a specific order. If you don't, your program will throw an exception.

**Note** You'll find all the code examples shown in this section in the `ModifyingRelatedData.cs` program.

### Setting Up the `DataAdapter` Objects

You'll need two `DataAdapter` objects:

- One to work with the `Customers` table, which will be named `customersDA`.
- One to work with the `Orders` table, which will be named `ordersDA`.

Let's take a look at setting up these two `DataAdapter` objects.

### Setting Up the `customersDA` `DataAdapter`

The following code creates and sets up a `DataAdapter` named `customersDA` that contains the necessary `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements to access the `Customers` table:

```
SqlDataAdapter customersDA = new SqlDataAdapter();

// create a SqlCommand object to hold the SELECT
SqlCommand customersSelectCommand = mySqlConnection.CreateCommand();
customersSelectCommand.CommandText =
    "SELECT CustomerID, CompanyName " +
    "FROM Customers";

// create a SqlCommand object to hold the INSERT
SqlCommand customersInsertCommand = mySqlConnection.CreateCommand();
customersInsertCommand.CommandText =
    "INSERT INTO Customers (" +
    "    CustomerID, CompanyName " +
    ") VALUES (" +
    "    @CustomerID, @CompanyName" +
    ")";

customersInsertCommand.Parameters.Add("@CustomerID", SqlDbType.NChar,
    5, "CustomerID");
customersInsertCommand.Parameters.Add("@CompanyName", SqlDbType.NVarChar,
    40, "CompanyName");

// create a SqlCommand object to hold the UPDATE
SqlCommand customersUpdateCommand = mySqlConnection.CreateCommand();
customersUpdateCommand.CommandText =
    "UPDATE Customers " +
    "SET " +
    "    CompanyName = @NewCompanyName " +
    "WHERE CustomerID = @OldCustomerID " +
    "AND CompanyName = @OldCompanyName";
customersUpdateCommand.Parameters.Add("@NewCompanyName",
    SqlDbType.NVarChar, 40, "CompanyName");
customersUpdateCommand.Parameters.Add("@OldCustomerID",
    SqlDbType.NChar, 5, "CustomerID");
customersUpdateCommand.Parameters.Add("@OldCompanyName",
```

```

SqlDbType.NVarChar, 40, "CompanyName");
customersUpdateCommand.Parameters["@OldCustomerID"].SourceVersion =
    DataRowVersion.Original;
customersUpdateCommand.Parameters["@OldCompanyName"].SourceVersion =
    DataRowVersion.Original;

// create a SqlCommand object to hold the DELETE
SqlCommand customersDeleteCommand = mySqlConnection.CreateCommand();
customersDeleteCommand.CommandText =
    "DELETE FROM Customers " +
    "WHERE CustomerID = @OldCustomerID " +
    "AND CompanyName = @OldCompanyName";
customersDeleteCommand.Parameters.Add("@OldCustomerID",
    SqlDbType.NChar, 5, "CustomerID");
customersDeleteCommand.Parameters.Add("@OldCompanyName",
    SqlDbType.NVarChar, 40, "CompanyName");
customersDeleteCommand.Parameters["@OldCustomerID"].SourceVersion =
    DataRowVersion.Original;
customersDeleteCommand.Parameters["@OldCompanyName"].SourceVersion =
    DataRowVersion.Original;

// set the customersDA properties
// to the SqlCommand objects previously created
customersDA.SelectCommand = customersSelectCommand;
customersDA.InsertCommand = customersInsertCommand;
customersDA.UpdateCommand = customersUpdateCommand;
customersDA.DeleteCommand = customersDeleteCommand;

```

Notice that the UPDATE statement modifies only the `CompanyName` column value; it doesn't modify the `CustomerID` primary key column value. You'll learn about the issues involved with updating a primary key column value later in the section "Issues When Updating the Primary Key of a Parent Row."

The `ModifyingRelatedData.cs` program contains a method named `SetupCustomersDA()` that performs the previous code.

## Setting Up the `ordersDA DataAdapter`

The following code creates and sets up a `DataAdapter` object named `ordersDA` that contains the necessary `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements to access the `Orders` table; notice that the `ordersInsertCommand` contains both an `INSERT` statement and a `SELECT` statement to retrieve the new `OrderID` column, which is an identity column that has a value automatically generated by the database:

```

SqlDataAdapter ordersDA = new SqlDataAdapter();

// create a SqlCommand object to hold the SELECT
SqlCommand ordersSelectCommand = mySqlConnection.CreateCommand();
ordersSelectCommand.CommandText =
    "SELECT OrderID, CustomerID, ShipCountry " +
    "FROM Orders";

// create a SqlCommand object to hold the INSERT
SqlCommand ordersInsertCommand = mySqlConnection.CreateCommand();
ordersInsertCommand.CommandText =
    "INSERT INTO Orders (" +
    "    CustomerID, ShipCountry " +
    ") VALUES (" +
    "    @CustomerID, @ShipCountry" +
    ");" +
    "SELECT @OrderID = SCOPE_IDENTITY();";

```

```

ordersInsertCommand.Parameters.Add("@CustomerID", SqlDbType.NChar,
    5, "CustomerID");
ordersInsertCommand.Parameters.Add("@ShipCountry", SqlDbType.NVarChar,
    15, "ShipCountry");
ordersInsertCommand.Parameters.Add("@OrderID", SqlDbType.Int,
    0, "OrderID");
ordersInsertCommand.Parameters["@OrderID"].Direction =
    ParameterDirection.Output;

// create a SqlCommand object to hold the UPDATE
SqlCommand ordersUpdateCommand = mySqlConnection.CreateCommand();
ordersUpdateCommand.CommandText =
    "UPDATE Orders " +
    "SET " +
    " ShipCountry = @NewShipCountry " +
    "WHERE OrderID = @OldOrderID " +
    "AND CustomerID = @OldCustomerID " +
    "AND ShipCountry = @OldShipCountry";
ordersUpdateCommand.Parameters.Add("@NewShipCountry",
    SqlDbType.NVarChar, 15, "ShipCountry");
ordersUpdateCommand.Parameters.Add("@OldOrderID",
    SqlDbType.Int, 0, "OrderID");
ordersUpdateCommand.Parameters.Add("@OldCustomerID",
    SqlDbType.NChar, 5, "CustomerID");
ordersUpdateCommand.Parameters.Add("@OldShipCountry",
    SqlDbType.NVarChar, 15, "ShipCountry");
ordersUpdateCommand.Parameters["@OldOrderID"].SourceVersion =
    DataRowVersion.Original;
ordersUpdateCommand.Parameters["@OldCustomerID"].SourceVersion =
    DataRowVersion.Original;
ordersUpdateCommand.Parameters["@OldShipCountry"].SourceVersion =
    DataRowVersion.Original;

// create a SqlCommand object to hold the DELETE
SqlCommand ordersDeleteCommand = mySqlConnection.CreateCommand();
ordersDeleteCommand.CommandText =
    "DELETE FROM Orders " +
    "WHERE OrderID = @OldOrderID " +
    "AND CustomerID = @OldCustomerID " +
    "AND ShipCountry = @OldShipCountry";
ordersDeleteCommand.Parameters.Add("@OldOrderID", SqlDbType.Int,
    0, "OrderID");
ordersDeleteCommand.Parameters.Add("@OldCustomerID",
    SqlDbType.NChar, 5, "CustomerID");
ordersDeleteCommand.Parameters.Add("@OldShipCountry",
    SqlDbType.NVarChar, 15, "ShipCountry");
ordersDeleteCommand.Parameters["@OldOrderID"].SourceVersion =
    DataRowVersion.Original;
ordersDeleteCommand.Parameters["@OldCustomerID"].SourceVersion =
    DataRowVersion.Original;
ordersDeleteCommand.Parameters["@OldShipCountry"].SourceVersion =
    DataRowVersion.Original;

// set the ordersDA properties
// to the SqlCommand objects previously created
ordersDA.SelectCommand = ordersSelectCommand;
ordersDA.InsertCommand = ordersInsertCommand;
ordersDA.UpdateCommand = ordersUpdateCommand;
ordersDA.DeleteCommand = ordersDeleteCommand;

```

The `ModifyingRelatedData.cs` program contains a method named `SetupOrdersDA()` that performs the previous code.

## Creating and Populating a *DataSet*

Next, the following example creates and populates a `DataSet` named `myDataSet` with the rows from the `Customers` and `Orders` tables using `customersDA` and `ordersDA`:

```
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
customersDA.Fill(myDataSet, "Customers");
ordersDA.Fill(myDataSet, "Orders");
mySqlConnection.Close();
DataTable customersDT = myDataSet.Tables["Customers"];
DataTable ordersDT = myDataSet.Tables["Orders"];
```

Notice that the `DataTable` objects are named `customersDT` and `ordersDT`.

The following examples set the `PrimaryKey` properties of `customersDT` and `ordersDT`:

```
customersDT.PrimaryKey =
    new DataColumn[]
    {
        customersDT.Columns["CustomerID"]
    };

ordersDT.PrimaryKey =
    new DataColumn[]
    {
        ordersDT.Columns["OrderID"]
    };
```

The following example sets up the `OrderID`  `DataColumn` of `ordersDT` as an identity:

```
ordersDT.Columns["OrderID"].AllowDBNull = false;
ordersDT.Columns["OrderID"].AutoIncrement = true;
ordersDT.Columns["OrderID"].AutoIncrementSeed = -1;
ordersDT.Columns["OrderID"].AutoIncrementStep = -1;
ordersDT.Columns["OrderID"].ReadOnly = true;
ordersDT.Columns["OrderID"].Unique = true;
```

The final example adds a `DataRelation` to `myDataSet` that specifies a relationship between `customersDT` and `ordersDT` using the `CustomerID`  `DataColumn`:

```
DataRelation customersOrdersDataRel =
    new DataRelation(
        "CustomersOrders",
        customersDT.Columns["CustomerID"],
        ordersDT.Columns["CustomerID"]
    );
myDataSet.Relations.Add(
    customersOrdersDataRel
);
```

The `ModifyingRelatedData.cs` program performs the previous code in the `Main()` method.

## Adding *DataRow* Objects to `customersDT` and `ordersDT`

The following example adds a `DataRow` named `customerDR` to `customersDT`; notice that the `CustomerID` is set to J6COM:

```
DataRow customerDR = customersDT.NewRow();
```

```
customerDR[ "CustomerID" ] = "J6COM";
customerDR[ "CompanyName" ] = "J6 Company";
customersDT.Rows.Add(customerDR);
```

The next example adds a `DataRow` named `orderDR` to `ordersDT`; notice that the `CustomerID` is also set to `J6COM`, indicating that this is the child `DataRow` for the previous `DataRow` in `customersDT`:

```
DataRow orderDR = ordersDT.NewRow();
orderDR[ "CustomerID" ] = "J6COM";
orderDR[ "ShipCountry" ] = "England";
ordersDT.Rows.Add(orderDR);
```

Because the `OrderID`  `DataColumn` of `ordersDT` is set up as an identity, it will automatically be assigned the initial value of -1. When this `DataRow` is pushed to the database, the `SELECT` statement in `ordersDA` will set the `OrderID` to the identity value generated by the database for the new row in the `Orders` table. You'll see how to push the changes to the database shortly in the section "[Pushing Changes in `customersDT` and `ordersDT` to the Database](#)."

The `ModifyingRelatedData.cs` program performs the previous code in the `Main()` method.

## Updating `DataRow` Objects in `customersDT` and `ordersDT`

The following example updates the `CompanyName` in `customerDR` to `Widgets Inc.`:

```
customerDR[ "CompanyName" ] = "Widgets Inc.;"
```

The next example updates the `ShipCountry` in `orderDR` to `USA`:

```
orderDR[ "ShipCountry" ] = "USA";
```

The `ModifyingRelatedData.cs` program performs the previous code in the `Main()` method.

## Deleting `DataRow` Objects from `customersDT` and `ordersDT`

The following example deletes the `customerDR` `DataRow` from the `customersDT` `DataTable`:

```
customerDR.Delete();
```

Earlier in the section "[Examining the Constraints Created by the `DataRelation`](#)," you saw that a `ForeignKeyConstraint` is added to the child `DataTable` by default when a `DataRelation` object is added to a `DataSet`. You also saw that this `ForeignKeyConstraint` object's `DeleteRule` property is set to `Cascade` by default. This means that when the `DataRow` in the parent `DataTable` is deleted, so are the corresponding `DataRow` objects in the child `DataTable`. Therefore, in the previous example, when `customerDR` is deleted from `customersDT`, so is `orderDR` in `ordersDT`.

The `ModifyingRelatedData.cs` program performs the previous code in the `Main()` method.

## Pushing Changes in `customersDT` and `ordersDT` to the Database

In this section, you'll learn how to push the changes previously made in the `customersDT` and `ordersDT` `DataTable` objects to the database. When pushing changes to the database, you have to apply them in an order that satisfies the foreign key constraints in the related tables.

For example, a row in the `Customers` table with a `CustomerID` of `J6COM` must exist *before* a row with that `CustomerID` can be added to the `Orders` table. Similarly, you can't delete the row with a `CustomerID` of `J6COM` while there are rows with that `CustomerID` in the `Orders` table. Finally, of course, you can update only rows that already exist in a table.

Follow these steps when pushing the changes from `customersDT` and `ordersDT` to the database:

1. Push the `DataRow` objects added to `customersDT` to the `Customers` table.

2. Push the `DataRow` objects added to `ordersDT` to the `Orders` table.
3. Push the `DataRow` objects updated in `customersDT` to the `Customers` table.
4. Push the `DataRow` objects updated in `ordersDT` to the `Orders` table.
5. Delete the `DataRow` objects removed from `ordersDT` from the `Orders` table.
6. Delete the `DataRow` objects removed from `customersDT` from the `Customers` table.

To get the `DataRow` objects that have been added, updated, or deleted, you use the `Select()` method of a `DataTable`. The `Select()` method was covered in the [previous chapter](#), and one of the overloaded versions of this method is

```
DataRow[] Select(string filterExpression, string sortExpression,
    DataViewRowState myDataViewState)
```

where

- **`filterExpression`** specifies the rows to select.
- **`sortExpression`** specifies how the selected rows are to be ordered.
- **`myDataViewState`** specifies the row state of the rows to select. You can see the members of the `DataViewRowState` enumeration in [Table 11.8](#) of the [previous chapter](#).

To get the `DataRow` objects that have been added to the `customersDT` `DataTable`, you can use the following code that calls the `Select()` method:

```
DataRow[] newCustomersDRArray =
    customersDT.Select("", "", DataViewRowState.Added);
```

Notice the use of the `Added` constant from the `DataViewRowState` enumeration. This indicates that only the newly added `DataRow` objects in `customersDT` are to be returned and stored in `newCustomersDRArray`.

You can then push the `DataRow` objects in `newCustomersDRArray` to the `Customers` table in the database using the following call to the `Update()` method of the `customersDA` `DataAdapter`:

```
int numRows = customersDA.Update(newCustomersDRArray);
```

The `numRows` `int` is the number of rows added to the `Customers` table.

The following code uses the six steps shown earlier to push all the changes to the database; notice the different constants used from the `DataViewRowState` enumerator to get the required `DataRow` objects:

```
mySqlConnection.Open();

// push the new rows in customersDT to the database
Console.WriteLine("Pushing new rows in customersDT to database");
DataRow[] newCustomersDRArray =
    customersDT.Select("", "", DataViewRowState.Added);
int numRows = customersDA.Update(newCustomersDRArray);
Console.WriteLine("numRows = " + numRows);

// push the new rows in ordersDT to the database
Console.WriteLine("Pushing new rows in ordersDT to database");
DataRow[] newOrdersDRArray =
    ordersDT.Select("", "", DataViewRowState.Added);
numRows = ordersDA.Update(newOrdersDRArray);
Console.WriteLine("numRows = " + numRows);
```

```

// push the modified rows in customersDT to the database
Console.WriteLine("Pushing modified rows in customersDT to database");
DataRow[] modifiedCustomersDRArray =
    customersDT.Select("", "", DataViewRowState.ModifiedCurrent);
numOfRows = customersDA.Update(modifiedCustomersDRArray);
Console.WriteLine("numOfRows = " + numOfRows);

// push the modified rows in ordersDT to the database
Console.WriteLine("Pushing modified rows in ordersDT to database");
DataRow[] modifiedOrdersDRArray =
    ordersDT.Select("", "", DataViewRowState.ModifiedCurrent);
numOfRows = ordersDA.Update(modifiedOrdersDRArray);
Console.WriteLine("numOfRows = " + numOfRows);
// push the deletes in ordersDT to the database
Console.WriteLine("Pushing deletes in ordersDT to database");
DataRow[] deletedOrdersDRArray =
    ordersDT.Select("", "", DataViewRowState.Deleted);
numOfRows = ordersDA.Update(deletedOrdersDRArray);
Console.WriteLine("numOfRows = " + numOfRows);

// push the deletes in customersDT to the database
Console.WriteLine("Pushing deletes in customersDT to database");
DataRow[] deletedCustomersDRArray =
    customersDT.Select("", "", DataViewRowState.Deleted);
numOfRows = customersDA.Update(deletedCustomersDRArray);
Console.WriteLine("numOfRows = " + numOfRows);

mySqlConnection.Close();

```

The `ModifyingRelatedData.cs` program contains a method named `PushChangesToDatabase()` that uses the previous code.

One thing you'll notice about `ModifyingRelatedData.cs` is that it calls `PushChangesToDatabase()` immediately after performing the following steps in the `Main()` method:

1. Adding `DataRow` objects to `customersDT` and `ordersDT`.
2. Updating the new `DataRow` objects.
3. Deleting the new `DataRow` objects.

`PushChangesToDatabase()` is immediately called after each of these steps so that you can see the database activity as the program progresses. I could have simply called `PushChangesToDatabase()` once at the end of the three steps-but then you wouldn't see any changes to the database, because the new rows would have been deleted in step 3 prior to `PushChangesToDatabase()` being called.

**Tip** In your own programs, you're likely add, update, and delete many different rows in your `DataTable` objects, and so pushing the changes once at the end will be more efficient.

## Issues Involved When Updating the Primary Key of a Parent Row

In this section, you'll learn about the issues involved when attempting to update the primary key in a parent `DataTable`, and then pushing the update to the underlying database table. The issues occur when the child database table already contains rows that use the primary key you want to change in the parent table.

The examples in this section will use the `Customers` and `Orders` table, which are related through the foreign key on the `CustomerID` column of the `Orders` table to the `CustomerID` column of the `Customers` table.

As you'll learn, you're much better off not allowing changes to the primary key column of a table. If you allow changes to the primary key column, then as you'll see shortly, you can run into problems when pushing the change to the database. Instead, you should set the `ReadOnly` property to `true` for the primary key  `DataColumn` in your parent `DataTable`, and also set `ReadOnly` to `true` for the foreign key  `DataColumn` in your child `DataTable`. That prevents changes to the values in these  `DataColumn` objects.

If you *really* need to change the primary key and foreign key values, you should delete and then recreate the rows in the database with the new primary key and foreign key values.

You can control how updates and deletes are performed using the properties of the foreign key in the SQL Server database and also the `UpdateRule` and `DeleteRule` properties of a `ForeignKeyConstraint` object. You'll explore both of these items in the following sections.

## Controlling Updates and Deletes Using SQL Server

You can control how updates and deletes are performed using SQL Server by setting the properties of the foreign key. You set these properties using the Relationships tab of a database table's Properties dialog box. You open this dialog box in Enterprise Manager for the `Orders` table by performing the following steps:

1. Right-click the `Orders` table in the Tables node of Enterprise Manager.
2. Select Design Table from the pop-up list.
3. Press the Manage Relationships button in the toolbar of the Design Table dialog box.
4. Select the foreign key you want to examine in the Select relationship drop-down list.

[Figure 12.1](#) shows the Relationships tab for the foreign key named `FK_Orders_Customers` that contains the details of the foreign key between the `Orders` and `Customers` tables. As you can see, these two tables are related through a foreign key on the `CustomerID` column.



**Figure 12.1:** The Relationships tab for `FK_Orders_Customers`

The Cascade Update Related Fields check box indicates whether a change to a value in the primary key column of the primary key table (the parent table) is also made to the foreign key column of the corresponding rows of the foreign key table (the child table). For example, assume this box is checked and you changed the `CustomerID` in the row of the `Customers` table from `ALFKI` to `ANATR`; this would also cause the `CustomerID` column to change from `ALFKI` to `ANATR` in the rows of the `Orders` table.

Similarly, the Cascade Delete Related Records check box indicates whether deleting a row in the primary key table also deletes any related rows from the foreign key table. For example, assume this box is checked and you deleted the row with the `CustomerID` of `ANTON` from the `Customers` table; this would cause the rows with the `CustomerID` of `ANTON` to also be deleted from the `Orders` table.

**Note** Typically, you should leave both check boxes in their default unchecked state. If you check them, the database will make changes to the rows in the child table behind the scenes and as you'll see shortly, you'll run into problems when pushing changes from your `DataSet` to the database.

## Controlling Updates and Deletes Using the `UpdateRule` and `DeleteRule` Properties of a `ForeignKeyConstraint` Object

You can also control updates and deletes using the `UpdateRule` and `DeleteRule` properties of a `ForeignKeyConstraint` object. These properties are of the `System.Data.Rule` enumeration type; members of this type are shown in [Table 12.4](#).

**Table 12.4: Rule ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
<code>Cascade</code>	Indicates that the delete or update to the <code>DataRow</code> objects in the parent <code>DataTable</code> are also made in the child <code>DataTable</code> . This is the default.
<code>None</code>	Indicates that no action takes place.
<code>SetDefault</code>	Indicates that the <code> DataColumn</code> values in the child <code>DataTable</code> are to be set to the value in the <code>DefaultValue</code> property of the <code> DataColumn</code> .
<code>SetNull</code>	Indicates that the <code> DataColumn</code> values in the child <code>DataTable</code> are to be set to <code>DBNull</code> .

By default, `UpdateRule` is set to `Cascade`; therefore, when you change the  `DataColumn` in the parent `DataTable` on which the `ForeignKeyConstraint` was created, then the same change is also made in any corresponding `DataRow` objects in the child

`DataTable`. You should set `UpdateRule` to `None` in your program; otherwise, as you'll learn in the [next section](#), you'll run into problems when pushing changes from your `DataSet` to the database.

By default, `DeleteRule` is set to `Cascade`; therefore, when you delete a `DataRow` in the parent `DataTable`, any corresponding `DataRow` objects in the child `DataTable` are also deleted. This is fine, as long as you remember to push the deletes to the child table before you push the deletes to the parent table.

## Updating the Primary Key of a Parent Table and Pushing the Change to the Database

In this section you'll learn what happens if you attempt to update the primary key in a parent table when there are corresponding rows in the child table. Assume the following:

- There is a row in the `Customers` table with a `CustomerID` of J6COM. A copy of this row is stored in a `DataTable` named `customersDT`.
- There is a row in the `Orders` table that also has a `CustomerID` of J6COM. A copy of this row is stored in a `DataTable` named `ordersDT`.
- The `customersDT` and `ordersDT` `DataTable` objects are related to each other using the following `DataRelation`:

```
    DataRelation customersOrdersDataRel =
        new DataRelation(
            "CustomersOrders",
            customersDT.Columns[ "CustomerID" ],
            ordersDT.Columns[ "CustomerID" ]
        );
    myDataSet.Relations.Add(
        customersOrdersDataRel
    );
```

Now, the two settings for the Cascade Update Related Fields check box for `FK_Orders_Customers` are

- Unchecked, meaning that changes to the `CustomerID` primary key value in the `Customers` table are not cascaded to the `Orders` table. This is the default.
- Checked, meaning that changes to the `CustomerID` primary key value in the `Customers` table are cascaded to the `Orders` table.

In addition, the settings of interest for the `UpdateRule` property of the `ForeignKeyConstraint` object added when the earlier `DataRelation` was created are

- `Cascade`, meaning that changes to the `CustomerID`  `DataColumn` of `customersDT` are cascaded to `ordersDT`. This is the default.
- `None`, meaning that changes to the `CustomerID`  `DataColumn` of `customersDT` are not cascaded to `ordersDT`.

Let's examine the three most important cases that vary the checking of the Cascade Update Related Fields box and setting of the `UpdateRule` property to `Cascade` and then `None`.

**Note** You can use the `ModifyingRelatedData2.cs` program as the basis for trying out the three cases described in this section.

### First Case

Assume the following:

- Cascade Update Related Fields box is checked.

- `UpdateRule` is set to `Cascade`.

If you change the `CustomerID` `DataTable` from J6COM to J7COM and push the change to the database, then the change is made successfully in the `customersDT` and `ordersDT` `DataTable` objects and also in the `Customers` and `Orders` database tables.

This works as long as you use only the `OrderID` column in the `WHERE` clause of the `Command` object in the `UpdateCommand` property of your `DataAdapter`. For example:

```
ordersUpdateCommand.CommandText =
    "UPDATE Orders " +
    "SET " +
    "  CustomerID = @NewCustomerID " +
    "WHERE OrderID = @OldOrderID";
```

This `UPDATE` uses "last one wins" concurrency since *only* the `OrderID` primary key column is used in the `WHERE` clause (the old `CustomerID` column is left out of the `WHERE` clause). As mentioned in the [previous chapter](#), "last one wins" concurrency is bad because one user might overwrite a change made by another user.

If instead you also include the old `CustomerID` column value in the `WHERE` clause of the `UPDATE`, as shown in the following example,

```
ordersUpdateCommand.CommandText =
    "UPDATE Orders " +
    "SET " +
    "  CustomerID = @NewCustomerID " +
    "WHERE OrderID = @OldOrderID " +
    "AND CustomerID = @OldCustomerID";
```

then pushing the change to the database would fail because the original row in the `orders` table wouldn't be found. The original row wouldn't be found since the `CustomerID` has already been changed from J6COM to J7COM in the `Orders` table automatically by the database because Cascade Update Related Fields is checked for the foreign key in the `Orders` table, but in `ordersDT` the *old* `CustomerID` is set to J6COM. Therefore, the addition of `OrderID = @OldOrderID` in the `WHERE` clause prevents the row from being found. Instead, the `UPDATE` causes a `DBConcurrencyException` to be thrown.

## Second Case

Assume the following:

- Cascade Update Related Fields is unchecked.
- `UpdateRule` is set to `Cascade`.
- The `CommandText` property of the `Command` object in the `UpdateCommand` property of the `DataAdapter` is set as follows:

```
ordersUpdateCommand.CommandText =
    "UPDATE Orders " +
    "SET " +
    "  CustomerID = @NewCustomerID " +
    "WHERE OrderID = @OldOrderID";
```

If you change the `CustomerID` from J6COM to J7COM in `customersDT` and push the change to the database, then the `UPDATE` will throw a `SqlException`. This is because the child `Orders` table currently contains a row with the `CustomerID` of J6COM, and because of the foreign key you can't change the `CustomerID` in the parent `Customers` table. Even if you tried to change the `CustomerID` in `ordersDT` first and attempted to push the change to the database, you'd run into the same exception.

## Third Case

Assume the following:

- Cascade Update Related Fields is unchecked.

- `UpdateRule` is set to `None`.
- The `CommandText` of the `Command` object in the `UpdateCommand` of the `DataAdapter` is the same as in the second case.

The following code sets the `UpdateRule` of the `ChildKeyConstraint` to `None`:

```
myDataSet.Relations[ "CustomersOrders" ].ChildKeyConstraint.UpdateRule =
    Rule.None;
```

If you try to change the `CustomerID` from `J6COM` to `J7COM` in `customersDT`, then you'll throw an `InvalidOperationException`. This is because the child `ordersDT` `DataTable` currently contains a `DataRow` with the `CustomerID` of `J6COM`, and because of the foreign key you can't change the `CustomerID` in the parent `customersDT` `DataTable`. Even if you tried to change the `CustomerID` in `ordersDT` first, you'd run into the same exception.

## Conclusion

The enforcement of the constraints in the previous three examples is correct, and they show what a headache changing the primary key column values can be. The first case is the only one that works, and even then you have to resort to using "last one wins" concurrency in the `UPDATE` statement, which you should typically avoid.

What do you do if you want to change the primary key column value and apply the same change to the child rows? The easiest way is to simply delete the rows in the child table first, change the primary key value in the parent table, and recreate the rows in the child table with the new primary key value.

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Nested XML

As mentioned in [Chapter 10](#), "Using DataSet Objects to Store Data," a `DataSet` contains two methods that output the contents of the `DataRow` objects as XML:

- `GetXml()` returns the XML representation of the data stored in the `DataSet` object as a string.
- `WriteXml()` writes the data from the `DataSet` object out to an XML file.

A `DataRelation` contains a property named `Nested` that gets or sets a `bool` value that indicates whether the `DataRelation` objects are nested. This is useful when defining hierarchical relationships in XML.

Specifically, when you set `Nested` to `true`, the child rows are nested within the parent rows in any XML that you output using the `GetXml()` and `WriteXml()` methods. Similarly, you can read the nested rows when calling the `ReadXml()` method of a `DataSet` to read an XML file.

The following example sets a `DataRelation` object's `Nested` property to `true`:

```
myDataSet.Relations["CustomersOrders"].Nested = true;
```

This is shown in [Listing 12.1](#). Notice that this program writes two XML files named `nonNestedXmlFile.xml` and `nestedXmlFile.xml`. The `nonNestedXmlFile.xml` contains the default non-nested rows, and `nestedXmlFile.xml` contains the nested rows after the `DataRelation` object's `Nested` property is set to `true`.

### **Listing 12.1: NESTEDXML.CS**

---

```
/*
 NestedXml.cs illustrates how setting the Nested property
 of a DataRelation to true causes the the child rows to be nested within the
 parent rows in the output XML
 */

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

class NestedXml
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );

        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 2 CustomerID, CompanyName " +
            "FROM Customers " +
            "ORDER BY CustomerID;" +
            "SELECT OrderID, CustomerID, ShipCountry " +
            "FROM Orders " +
            "WHERE CustomerID IN (" +
            "    SELECT TOP 2 CustomerID " +
            "    FROM Customers " +
            "    ORDER BY CustomerID " +
            ") ";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
```

```

DataSet myDataSet = new DataSet();
mySqlConnection.Open();
int numberOfRows = mySqlDataAdapter.Fill(myDataSet);
Console.WriteLine("numberOfRows = " + numberOfRows);
mySqlConnection.Close();
DataTable customersDT = myDataSet.Tables["Table"];
DataTable ordersDT = myDataSet.Tables["Table1"];

// create a DataRelation object named customersOrdersDataRel
DataRelation customersOrdersDataRel =
    new DataRelation(
        "CustomersOrders",
        customersDT.Columns["CustomerID"],
        ordersDT.Columns["CustomerID"]
    );
myDataSet.Relations.Add(
    customersOrdersDataRel
);

// write the XML out to a file
Console.WriteLine("Writing XML out to file nonNestedXmlFile.xml");
myDataSet.WriteXml("nonNestedXmlFile.xml");

// set the DataRelation object's Nested property to true
// (causes child rows to be nested in the parent rows of the
// XML output)
myDataSet.Relations["CustomersOrders"].Nested = true;

// write the XML out again (this time the child rows are nested
// within the parent rows)
Console.WriteLine("Writing XML out to file nestedXmlFile.xml");
myDataSet.WriteXml("nestedXmlFile.xml");
}
}

```

---

[Listing 12.2](#) shows the `nonNestedXmlFile.xml` file produced by the program. Notice that the parent rows from the `Customers` table are listed first, followed by the child rows from the `Orders` table. The child rows are *not* nested within the parent rows.

### **Listing 12.2: NONNESTEDXMLFILE.XML**

---

```

<?xml version="1.0" standalone="yes"?>
<NewDataSet>
    <Table>
        <CustomerID>ALFKI</CustomerID>
        <CompanyName>Alfreds Futterkiste</CompanyName>
    </Table>
    <Table>
        <CustomerID>ANATR</CustomerID>
        <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
    </Table>
    <Table1>
        <OrderID>10308</OrderID>
        <CustomerID>ANATR</CustomerID>
        <ShipCountry>Mexico</ShipCountry>
    </Table1>
    <Table1>
        <OrderID>10625</OrderID>
        <CustomerID>ANATR</CustomerID>
        <ShipCountry>Mexico</ShipCountry>
    </Table1>

```

```

</Table1>
<Table1>
  <OrderID>10643</OrderID>
  <CustomerID>ALFKI</CustomerID>
  <ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
  <OrderID>10692</OrderID>
  <CustomerID>ALFKI</CustomerID>
  <ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
  <OrderID>10702</OrderID>
  <CustomerID>ALFKI</CustomerID>
  <ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
  <OrderID>10759</OrderID>
  <CustomerID>ANSTR</CustomerID>
  <ShipCountry>Mexico</ShipCountry>
</Table1>
<Table1>
  <OrderID>10835</OrderID>
  <CustomerID>ALFKI</CustomerID>
  <ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
  <OrderID>10926</OrderID>
  <CustomerID>ANSTR</CustomerID>
  <ShipCountry>Mexico</ShipCountry>
</Table1>
<Table1>
  <OrderID>10952</OrderID>
  <CustomerID>ALFKI</CustomerID>
  <ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
  <OrderID>11011</OrderID>
  <CustomerID>ALFKI</CustomerID>
  <ShipCountry>Germany</ShipCountry>
</Table1>
</NewDataSet>

```

---

[Listing 12.3](#) shows the `nestedXmlFile.xml` file produced by the program. Notice that this time the child rows from the `Orders` table are nested within the parent rows from the `Customers` table.

### **Listing 12.3: NESTEDXMLFILEL.CS**

---

```

<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Table>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkiste</CompanyName>
    <Table1>
      <OrderID>10643</OrderID>
      <CustomerID>ALFKI</CustomerID>
      <ShipCountry>Germany</ShipCountry>
    </Table1>
    <Table1>

```

```
<OrderID>10692</OrderID>
<CustomerID>ALFKI</CustomerID>
<ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
    <OrderID>10702</OrderID>
    <CustomerID>ALFKI</CustomerID>
    <ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
    <OrderID>10835</OrderID>
    <CustomerID>ALFKI</CustomerID>
    <ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
    <OrderID>10952</OrderID>
    <CustomerID>ALFKI</CustomerID>
    <ShipCountry>Germany</ShipCountry>
</Table1>
<Table1>
    <OrderID>11011</OrderID>
    <CustomerID>ALFKI</CustomerID>
    <ShipCountry>Germany</ShipCountry>
</Table1>
</Table>
<Table>
    <CustomerID>ANATR</CustomerID>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
    <Table1>
        <OrderID>10308</OrderID>
        <CustomerID>ANATR</CustomerID>
        <ShipCountry>Mexico</ShipCountry>
    </Table1>
    <Table1>
        <OrderID>10625</OrderID>
        <CustomerID>ANATR</CustomerID>
        <ShipCountry>Mexico</ShipCountry>
    </Table1>
    <Table1>
        <OrderID>10759</OrderID>
        <CustomerID>ANATR</CustomerID>
        <ShipCountry>Mexico</ShipCountry>
    </Table1>
    <Table1>
        <OrderID>10926</OrderID>
        <CustomerID>ANATR</CustomerID>
        <ShipCountry>Mexico</ShipCountry>
    </Table1>
</Table>
</NewDataSet>
```

## Team LiB

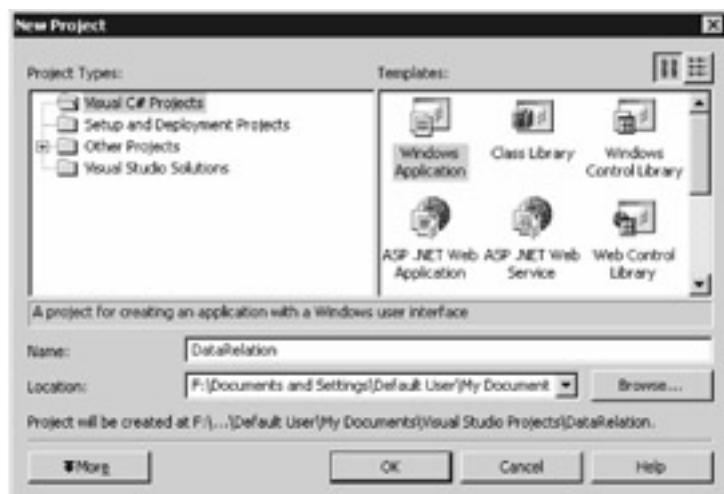
# Defining a Relationship Using Visual Studio .NET

In this section, you'll see how to create a Windows application in Visual Studio .NET (VS .NET) with a `DataSet` containing two `DataTable` objects. These `DataTable` objects will reference the `Customers` and `Orders` database tables. You'll then learn how to define a relationship between the two `DataTable` objects in the XML schema.

## Create the Windows Application

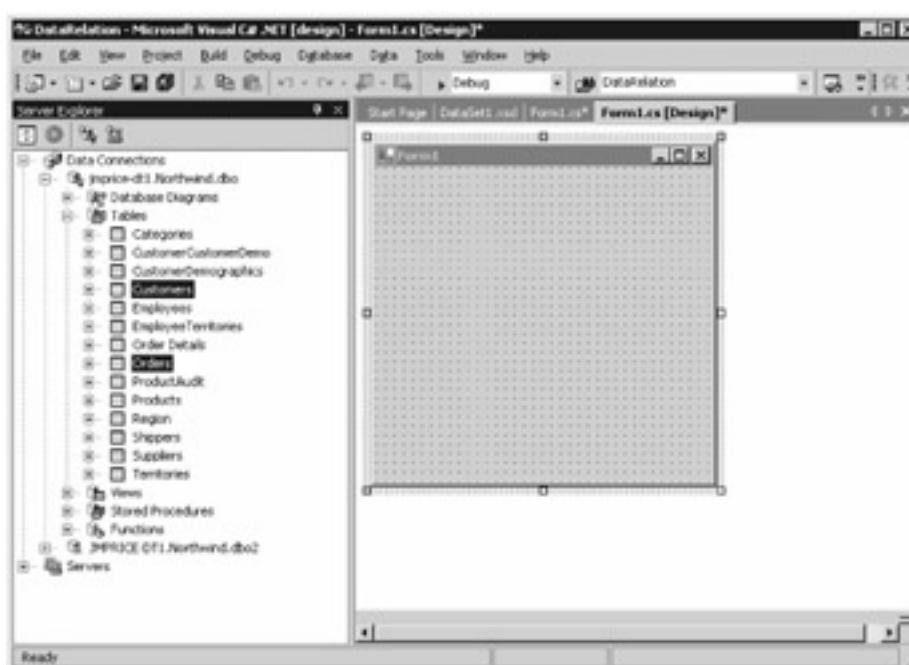
Perform the following steps to create the Windows application:

1. Open VS .NET and select File > New > Project and create a new Windows Application. Enter the name of the project as `DataRelation`, as shown in [Figure 12.2](#).



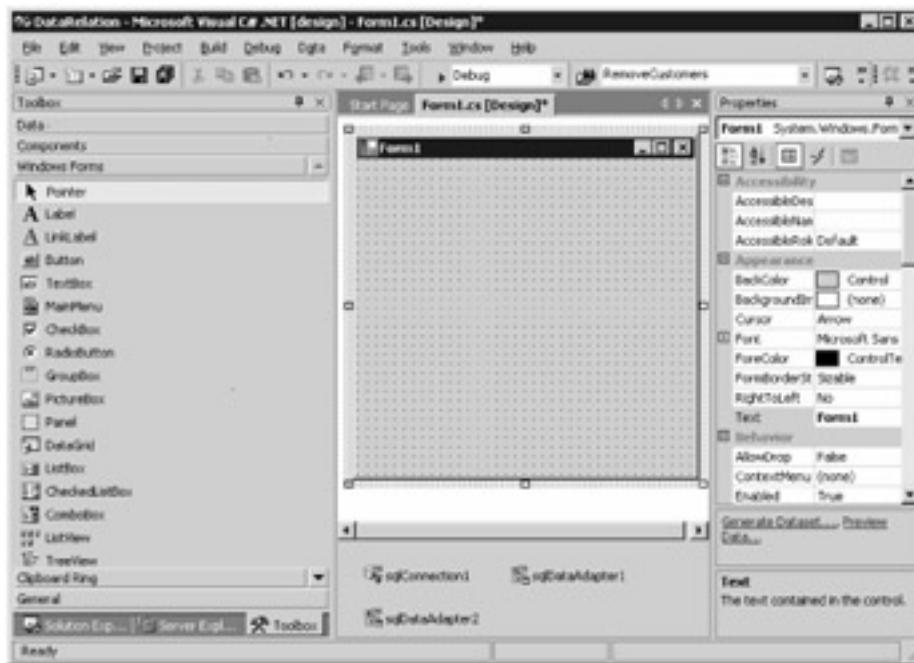
**Figure 12.2:** Creating the Windows application

2. Click the OK button to continue.
3. Open Server Explorer and connect to the Northwind database using the connection you used in the previous chapters. You can open Server Explorer by selecting View > Server Explorer. Expand the Tables node in the tree and select both the `Customers` and `Orders` tables by Ctrl +left-clicking each table, as shown in [Figure 12.3](#).



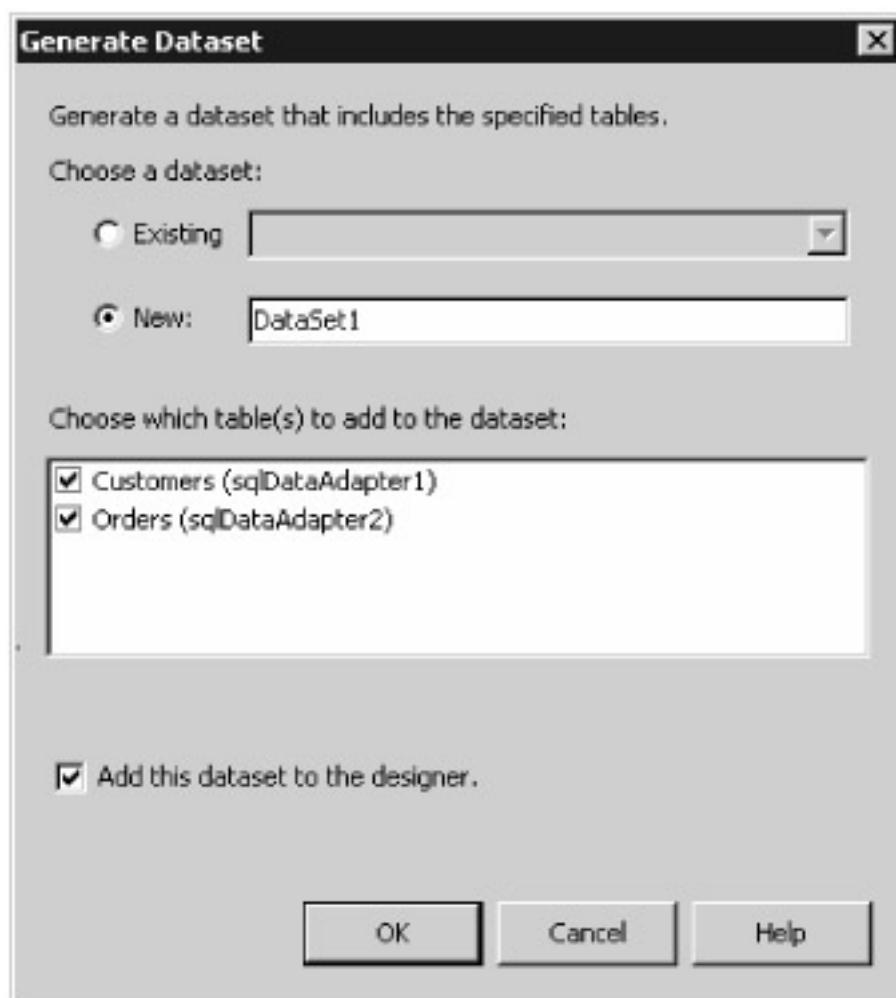
**Figure 12.3:** Selecting both the Customers and Orders tables from Server Explorer

4. Drag the Customers and Orders tables to your form. VS .NET then creates three objects in the tray beneath your form. These objects are named `sqlConnection1` (used to access the Northwind database), `sqlDataAdapter1` (used to handle access to the Customers table), and `sqlDataAdapter2` (used to handle access to the Orders table). These objects are shown in [Figure 12.4](#).

**Figure 12.4:** The new objects in the tray beneath the form

5. Next, you need a `DataSet` object that contains `DataTable` objects to store the rows from the Customers and Orders tables. To create a `DataSet` object, click your blank form and then click the Generate Dataset link at the bottom of the Properties window for the form shown earlier in [Figure 12.4](#).

This displays the Generate Dataset dialog box, as shown in [Figure 12.5](#). Leave all the settings in this dialog in their default state.



**Figure 12.5:** The Generate Dataset dialog box

Notice that a new `DataSet` is to be created, and the `Customers` and `Orders` tables are used in the new `DataSet`. The new `DataSet` will also be added to the designer, as indicated by the check box in the dialog box. Click the OK button to continue.

A new `DataSet` named `dataSet11` will be added to the tray beneath your form.

## Adding a Relation to the XML Schema of the *DataSet*

Perform the following steps to add a relation to the XML schema of your `DataSet`:

1. Select your new `DataSet` in the tray and click the View Schema link in the Properties window. This displays the Schema Editor, as shown in [Figure 12.6](#).

**Figure 12.6:** The Schema Editor

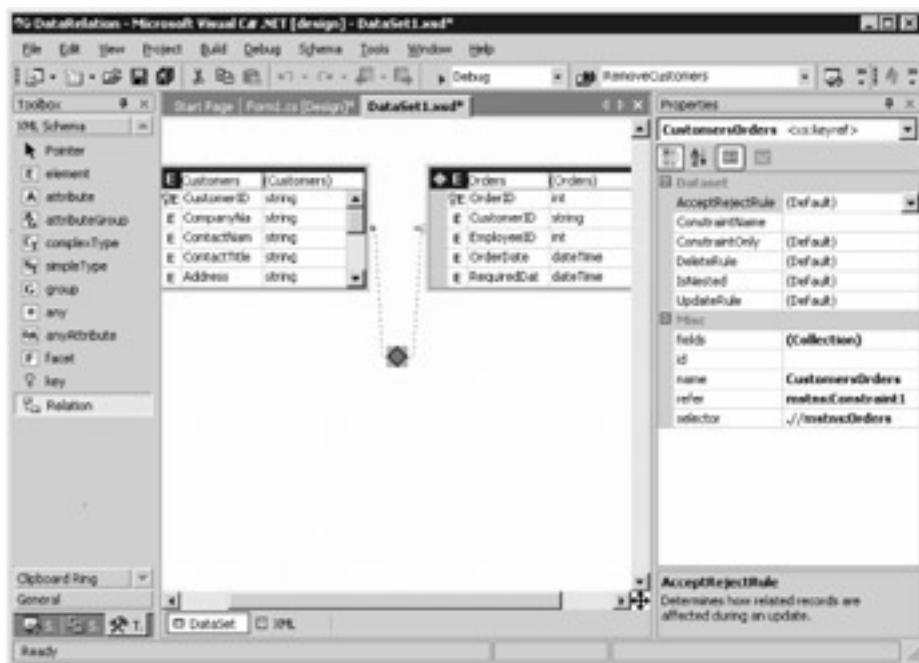
2. Drag a relation from the XML Schema tab in the Toolbox to the `Orders` table entity. This opens the Edit Relation dialog box, as shown in [Figure 12.7](#).

**Figure 12.7:** The Edit Relation dialog box

As you can see from [Figure 12.7](#), you can set the parent and child tables, the primary key and foreign keys, along with other details for the relation.

3. Leave the settings in the Edit Relation dialog box in their default state and click the OK button to continue.

This adds the new relation to the `Customers` and `Orders` entities in the XML schema. You can click the diamond between the `Customers` and `Orders` entities to view the properties of the relation, as shown in [Figure 12.8](#).



**Figure 12.8:** The properties of the new relation

**Note** You can add a relation to a strongly typed *DataSet* in the same way as described in this section.

**Team LiB**

◀ PREVIOUS ▶ NEXT ▶

**Team LiB****◀ PREVIOUS****NEXT ▶**

## Summary

In this chapter, you delved into the details of `UniqueConstraint` and `ForeignKeyConstraint` objects. You also learned how to define a relationship between `DataTable` objects using a `DataRelation` object, which by default automatically creates a `UniqueConstraint` and a `ForeignKeyConstraint` object for you. The `UniqueConstraint` object is added to your parent `DataTable` and the `ForeignKeyConstraint` is added to your child `DataTable`.

You also saw how to navigate rows in related `DataTable` objects, make changes in related `DataTable` objects, and finally push those changes to the database. You also saw how to use VS .NET to define a relationship.

In the [next chapter](#), you'll learn how to use `DataView` objects.

**Team LiB****◀ PREVIOUS****NEXT ▶**

## Chapter 13: Using *DataView* Objects

### Overview

In [chapter 11](#), "Using DataSet Objects to Modify Data," you saw that you can filter and sort the `DataRow` objects in a `DataTable` using the `Select( )` method. In this chapter, you'll learn how to use `DataView` objects to also filter and sort rows. The advantage of a `DataView` is that you can bind it to a visual component such as a `DataGridView` control, and you'll also see how to do that in this chapter.

A `DataView` stores copies of the rows in a `DataTable` as `DataRowView` objects. The `DataRowView` objects provide access to the underlying `DataRow` objects in a `DataTable`. Therefore, when you examine and edit the contents of a `DataRowView`, you are actually working with the underlying `DataRow`. Keep that in mind when reading this chapter.

Featured in this chapter:

- The `DataView` class
- Creating and using a `DataView` object
- Using the default sort algorithm
- Performing advanced filtering
- [The `DataRowView` class](#)
- Finding `DataRowView` objects in a `DataView`
- [Adding, modifying, and removing `DataRowView` objects from a `DataView`](#)
- Creating child `DataView` objects
- The `DataViewManager` class
- [Creating and using a `DataViewManager` object](#)
- Creating a `DataView` using Visual Studio .NET

## The *DataView* Class

You use an object of the *DataView* class to view only specific rows in a *DataTable* object using a filter. You can also sort the rows viewed by a *DataView*. You can add, modify, and remove rows from a *DataView*, and those changes will also be applied to the underlying *DataTable* that the *DataView* reads from; you'll learn more about this later in the section "[Adding, Modifying, and Removing DataRowView Objects from a DataView](#)." [Table 13.1](#) shows some of the *DataView* properties, and [Table 13.2](#) shows some of the *DataView* methods.

**Table 13.1: *DataView* PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
AllowDelete	bool	Gets or sets a bool that indicates whether deletion of <i>DataRowView</i> objects from your <i>DataView</i> is permitted. The default is <code>true</code> .
AllowEdit	bool	Gets or sets a bool that indicates whether editing of <i>DataRowView</i> objects in your <i>DataView</i> is permitted. The default is <code>true</code> .
AllowNew	bool	Gets or sets a bool that indicates whether adding new <i>DataRowView</i> objects to your <i>DataView</i> is permitted. The default is <code>true</code> .
ApplyDefaultSort	bool	Gets or sets a bool that indicates whether to use the default sorting algorithm to sort rows in your <i>DataView</i> . When set to <code>true</code> , the default sort is used and is set to the ascending order of the <i>PrimaryKey</i> property of the underlying <i>DataTable</i> (if the <i>PrimaryKey</i> is set). The default is <code>false</code> .
Count	int	Gets the number of rows visible to your <i>DataView</i> .
DataViewManager	<i>DataViewManager</i>	Gets the <i>DataViewManager</i> associated with your <i>DataView</i> . You'll learn about <i>DataViewManager</i> objects later in the section " <a href="#">Creating and Using a <i>DataViewManager</i> Object</a> ."
RowFilter	string	Gets or sets the expression used to filter rows in your <i>DataView</i> .
RowStateFilter	<i>DataViewRowState</i>	Gets or sets the expression used to filter rows based on constants from the <i>DataViewRowState</i> enumeration. Values are shown in <a href="#">Table 13.3</a> .
Sort	string	Gets or sets an expression that specifies the columns to sort by and optional sort order for the rows in your <i>DataView</i> . This string expression contains the column name followed by <code>ASC</code> (for ascending sort) or <code>DESC</code> (for descending sort). A column is sorted in ascending order by default. You separate multiple columns by commas in the string. For example: <code>CustomerID ASC, CompanyName DESC</code> .
Table	<i>DataTable</i>	Gets or sets the underlying <i>DataTable</i> that your <i>DataView</i> is associated with.

**Table 13.2: *DataView* METHODS**

METHOD	RETURN TYPE	DESCRIPTION
AddNew( )	DataViewRow	Adds a new DataRowView to your DataView, and therefore adds a new DataRow to the underlying DataTable.
BeginInit( )	void	Begins the runtime initialization of your DataView in a form or component.
CopyTo( )	void	Copies the rows from your DataView into an array. This method is for Web Forms Interfaces only.
Delete()	void	Deletes the DataRowView at the specified index from your DataView. The deletion of the underlying DataRow isn't permanent until you call the AcceptChanges( ) method of your DataTable. You can undo the delete by calling the RejectChanges( ) method of your DataTable.
EndInit( )	void	Ends the runtime initialization of your DataView in a form or component.
Find()	int	Overloaded. Finds and returns the index of the DataRowView with the specified primary key in your DataView. The int returned by this method is the index of the DataRowView if found; otherwise -1 is returned. You must first set the Sort property of your DataView to sort on the primary key. For example, if you want to find a DataRowView based on the CustomerID, you must set Sort to CustomerID, CustomerID ASC, or CustomerID DESC.
FindRows( )	DataRowView [ ]	Overloaded. Finds and returns an array of DataRowView objects that have columns matching the specified primary key. As with the Find( ) method, you must set the Sort property of your DataView to sort on the primary key before calling the FindRows( ) method.
GetEnumerator( )	IEnumerator	Returns an enumerator for your DataView.
ToString()	string	Returns a string that represents your DataView.

**Table 13.3: DataViewRowState ENUMERATION MEMBERS**

CONSTANT	DESCRIPTION
Added	A new row.
CurrentRows	The current rows, which include Unchanged, Added, and ModifiedCurrent rows.
Deleted	A deleted row.
ModifiedCurrent	A current row that has been modified.
ModifiedOriginal	The original row before it was modified.
None	Doesn't match any of the rows in the DataTable.
OriginalRows	The original rows, which include Unchanged and Deleted rows.
Unchanged	A row that hasn't been changed.

One of the DataView events is `ListChanged`. It fires when the list managed by your DataView changes. Its event handler is `ListChangedEventHandler`.

[Table 13.3](#) shows the members of the `System.Data.DataViewState` enumeration. This enumeration is used with the `RowState` property of a `DataTable`; this property is used to specify that the rows viewed by the `DataView` are filtered by their `DataViewState`.

**Team LiB**

[PREVIOUS](#)

[NEXT](#)

## Creating and Using a *DataView* Object

In this section, you'll learn how to filter and sort rows with a *DataView* object. You create a *DataView* object using one of the following constructors:

```
DataTable()
DataView(DataTable myDataTable)
DataView(DataTable myDataTable, string filterExpression, string sortExpression,
DataViewRowState rowState)
```

where

- **myDataTable** specifies the *DataTable* that your *DataView* is associated with. Your *DataView* will read the rows from this *DataTable*. The *Table* property of your *DataView* is set to *myDataTable*.
- **filterExpression** specifies a string containing the expression you want to filter the rows by. The *RowFilter* property of your *DataView* is set to *filterExpression*.
- **sortExpression** specifies a string containing the expression you want to sort the rows by. The *Sort* property of your *DataView* is set to *sortExpression*.
- **rowState** specifies an additional filter to apply to the rows; *rowState* filters by the *DataRowView-State* of the *DataViewRow* objects in your *DataView*. The *RowStateFilter* of your *DataView* is set to *rowState*.

Before you create a *DataView*, you first need a *DataTable* from which to read rows. The following example creates and populates a *DataTable* named *customersDT* that contains rows from the *Customers* table:

```
SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
mySqlCommand.CommandText =
    "SELECT CustomerID, CompanyName, Country " +
    "FROM Customers";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet, "Customers");
mySqlConnection.Close();
DataTable customersDT = myDataSet.Tables["Customers"];
```

Let's say you want to filter the rows in *CustomersDT* to view just the customers in the UK. Your filter string expression would be

```
string filterExpression = "Country = 'UK'";
```

**Note** Notice that *UK* is placed within single quotes. This is because *UK* is a string literal.

Also, let's say you want to sort those rows by ascending *CustomerID* column values and descending *CompanyName* column values. Therefore, your sort expression would be

```
string sortExpression = "CustomerID ASC, CompanyName DESC";
```

**Note** *ASC* sorts in ascending order. *DESC* sorts in descending order.

Finally, let's say you wanted to view only the original rows in the *DataView*; you therefore set your row state filter to *DataViewRowState.OriginalRows*:

```
DataViewRowState rowStateFilter = DataViewRowState.OriginalRows;
```

**Note** The default is `DataViewState.CurrentRows`, which includes rows in your `DataView` for which the `DataViewState` is `Unchanged`, `Added`, and `ModifiedCurrent`.

The following example creates a `DataView` object named `customersDV` and passes `customersDT`, `filterExpression`, `sortExpression`, and `rowStateFilter` to the `DataView` constructor:

```
Code: DataView customersDV =
    new DataView(
        customersDT, filterExpression, sortExpression, rowStateFilter
    );
```

You can also create a `DataView` and set the `Table`, `RowFilter`, `Sort`, and `RowStateFilter` properties individually. For example:

```
Code: DataView customersDV = new DataView();
customersDV.Table = customersDT;
customersDV.RowFilter = filterExpression;
customersDV.Sort = sortExpression;
customersDV.RowStateFilter = rowStateFilter;
```

A `DataView` stores rows as `DataRowView` objects, and the rows are read from the `DataRow` objects stored in the underlying `DataTable`. The following example uses a `foreach` loop to display the `DataRowView` objects in the `customersDV` `DataView`:

```
Code: foreach (DataRowView myDataRowView in customersDV)
{
    for (int count = 0; count < customersDV.Table.Columns.Count; count++)
    {
        Console.WriteLine(myDataRowView[count]);
    }
    Console.WriteLine("");
}
```

Note that `myDataRowView[count]` returns the value of the column at the numeric position specified by `count`. For example, `myDataRowView[0]` returns the value of the `CustomerID` column. You'll learn more about the `DataRowView` class later in the section "[The `DataRowView` Class](#)".

[Listing 13.1](#) shows a program that uses the previous code examples.

### **Listing 13.1: USINGDATAVIEW.CS**

---

```
Code: /*
Using DataView.cs illustrates the use of a DataView object to
filter and sort rows
*/
using System;
using System.Data;
using System.Data.SqlClient;

class UsingDataView
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
```

## Creating and Using a DataView Object

```
"SELECT CustomerID, CompanyName, Country " +
"FROM Customers";
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet, "Customers");
mySqlConnection.Close();
DataTable customersDT = myDataSet.Tables["Customers"];
// set up the filter and sort expressions
string filterExpression = "Country = 'UK'";
string sortExpression = "CustomerID ASC, CompanyName DESC";
DataViewRowState rowStateFilter = DataViewRowState.OriginalRows;

// create a DataView object named customersDV
DataView customersDV = new DataView();
customersDV.Table = customersDT;
customersDV.RowFilter = filterExpression;
customersDV.Sort = sortExpression;
customersDV.RowStateFilter = rowStateFilter;

// display the rows in the customersDV DataView object
foreach (DataRowView myDataRowView in customersDV)
{
    for (int count = 0; count < customersDV.Table.Columns.Count; count++)
    {
        Console.WriteLine(myDataRowView[count]);
    }
    Console.WriteLine(" ");
}
}
```

---

Notice that the rows in `customersDV` are filtered to those for which the `Country` is `UK`, and the resulting rows are then sorted by `CustomerID`. The output from this program is as follows:

AROUT  
Around the Horn  
UK

BSBEV  
B's Beverages  
UK

CONSH  
Consolidated Holdings  
UK

EASTC  
Eastern Connection  
UK

ISLAT  
Island Trading  
UK  
NORTS  
North/South  
UK

SEVES  
Seven Seas Imports  
UK

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## Using the Default Sort Algorithm

If you want to sort the `DataRowView` objects in your `DataView` based on the primary key of your `DataTable`, you can use a shortcut. Instead of setting the `Sort` property of your `DataView`, you set the `PrimaryKey` property of your `DataTable` and then set the `ApplyDefaultSort` property of your `DataView` to `true`.

The `Sort` property of your `DataView` is then automatically set to the primary key of your `DataTable`. This causes the `DataRowView` objects in your `DataView` to be sorted in ascending order based on the primary key column values.

Let's take a look at an example. The following code sets the `PrimaryKey` property of the `customersDT` `DataTable` to the `CustomerID`  `DataColumn`:

```
customersDT.PrimaryKey =  
    new DataColumn[]  
    {  
        customersDT.Columns["CustomerID"]  
    };
```

The next example sets the `ApplyDefaultSort` property of `customersDV` to `true`:

```
customersDV.ApplyDefaultSort = true;
```

The `Sort` property of `customersDV` is then set to `CustomerID`, which causes the `DataRowView` objects to be sorted by the ascending `CustomerID` values.

**Note** You'll find the code examples in this section in the `UsingDefaultSort.cs` program. The listing is omitted from this book for brevity.

## Performing Advanced Filtering

The `RowFilter` property of a `DataGridView` is similar to a `WHERE` clause in a `SELECT` statement. You can therefore use very powerful filter expressions in your `DataGridView`. For example, you can use `AND`, `OR`, `NOT`, `IN`, `LIKE`, comparison operators, arithmetic operators, wildcard characters (`*` and `%`), and aggregate functions.

**Note** For full details on how to use such filter expressions in your `DataGridView` objects, refer to the `DataColumn` *Expression* property in the .NET online documentation.

Here's a simple example that uses the `LIKE` operator and the percent (%) wildcard character to filter rows with a `CustomerName` that starts with `Fr`:

```
string filterExpression = "CompanyName LIKE 'Fr%'";  
customersDV.RowFilter = filterExpression;
```

Notice that the string `Fr%` is placed in single quotes—which you must do for all string literals. When this code replaces the existing code in the `UsingDataGridView.cs` program shown earlier in [Listing 13.1](#), the output is as follows:

FRANK  
Frankenversand  
Germany

FRANR  
France restauration  
France

FRANS  
Franchi S.p.A.  
Italy

**Note** I've made this change in the `UsingDataGridView2.cs` program (the listing is omitted from this book for brevity). Feel free to examine and run this program.

## The *DataRowView* Class

Rows in a *DataView* object are stored as objects of the *DataRowView* class. A *DataRowView* object provides access to the underlying *DataRow* object in a *DataTable*. When you examine and edit the contents of a *DataRowView*, you are actually working with the underlying *DataRow*. Keep this in mind when working with *DataRowView* objects. [Table 13.4](#) shows some of the *DataRowView* properties, and [Table 13.5](#) shows some of the *DataRowView* methods.

**Table 13.4: *DataRowView* PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
DataGridView	DataGridView	Gets the <i>DataView</i> that the <i>DataRowView</i> belongs to.
IsEdit	bool	Gets a <i>bool</i> that indicates whether the <i>DataRowView</i> (and therefore the underlying <i>DataRow</i> ) is in edit mode.
IsNew	bool	Gets a <i>bool</i> that indicates whether the <i>DataRowView</i> has just been added.
Row	DataRow	Gets the underlying <i>DataRow</i> that is being viewed from the <i>DataTable</i> .
RowVersion	DataRowVersion	Gets the <i>DataRowVersion</i> of the underlying <i>DataRow</i> . Members of the <i>System.Data.DataRowVersion</i> enumeration are <ul style="list-style-type: none"> <li>• <i>Current</i>, which indicates the <i>DataRow</i> contains the current values.</li> <li>• <i>Default</i>, which indicates the <i>DataRow</i> contains the default values.</li> <li>• <i>Original</i>, which indicates the <i>DataRow</i> contains the original values.</li> <li>• <i>Proposed</i>, which indicates the <i>DataRow</i> contains proposed values.</li> </ul>

**Table 13.5: *DataRowView* METHODS**

METHOD	RETURN TYPE	DESCRIPTION
BeginEdit()	void	Begins editing of the <i>DataRowView</i> in your <i>DataView</i> , and therefore begins the editing of the underlying <i>DataRow</i> in your <i>DataTable</i> . You then edit this <i>DataRow</i> through the <i>DataRowView</i> .
CancelEdit()	void	Cancels editing of the <i>DataRowView</i> in your <i>DataView</i> , and therefore cancels editing of the underlying <i>DataRow</i> .
CreateChildView()	DataGridView	Overloaded. Returns a <i>DataView</i> for a child <i>DataTable</i> , if present.

Delete()	void	Deletes the <code>DataRowView</code> in your <code>DataView</code> . The deletion of the underlying <code>DataRow</code> isn't committed in the <code>DataTable</code> until you call the <code>AcceptChanges()</code> method of your <code>DataTable</code> . You can undo the deletion by calling the <code>RejectChanges()</code> method of your <code>DataTable</code> , which also undoes any uncommitted additions or modifications.
EndEdit()	void	Ends the editing of a <code>DataRowView</code> .

**Team LiB****◀ PREVIOUS****NEXT ▶**

## Finding *DataRowView* Objects in a *DataView*

You can find the index of a *DataRowView* in a *DataView* using the `Find()` method of a *DataView*. You can also get an array of *DataRowView* objects using the `FindRows()` method of a *DataView*. You'll learn how to use the `Find()` and `FindRows()` methods in this section.

### Finding the Index of a *DataRowView* Using the *Find()* Method

The `Find()` method returns the index of the *DataRowView* with the specified primary key in your *DataView*. The `int` returned by this method is the index of the *DataRowView* if found; otherwise -1 is returned.

To find the correct index, you must first set the `Sort` property of your *DataView* to sort on the primary key. For example, if you want to find a *DataRowView* based on the `CustomerID`, you must set the `Sort` property of your *DataView* to `CustomerID, CustomerID ASC`, or `CustomerID DESC`:

```
string sortExpression = "CustomerID";
customersDV.Sort = sortExpression;
```

Assume that the sorted *DataRowView* objects in `customersDV` are as follows:

AROUT

Around the Horn

UK

BSBEV

B's Beverages

UK

CONSH

Consolidated Holdings

UK

EASTC

Eastern Connection

UK

ISLAT

Island Trading

UK

NORTS

North/South

UK

SEVES

Seven Seas Imports

UK

The following example calls the `Find()` method to find the index of the *DataRowView* in `customersDV` with a `CustomerID` of `BSBEV`:

```
int index = customersDV.Find("BSBEV");
```

Because `BSBEV` occurs at index 1, the `Find()` method returns 1.

**Note** *DataRowView* objects in a *DataView* start at index 0. Therefore, `BSBEV` occurs at index 1.

### Finding *DataRowView* Objects Using the *FindRows()* Method

The `FindRows()` method of a `DataView` finds and returns an array of `DataRowView` objects for which the primary key column matches the primary key in your `DataView`. If no rows were found, then the returned array will have zero elements, and the `Length` property of the array will be 0.

To find `DataRowView` objects using the `FindRows()` method, you must first set the `Sort` property of your `DataView` to sort on the primary key. For example, if you want to find `DataRowView` objects based on the `CustomerID`, you must set the `Sort` property of your `DataView` to `CustomerID`, `CustomerID ASC`, or `CustomerID DESC`:

```
string sortExpression = "CustomerID";
customersDV.Sort = sortExpression;
```

The following example calls the `FindRows()` method to find the `DataRowView` that has the `CustomerID` of BSBEV:

```
DataRowView[ ] customersDRVs = customersDV.FindRows( "BSBEV" );
```

Since there is only one match, the `customersDRVs` array will contain one `DataRowView`.

[Listing 13.2](#) shows a program that uses the `Find()` and `FindRows()` methods.

## **Listing 13.2: FINDINGDATAROWVIEWS.CS**

---

```
/*
 FindingDataRowViews.cs illustrates the use of the Find() and
 FindRows() methods of a DataView to find DataRowView objects
 */

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

class FindingDataRowViews
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT CustomerID, CompanyName, Country " +
            "FROM Customers";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        mySqlDataAdapter.Fill(myDataSet, "Customers");
        mySqlConnection.Close();
        DataTable customersDT = myDataSet.Tables["Customers"];

        // set up the filter and sort expressions
        string filterExpression = "Country = 'UK'";
        string sortExpression = "CustomerID";
        DataViewRowState rowStateFilter = DataViewRowState.OriginalRows;

        // create a DataView object named customersDV
        DataView customersDV = new DataView();
        customersDV.Table = customersDT;
        customersDV.RowFilter = filterExpression;
        customersDV.Sort = sortExpression;
        customersDV.RowStateFilter = rowStateFilter;
```

```

// display the rows in the customersDV DataView object
foreach (DataRowView myDataRowView in customersDV)
{
    for (int count = 0; count < customersDV.Table.Columns.Count; count++)
    {
        Console.WriteLine(myDataRowView[count]);
    }
    Console.WriteLine(" ");
}

// use the Find() method of customersDV to find the index of
// the DataRowView whose CustomerID is BSBEV
int index = customersDV.Find("BSBEV");
Console.WriteLine("BSBEV found at index " + index + "\n");

// use the FindRows() method of customersDV to find the DataRowView
// whose CustomerID is BSBEV
DataRowView[] customersDRVs = customersDV.FindRows("BSBEV");
foreach (DataRowView myDataRowView in customersDRVs)
{
    for (int count = 0; count < customersDV.Table.Columns.Count; count++)
    {
        Console.WriteLine(myDataRowView[count]);
    }
    Console.WriteLine(" ");
}
}
}

```

---

**Tip** If you are using an early version of the .NET SDK, you might encounter the following compilation error when compiling this program:

*FindingDataRowViews.cs(59,35): error CS0117: 'System.Data.DataView' does not contain a definition for 'FindRows'*

If you get this error, compile the program with Visual Studio .NET.

The output from this program is as follows:

AROUT  
Around the Horn  
UK

BSBEV  
B's Beverages  
UK

CONSH  
Consolidated Holdings  
UK

EASTC  
Eastern Connection  
UK

ISLAT  
Island Trading  
UK

NORTS

North/South

UK

SEVES

Seven Seas Imports

UK

BSBEV found at index 1

BSBEV

B's Beverages

UK

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Adding, Modifying, and Removing *DataRowView* Objects from a *DataView*

It's important to understand that *DataRowView* objects in a *DataView* provide access to the underlying *DataRow* objects in a *DataTable*. Therefore, when you examine and edit the contents of a *DataRowView*, you're actually working with the underlying *DataRow*. Similarly, when you remove a *DataRowView*, you are removing the underlying *DataRow*.

### Adding a *DataRowView* to a *DataView*

To add a new *DataRowView* to a *DataView*, you call the *AddNew()* method of your *DataView*. The *AddNew()* method returns a *DataRowView* object that you use to set the column values for the new row. The following example calls the *AddNew()* method of the *customersDV* *DataView*:

```
DataRowView customerDRV = customersDV.AddNew();
customerDRV[ "CustomerID" ] = "J7COM";
customerDRV[ "CompanyName" ] = "J7 Company";
customerDRV[ "Country" ] = "UK";
customerDRV.EndEdit();
```

Notice the use of the *EndEdit()* method of the *customerDRV* *DataRowView* to end the editing. The *EndEdit()* method creates a new *DataRow* in the underlying *DataTable*. The  *DataColumn* objects in the new *DataRow* will contain the column values specified in the previous code.

**Note** You can undo the addition by calling the *CancelEdit()* method of a *DataRowView*.

You can get the underlying *DataRow* added to the *DataTable* using the *Row* property of a *DataRowView*. For example:

```
DataRow customerDR = customerDRV.Row;
```

### Modifying an Existing *DataRowView*

To begin modifying an existing *DataRowView* in a *DataView*, you call the *BeginEdit()* method of the *DataRowView* in your *DataView*. The following example calls the *BeginEdit()* method for the first *DataRowView* in *customersDV*:

```
customersDV[ 0 ].BeginEdit();
```

**Note** Remember that *DataRowView* objects in a *DataView* start at index 0, and therefore *customersDV[ 0 ]* is the first *DataRowView* in *customersDV*.

You can then modify a  *DataColumn* in the underlying *DataRow* through the *DataRowView*. The following example sets the *CompanyName*  *DataColumn* to Widgets Inc.:

```
customersDV[ 0 ][ "CompanyName" ] = "Widgets Inc. ";
```

Once you've finished making your modifications, you call the *EndEdit()* method to make your modifications permanent in the underlying *DataTable*. For example:

```
customersDV[ 0 ].EndEdit();
```

**Note** You can undo the modification by calling the *CancelEdit()* method of a *DataRowView*.

### Removing an Existing *DataRowView*

To remove an existing *DataRowView* from a *DataView*, you can call the *Delete()* method of either the *DataView* or the *DataRowView*. When calling the *Delete()* method of a *DataView*, you pass the index of the *DataRowView* you want to remove. The following example removes the second *DataRowView* from *customersDV*:

```
customersDV.Delete(1);
```

When calling the `Delete()` method of a `DataRowView`, you simply call that method of the `DataRowView` in your `DataView`. The following example removes the third `DataRowView` from `customersDV`:

```
customersDV[2].Delete();
```

With either of these `Delete()` methods, the deletion isn't committed in the underlying `DataTable` until you call the `AcceptChanges()` method of your `DataTable`. For example:

```
customersDT.AcceptChanges();
```

**Note** You can call the `RejectChanges()` method of a `DataTable` to undo the deletions. This method will also undo any uncommitted additions and modifications of rows.

[Listing 13.3](#) shows a program that adds, modifies, and removes `DataRowView` objects from a `DataView`. This program also displays the `IsNew` and `IsEdit` properties of the `DataRowView` objects, which indicate whether the `DataRowView` is new and is being edited.

### **Listing 13.3: ADDMODIFYANDREMOVEDATAROWVIEWS.CS**

---

```
/*
AddModifyAndRemoveDataRowViews.cs illustrates how to
add, modify, and remove DataRowView objects from a DataView
*/

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

class AddModifyAndRemoveDataRowViews
{
    public static void DisplayDataRow(
        DataRow myDataRow,
        DataTable myDataTable
    )
    {
        Console.WriteLine("\nIn DisplayDataRow()");
        foreach (DataColumn my DataColumn in myDataTable.Columns)
        {
            Console.WriteLine(my DataColumn + " = " +
                myDataRow[my DataColumn]);
        }
    }

    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT CustomerID, CompanyName, Country " +
            "FROM Customers";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        mySqlDataAdapter.Fill(myDataSet, "Customers");
    }
}
```

```
mySqlConnection.Close();
DataTable customersDT = myDataSet.Tables["Customers"];

// set up the filter expression
string filterExpression = "Country = 'UK'";

// create a DataView object named customersDV
DataView customersDV = new DataView();
customersDV.Table = customersDT;
customersDV.RowFilter = filterExpression;

// add a new DataRowView (adds a DataRow to the DataTable)
Console.WriteLine("\nCalling customersDV.AddNew()");
DataRowView customerDRV = customersDV.AddNew();
customerDRV["CustomerID"] = "J7COM";
customerDRV["CompanyName"] = "J7 Company";
customerDRV["Country"] = "UK";
Console.WriteLine("customerDRV[\"CustomerID\"] = " +
    customerDRV["CustomerID"]);
Console.WriteLine("customerDRV[\"CompanyName\"] = " +
    customerDRV["CompanyName"]);
Console.WriteLine("customerDRV[\"Country\"] = " +
    customerDRV["Country"]);
Console.WriteLine("customerDRV.IsNew = " + customerDRV.IsNew);
Console.WriteLine("customerDRV.IsEdit = " + customerDRV.IsEdit);
customerDRV.EndEdit();

// get and display the underlying DataRow
DataRow customerDR = customerDRV.Row;
DisplayDataRow(customerDR, customersDT);

// modify the CompanyName of customerDRV
Console.WriteLine("\nSetting customersDV[0][\"CompanyName\"] to Widgets Inc.");
customersDV[0].BeginEdit();
customersDV[0]["CompanyName"] = "Widgets Inc.";
Console.WriteLine("customersDV[0][\"CustomerID\"] = " +
    customersDV[0]["CustomerID"]);
Console.WriteLine("customersDV[0][\"CompanyName\"] = " +
    customersDV[0]["CompanyName"]);
Console.WriteLine("customersDV[0].IsNew = " + customersDV[0].IsNew);
Console.WriteLine("customersDV[0].IsEdit = " + customersDV[0].IsEdit);
customersDV[0].EndEdit();

// display the underlying DataRow
DisplayDataRow(customersDV[0].Row, customersDT);

// remove the second DataRowView from customersDV
Console.WriteLine("\ncustomersDV[1][\"CustomerID\"] = " +
    customersDV[1]["CustomerID"]);
Console.WriteLine("\nCalling customersDV.Delete(1)");
customersDV.Delete(1);
Console.WriteLine("customersDV[1].IsNew = " + customersDV[1].IsNew);
Console.WriteLine("customersDV[1].IsEdit = " + customersDV[1].IsEdit);

// remove the third DataRowView from customersDV
Console.WriteLine("\ncustomersDV[2][\"CustomerID\"] = " +
    customersDV[2]["CustomerID"]);
Console.WriteLine("\nCalling customersDV[2].Delete()");
customersDV[2].Delete();

// call the AcceptChanges() method of customersDT to
```

```

// make the deletes permanent in customersDT
customersDT.AcceptChanges();

// display the rows in the customersDV DataView object
Console.WriteLine("\nDataRowView objects in customersDV:\n");
foreach (DataRowView myDataRowView in customersDV)
{
    for (int count = 0; count < customersDV.Table.Columns.Count; count++)
    {
        Console.WriteLine(myDataRowView[count]);
    }
    Console.WriteLine(" ");
}
}

```

---

The output from this program is as follows:

```

Calling customersDV.AddNew()
customerDRV[ "CustomerID" ] = J7COM
customerDRV[ "CompanyName" ] = J7 Company
customerDRV[ "Country" ] = UK
customerDRV.IsNew = True
customerDRV.IsEdit = True

```

```

In DisplayDataRow()
CustomerID = J7COM
CompanyName = J7 Company
Country = UK

```

```

Setting customersDV[0][ "CompanyName" ] to Widgets Inc.
customersDV[0][ "CustomerID" ] = AROUT
customersDV[0][ "CompanyName" ] = Widgets Inc.
customersDV[0].IsNew = False
customersDV[0].IsEdit = True

```

```

In DisplayDataRow()
CustomerID = AROUT
CompanyName = Widgets Inc.
Country = UK

```

```
customersDV[1][ "CustomerID" ] = BSBEV
```

```

Calling customersDV.Delete(1)
customersDV[1].IsNew = False
customersDV[1].IsEdit = False

```

```
customersDV[2][ "CustomerID" ] = EASTC
```

```
Calling customersDV[2].Delete()
```

DataRowView objects in customersDV:

```

AROUT
Widgets Inc.
UK
CONSH
Consolidated Holdings
UK

```

ISLAT  
Island Trading  
UK

NORTS  
North/South  
UK

SEVES  
Seven Seas Imports  
UK

J7COM  
J7 Company  
UK

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## Creating Child *DataView* Objects

You can create a child *DataView* from a parent *DataRowView* using the *CreateChildView()* method. You can then view the *DataRowView* objects from the child *DataView*. To call the *CreateChildView()* method, you must first add a *DataRelation* to the *DataSet* that defines a relationship between the two underlying *DataTable* objects. (See [Chapter 12, "Navigating and Modifying Related Data,"](#) for information about *DataRelation* objects.)

Let's take a look at an example. Assume you have two *DataTable* objects named *customersDT* and *ordersDT*. Also assume you've added the following *DataRelation* to the *DataSet* that defines a relationship between *customersDT* and *ordersDT*:

```
DataRelation customersOrdersDataRel =
    new DataRelation(
        "CustomersOrders",
        customersDT.Columns[ "CustomerID" ],
        ordersDT.Columns[ "CustomerID" ]
    );
myDataSet.Relations.Add(
    customersOrdersDataRel
);
```

Finally, assume you have a *DataView* named *customersDV* that views the customers that have a *Country* column of UK. You can then call the *CreateChildView()* method from a *DataRowView* in *customersDV* to create a child *DataView*; notice that the name of the *DataRelation* (*CustomersOrders*) is passed to the *CreateChildView()* method:

```
DataView ordersDV = customersDV[ 0 ].CreateChildView( "CustomersOrders" );
```

The *ordersDV* *DataView* allows you to access the child rows from the *ordersDT* *DataTable*. The parent in this example is the first *DataRowView* from *customersDV* with a *CustomerID* of AROUT. The child *ordersDV* *DataView* contains *DataRowView* objects with the details of the orders for the AROUT customer.

**Note** The *CreateChildView()* method is overloaded. The other version of this method accepts a *DataRelation* object as the parameter.

[Listing 13.4](#) shows a complete example program.

### Listing 13.4: CREATECHILDDATAVIEW.CS

---

```
/*
CreateChildDataView.cs illustrates how to create a
child DataView
*/
using System;
using System.Data;
using System.Data.SqlClient;

class CreateChildDataView
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT CustomerID, CompanyName, Country " +
```

## Creating Child DataView Objects

```
"FROM Customers;" +
"SELECT OrderID, CustomerID " +
"FROM Orders;";

SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet);
mySqlConnection.Close();
myDataSet.Tables["Table"].TableName = "Customers";
myDataSet.Tables["Table1"].TableName = "Orders";
DataTable customersDT = myDataSet.Tables["Customers"];
DataTable ordersDT = myDataSet.Tables["Orders"];
// add a DataRelation object to myDataSet
DataRelation customersOrdersDataRel =
    new DataRelation(
        "CustomersOrders",
        customersDT.Columns["CustomerID"],
        ordersDT.Columns["CustomerID"]
    );
myDataSet.Relations.Add(
    customersOrdersDataRel
);

// create a DataView object named customersDV
DataView customersDV = new DataView();
customersDV.Table = customersDT;
customersDV.RowFilter = "Country = 'UK'";
customersDV.Sort = "CustomerID";

// display the first row in the customersDV DataView object
Console.WriteLine("Customer:");
for (int count = 0; count < customersDV.Table.Columns.Count; count++)
{
    Console.WriteLine(customersDV[0][count]);
}

// create a child DataView named ordersDV that views
// the child rows for the first customer in customersDV
DataView ordersDV = customersDV[0].CreateChildView("CustomersOrders");

// display the child rows in the customersDV DataView object
Console.WriteLine("\nOrderID's of the orders placed by this customer:");
foreach (DataRowView ordersDRV in ordersDV)
{
    Console.WriteLine(ordersDRV["OrderID"]);
}
}
```

The output from this program is as follows:

Customer:

AROUT

Around the Horn

UK

OrderID's of the orders placed by this customer:  
10355

10383  
10453  
10558  
10707  
10741  
10743  
10768  
10793  
10864  
10920  
10953  
11016

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## The *DataViewManager* Class

A *DataViewManager* allows you to centrally manage multiple *DataView* objects in a *DataSet*. A *DataViewManager* also allows you to create *DataView* objects on the fly at runtime. [Table 13.6](#) shows some of the *DataViewManager* properties.

**Table 13.6: *DataViewManager* PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
DataSet	DataSet	Gets or sets the <i>DataSet</i> used by your <i>DataViewManager</i> .
DataViewSettings	DataViewSettingCollection	Gets the <i>DataViewSettingCollection</i> for each <i>DataTable</i> in your <i>DataSet</i> . A <i>DataViewSettingCollection</i> gives you access to the properties of the <i>DataView</i> for each <i>DataTable</i> .

One of the *DataViewManager* methods is *CreateDataView()*. It creates a new *DataView* for the specified *DataTable*. The *DataTable* is passed as a parameter to the *CreateDataView()* method. Its return type is *DataView*.

One of the *DataViewManager* events is *ListChanged*. It fires when the list managed by a *DataView* in your *DataViewManager* changes. Its event handler is *ListChangedEventHandler*.

## Team LiB

## Creating and Using a *DataViewManager* Object

To create a *DataViewManager*, you use one of the following constructors:

```
public class DataViewManager
{
    public DataViewManager()
    public DataViewManager(DataSet myDataSet)
```

where *myDataSet* specifies the *DataSet* used by the *DataViewManager* object. This sets the *DataSet* property of the new *DataViewManager* object to *myDataSet*.

Let's take a look at an example of creating and using a *DataViewManager*. Assume you have a *DataSet* named *myDataSet*, which contains a *DataTable* populated with rows from the *Customers* table. The following example creates a *DataViewManager* object named *myDVM*, passing *myDataSet* to the constructor:

```
DataViewManager myDVM = new DataViewManager(myDataSet);
```

The next example sets the *Sort* and *RowFilter* properties that will be used later when a *DataGridView* for the *Customers* *DataTable* is created:

```
myDVM.DataViewSettings["Customers"].Sort = "CustomerID";
myDVM.DataViewSettings["Customers"].RowFilter = "Country = 'UK'";
```

**Note** The previous code doesn't actually create a *DataGridView*; it merely sets the properties of any *DataGridView* created in the future that views rows from the *Customers* *DataTable*.

The following example actually creates a *DataGridView* by calling the *CreateDataGridView()* method of the *myDVM* *DataViewManager*, passing the *customersDT* *DataTable* to *CreateDataGridView()*:

```
DataGridView customersDV = myDVM.CreateDataGridView(customersDT);
```

The *Sort* and *RowFilter* properties of the *customersDV* *DataGridView* are set to *CustomerID* and *Country = 'UK'* respectively. These are the same settings as those set earlier in the *DataViewSettings* property.

[Listing 13.4A](#) shows a complete example that creates and uses the *DataViewManager* shown in this section.

### **Listing 13.4A: USINGDATAVIEWMANAGER.CS**

---

```
/*
UsingDataViewManager.cs illustrates the use of a
DataViewManager object
*/

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

class UsingDataViewManager
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT CustomerID, CompanyName, Country " +
            "FROM Customers";
```

## Creating and Using a DataViewManager Object

```
SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
mySqlDataAdapter.SelectCommand = mySqlCommand;
DataSet myDataSet = new DataSet();
mySqlConnection.Open();
mySqlDataAdapter.Fill(myDataSet, "Customers");
mySqlConnection.Close();
DataTable customersDT = myDataSet.Tables["Customers"];

// create a DataViewManager object named myDVM
DataViewManager myDVM = new DataViewManager(myDataSet);

// set the Sort and RowFilter properties for the Customers DataTable
myDVM.DataViewSettings["Customers"].Sort = "CustomerID";
myDVM.DataViewSettings["Customers"].RowFilter = "Country = 'UK'";

// display the DataViewSettingCollectionString property of myDVM
Console.WriteLine("myDVM.DataViewSettingCollectionString = " +
    myDVM.DataViewSettingCollectionString + "\n");

// call the CreateDataView() method of myDVM to create a DataView
// named customersDV for the customersDT DataTable
DataView customersDV = myDVM.CreateDataView(customersDT);

// display the rows in the customersDV DataView object
foreach (DataRowView myDataRowView in customersDV)
{
    for (int count = 0; count < customersDV.Table.Columns.Count; count++)
    {
        Console.WriteLine(myDataRowView[count]);
    }
    Console.WriteLine(" ");
}
}
```

---

The output from this program is as follows:

```
myDVM.DataViewSettingCollectionString =
<DataViewSettingCollectionString>
<Customers Sort="CustomerID" RowFilter="Country = 'UK'" 
RowStateFilter="CurrentRows" />
</DataViewSettingCollectionString>
```

```
AROUT
Around the Horn
UK
BSBEV
B's Beverages
UK
```

```
CONSH
Consolidated Holdings
UK
```

```
EASTC
Eastern Connection
UK
```

```
ISLAT
```

Island Trading  
UK

NORTS  
North/South  
UK

SEVES  
Seven Seas Imports  
UK

**Team LiB**

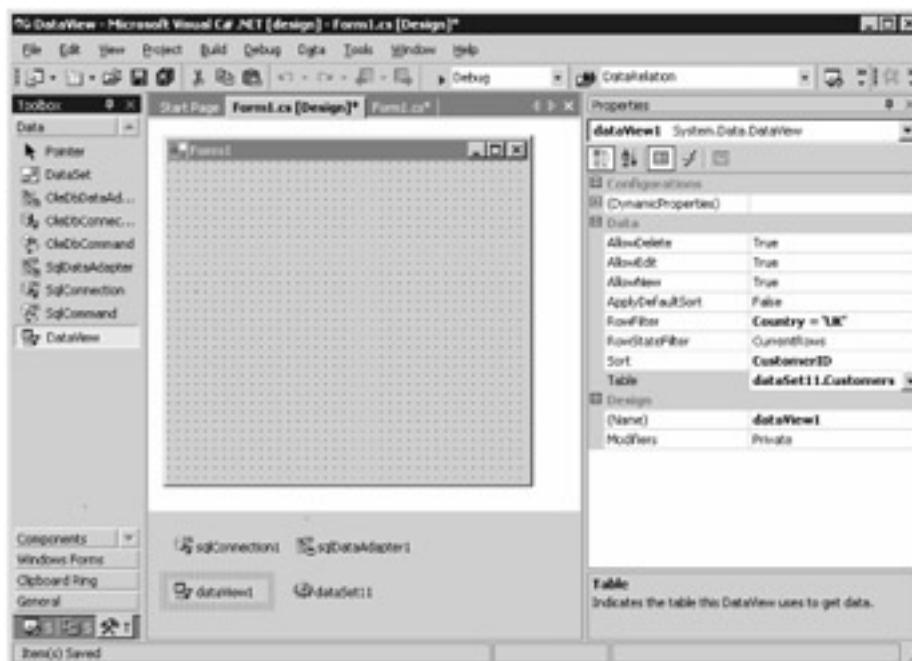
◀ PREVIOUS

NEXT ▶

## Creating a *DataView* Using Visual Studio .NET

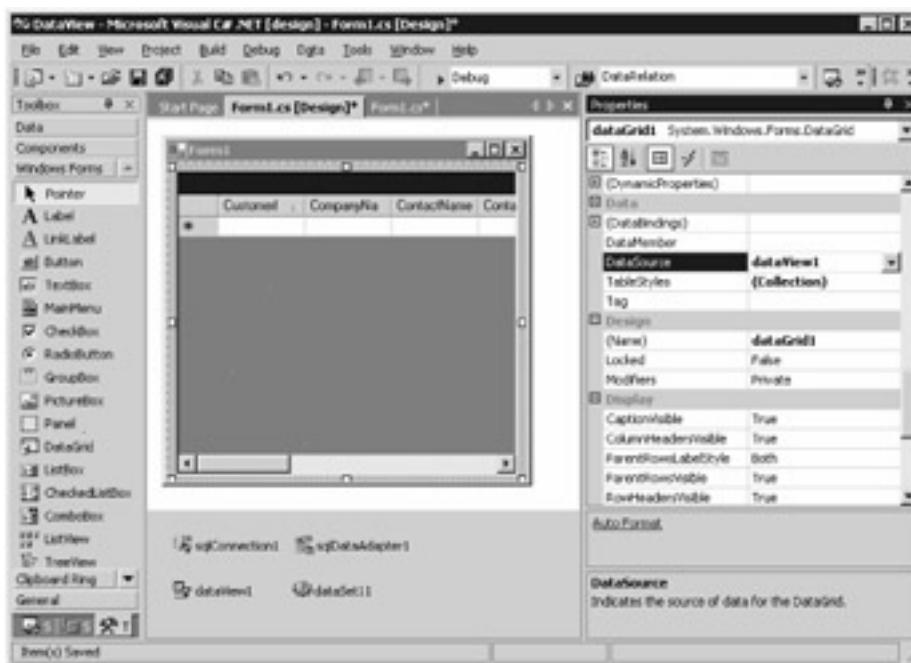
In this section, you'll learn how to create a *DataView* using Visual Studio .NET (VS .NET). You can follow along with the steps described in this section:

1. Open VS .NET and create a new Windows application named `myDataView`.
2. Display Server Explorer, connect to your Northwind database, and drag the `Customers` table to your form. This creates a `SqlConnection` object named `sqlConnection1` and a `SqlDataAdapter` object named `sqlDataAdapter1`. These objects are placed in the tray beneath your form.
3. Alter the `ConnectionString` property of `sqlConnection1` to connect to your Northwind database. Remember to add a substring containing the password (`pwd=sa`, or similar).
4. Click on the `sqlDataAdapter1` object in your form, and then click the `Generate Dataset` link at the bottom of the Properties window for `sqlDataAdapter1`. Accept the defaults in the dialog box, and click the `OK` button to create a `DataSet` object named `dataSet1`.
5. Drag a *DataView* object from the Data tab of the Toolbox to your form. This creates a *DataView* object named `dataView1`.
6. Set the `Table` property of your `dataView1` object to `dataSet1.Customers` using the drop-down list to the right of the `Table` property; set the `RowFilter` property to `Country = 'UK'`; and set the `Sort` property to `CustomerID`. See [Figure 13.1](#).



**Figure 13.1:** Setting the Properties of `dataView1`

7. Drag a *DataGridView* control from the Windows Forms tab of the Toolbox to your form. This creates a *DataGridView* object named `dataGridView1`.
8. Set the `DataSource` property of `dataGridView1` to `dataView1` using the drop-down list to the right of the `DataSource` property, as shown in [Figure 13.2](#). This binds the data stored in `dataView1` to `dataGridView1` and allows `dataGridView1` to access any data stored in `dataView1`.



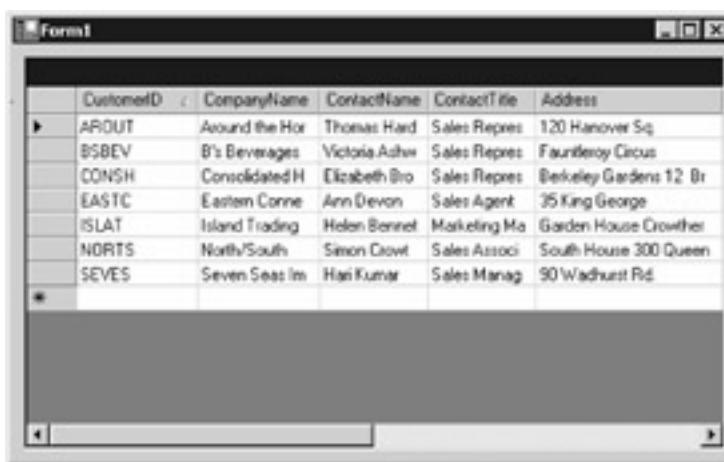
**Figure 13.2:** Setting the Properties of dataGridView1

9. Select View > Code and set the `Form1()` method of your form to

```
public Form1()
{
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    // call the Fill() method of sqlDataAdapter1
    // to populate dataSet11 with a DataTable named
    // Customers
    sqlDataAdapter1.Fill(dataSet11, "Customers");
}
```

Compile and run your form by pressing Ctrl+F5. [Figure 13.3](#) shows the running form. Notice that the information in the form comes from the `DataView` you created.



**Figure 13.3:** The running form

## Summary

In this chapter, you learned how to use `DataView` objects to filter and sort rows. The advantage of a `DataView` is that you can bind it to a visual component such as a `DataGridView` control.

A `DataView` stores copies of the rows in a `DataTable` as `DataRowView` objects. The `DataRowView` objects provide access to the underlying `DataRow` objects in a `DataTable`. Therefore, when you examine and edit the contents of a `DataRowView`, you are actually working with the underlying `DataRow`.

The `RowFilter` property of a `DataView` is similar to a `WHERE` clause in a `SELECT` statement. You can therefore use very powerful filter expressions in your `DataView`. For example, you can use `AND`, `OR`, `NOT`, `IN`, `LIKE`, comparison operators, arithmetic operators, wildcard characters (`*` and `%`), and aggregate functions.

You can find the index of a `DataRowView` in a `DataView` using the `Find()` method of a `DataView`. You can also get an array of `DataRowView` objects using the `FindRows()` method of a `DataView`.

A `DataViewManager` allows you to centrally manage multiple `DataView` objects in a `DataSet`. A `DataViewManager` also allows you to create `DataView` objects on the fly at runtime.

Team LiB

◀ PREVIOUS    NEXT ▶

## Part 3: Advanced Database Programming with ADO.NET

### Chapter List

[Chapter 14:](#) Advanced Transaction Control

[Chapter 15:](#) Introducing Web Applications-ASP.NET

[Chapter 16:](#) Using SQL Server's XML Support

[Chapter 17:](#) Web Services

Team LiB

◀ PREVIOUS    NEXT ▶

## Chapter 14: Advanced Transaction Control

### Overview

In [Chapter 3](#), "Introduction to the Structured Query Language," you saw how you can group SQL statements into transactions. These SQL statements are considered a logical unit of work. One example of this is a transfer of money from one bank account to another using two `UPDATE` statements, one that takes money out of one account, and another that puts that money into a different account. Both `UPDATE` statements may be considered to be a single transaction because both statements must be either committed or rolled back together, otherwise money might be lost.

Today's databases can handle many users and programs accessing the database at the same time, each potentially running their own transactions in the database. These are known as *concurrent* transactions because they are run at the same time. The database software must be able to satisfy the needs of all these concurrent transactions, as well as maintain the integrity of rows stored in the database tables. You can control the level of isolation that exists between your transactions and other transactions that might be running in the database.

In [Chapter 8](#), "Executing Database Commands," you saw how to use a transaction with a `Command` object. In [Chapter 11](#), "Using `DataSet` Objects to Modify Data," you saw how to use a transaction with a `DataAdapter`. In this chapter, you'll delve into advanced transaction control using SQL Server and ADO.NET.

Featured in this chapter:

- The `SqlTransaction` class
- ACID transaction properties
- [Setting a savepoint](#)
- [Setting the transaction isolation level](#)
- [Understanding SQL Server locks](#)

## The *SqlTransaction* Class

There are three Transaction classes: *SqlTransaction*, *OleDbTransaction*, and *OdbcTransaction*. You use a *Transaction* object to represent a database transaction, and an object of the *SqlTransaction* class to represent a database transaction in a SQL Server database. [Table 14.1](#) shows some of the *SqlTransaction* properties, and [Table 14.2](#) shows some of the *SqlTransaction* methods. You'll see the use of some of these properties and methods in this chapter.

**Table 14.1: *SqlTransaction* PROPERTIES**

PROPERTY	TYPE	DESCRIPTION
Connection	<i>SqlConnection</i>	Gets the connection for the transaction.
IsolationLevel	<i>IsolationLevel</i>	Gets the isolation level for the transaction (see " <a href="#">Setting the Transaction Isolation Level</a> ").

**Table 14.2: *SqlTransaction* METHODS**

METHOD	RETURN TYPE	DESCRIPTION
Commit()	void	Performs a commit to permanently record the SQL statements in the transaction.
Rollback()	void	Overloaded. Performs a rollback to undo the SQL statements in the transaction.
Save()	void	Creates a <i>savepoint</i> in the transaction that can be used to undo a portion of that transaction. The string passed to this method specifies the savepoint name. You can then roll back the transaction to that savepoint (see " <a href="#">Setting a Savepoint</a> ").

## Setting a Savepoint

You can set a savepoint anywhere within a transaction. This allows you to roll back any changes made to database rows after your savepoint. This might be useful if you have a very long transaction because if you make a mistake after you've set a savepoint, you don't have to roll back the transaction all the way to the start.

### Setting a Savepoint Using T-SQL

You set a savepoint in T-SQL using the `SAVE TRANSACTION` statement, or the shorthand version, `SAVE TRANS`. The syntax for this statement is as follows:

```
SAVE TRANS[ACTION] { savepointName | @savepointVariable }
```

where

- ***savepointName*** specifies a string containing the name you want to assign to your savepoint.
- ***savepointVariable*** specifies a T-SQL variable that contains your savepoint name. Your variable must be of the `char`, `varchar`, `nchar`, or `nvarchar` data type.

The following example sets a savepoint named `SaveCustomer`:

```
SAVE TRANSACTION SaveCustomer
```

Let's look at a complete T-SQL example script that sets a savepoint within a transaction. [Listing 14.1](#) shows a T-SQL script that performs the following steps:

1. Begins a transaction.
2. Inserts a row into the `Customers` table with a `CustomerID` of J8COM.
3. Sets a savepoint.
4. Inserts a row into the `Orders` table with a `CustomerID` of J8COM.
5. Performs a rollback to the savepoint, which undoes the previous insert performed in step 4, but preserves the insert performed in step 2.
6. Commits the transaction, which commits the row inserted into the `Customers` table in step 2.
7. Selects the new row from the `Customers` table.
8. Attempts to select the from the `Orders` table that was rolled back in step 5.
9. Deletes the new row from the `Customers` table.

#### **[Listing 14.1: SAVEPOINT.SQL](#)**

---

```
/*
 Savepoint.sql illustrates how to use a savepoint
 */

USE Northwind

- step 1: begin the transaction
BEGIN TRANSACTION

- step 2: insert a row into the Customers table
INSERT INTO Customers (
    CustomerID, CompanyName
```

## Setting a Savepoint

```
) VALUES (
    'J8COM', 'J8 Company'
)

- step 3: set a savepoint
SAVE TRANSACTION SaveCustomer
- step 4: insert a row into the Orders table
INSERT INTO Orders (
    CustomerID
) VALUES (
    'J8COM'
);

- step 5: rollback to the savepoint set in step 3
ROLLBACK TRANSACTION SaveCustomer

- step 6: commit the transaction
COMMIT TRANSACTION

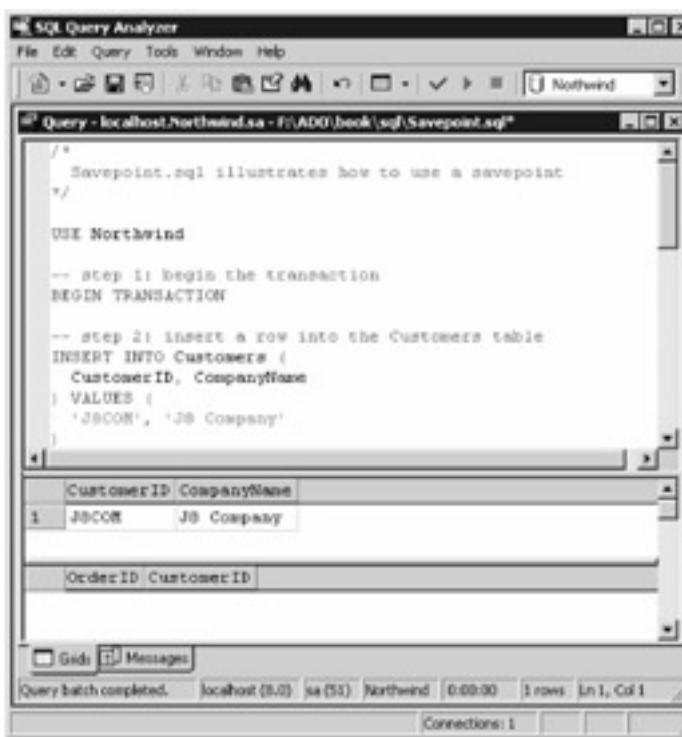
- step 7: select the new row from the Customers table
SELECT CustomerID, CompanyName
FROM Customers
WHERE CustomerID = 'J8COM'

- step 8: attempt to select the row from the Orders table
- that was rolled back in step 5
SELECT OrderID, CustomerID
FROM Orders
WHERE CustomerID = 'J8COM'

- step 9: delete the new row from the Customers table
DELETE FROM Customers
WHERE CustomerID = 'J8COM'
```

---

To run the `Savepoint.sql` script using Query Analyzer, you select File > Open, open the script from the `sql` directory, and then press F5 on the keyboard or select Query > Execute from the menu. [Figure 14.1](#) shows the `Savepoint.sql` script being run in Query Analyzer.



**Figure 14.1:** Running the `Savepoint.sql` script in Query Analyzer

## Setting a Savepoint Using a `SqlTransaction` Object

You set a savepoint in a `SqlTransaction` object by calling its `Save()` method, passing a string containing the name you wish to assign to your savepoint. Assume you have a `SqlTransaction` object named `mySqlTransaction`; the following example sets a savepoint named `SaveCustomer` by calling the `Save()` method of `mySqlTransaction`:

```
mySqlTransaction.Save("SaveCustomer");
```

You can then roll back any subsequent changes made to the rows in the database by calling the `Rollback()` method of `mySqlTransaction`, passing the savepoint name to the `Rollback()` method. For example:

```
mySqlTransaction.Rollback("SaveCustomer");
```

Let's look at a complete C# program that sets a savepoint within a transaction. [Listing 14.2](#) shows a program that performs the following steps:

1. Creates a `SqlTransaction` object named `mySqlTransaction`.
2. Creates a `SqlCommand` and sets its `Transaction` property to `mySqlTransaction`.
3. Inserts a row into the `Customers` table.
4. Sets a savepoint by calling the `Save()` method of `mySqlTransaction`, passing the name `SaveCustomer` to the `Save()` method.
5. Inserts a row into the `Orders` table.
6. Performs a rollback to the savepoint set in step 4, which undoes the previous insert performed in step 5, but preserves the insert performed in step 3.
7. Displays the new row added to the `Customers` table.
8. Deletes the new row from the `Customers` table.
9. Commits the transaction.

**Listing 14.2: SAVEPOINT.CS**

```
/*
 Savepoint.cs illustrates how to set a savepoint in a transaction
*/
using System;
using System.Data;
using System.Data.SqlClient;

class Savepoint
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        mySqlConnection.Open();

        // step 1: create a SqlTransaction object
        SqlTransaction mySqlTransaction =
            mySqlConnection.BeginTransaction();

        // step 2: create a SqlCommand and set its Transaction property
        // to mySqlTransaction
        SqlCommand mySqlCommand =
            mySqlConnection.CreateCommand();
        mySqlCommand.Transaction = mySqlTransaction;

        // step 3: insert a row into the Customers table
        Console.WriteLine("Inserting a row into the Customers table "+
            "with a CustomerID of J8COM");
        mySqlCommand.CommandText =
            "INSERT INTO Customers ( " +
            " CustomerID, CompanyName " +
            ") VALUES ( " +
            " 'J8COM', 'J8 Company' " +
            " )";
        int numberOfRows = mySqlCommand.ExecuteNonQuery();
        Console.WriteLine("Number of rows inserted = "+ numberOfRows);

        // step 4: set a savepoint by calling the Save() method of
        // mySqlTransaction, passing the name "SaveCustomer" to
        // the Save() method
        mySqlTransaction.Save("SaveCustomer");

        // step 5: insert a row into the Orders table
        Console.WriteLine("Inserting a row into the Orders table "+
            "with a CustomerID of J8COM");
        mySqlCommand.CommandText =
            "INSERT INTO Orders ( " +
            " CustomerID " +
            ") VALUES ( " +
            "'J8COM' " +
            " )";
        numberOfRows = mySqlCommand.ExecuteNonQuery();
        Console.WriteLine("Number of rows inserted = "+ numberOfRows);

        // step 6: rollback to the savepoint set in step 4
        Console.WriteLine("Performing a rollback to the savepoint");
    }
}
```

```

mySqlTransaction.Rollback("SaveCustomer");

// step 7: display the new row added to the Customers table
mySqlCommand.CommandText =
    "SELECT CustomerID, CompanyName " +
    "FROM Customers " +
    "WHERE CustomerID = 'J8COM'";
SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();
while (mySqlDataReader.Read())
{
    Console.WriteLine("mySqlDataReader[\"CustomerID\"] = " +
        mySqlDataReader["CustomerID"]);
    Console.WriteLine("mySqlDataReader[\"CompanyName\"] = " +
        mySqlDataReader["CompanyName"]);
}
mySqlDataReader.Close();

// step 8: delete the new row from the Customers table
Console.WriteLine("Deleting row with CustomerID of J8COM");
mySqlCommand.CommandText =
    "DELETE FROM Customers " +
    "WHERE CustomerID = 'J8COM'";
numberOfRows = mySqlCommand.ExecuteNonQuery();
Console.WriteLine("Number of rows deleted = " + numberOfRows);

// step 9: commit the transaction
Console.WriteLine("Committing the transaction");
mySqlTransaction.Commit();

mySqlConnection.Close();
}

}

```

---

The output from this program is as follows:

```

Inserting a row into the Customers table with a CustomerID of J8COM
Number of rows inserted = 1
Inserting a row into the Orders table with a CustomerID of J8COM
Number of rows inserted = 1
Performing a rollback to the savepoint
mySqlDataReader["CustomerID"] = J8COM
mySqlDataReader["CompanyName"] = J8 Company
Deleting row with CustomerID of J8COM
Number of rows deleted = 1
Committing the transaction

```

## Setting the Transaction Isolation Level

The transaction *isolation level* is the degree to which the changes made by one transaction are separated from other concurrent transactions. Before I get into the details of the various transaction isolation levels, you need to understand the types of problems that might occur when current transactions attempt to access the same rows in a table. In the following list, I'll use examples of two concurrent transactions that are accessing the same rows to illustrate the three types of potential transaction processing problems:

- **Phantoms** Transaction 1 reads a set of rows returned by a specified `WHERE` clause. Transaction 2 then inserts a new row, which also happens to satisfy the `WHERE` clause of the query previously used by Transaction 1. Transaction 1 then reads the rows again using the same query but now sees the additional row just inserted by Transaction 2. This new row is known as a "phantom," because to Transaction 1, this row seems to have magically appeared.
- **Nonrepeatable reads** Transaction 1 reads a row, and Transaction 2 updates the same row just read by Transaction 1. Transaction 1 then reads the same row again and discovers that the row it read earlier is now different. This is known as a "nonrepeatable read," because the row originally read by Transaction 1 has been changed.
- **Dirty reads** Transaction 1 updates a row but doesn't commit the update. Transaction 2 reads the updated row. Transaction 1 then performs a rollback, undoing the previous update. Now the row just read by Transaction 2 is no longer valid (or it's "dirty") because the update made by Transaction 1 wasn't committed when the row was read by Transaction 2.

To deal with these potential problems, databases implement various levels of transaction isolation to prevent concurrent transactions from interfering with each other. The SQL standard defines four isolation levels, which are shown in [Table 14.3](#). These levels are shown in order of increasing isolation.

**Table 14.3: SQL Standard Isolation Levels**

ISOLATION LEVEL	DESCRIPTION
READ UNCOMMITTED	Phantoms, nonrepeatable reads, and dirty reads are permitted.
READ COMMITTED	Phantoms and nonrepeatable reads are permitted, but dirty reads are not. This is the default for SQL Server.
REPEATABLE READ	Phantoms are permitted, but nonrepeatable and dirty reads are not.
SERIALIZABLE	Phantoms, nonrepeatable reads, and dirty reads are not permitted. This is the default for the SQL standard.

SQL Server supports all of these transaction isolation levels. The default transaction isolation level defined by the SQL standard is `SERIALIZABLE`, but the default used by SQL Server is `READ COMMITTED`, which is acceptable for most applications.

**Warning** When you set the transaction isolation level to `SERIALIZABLE`, any rows you access within a subsequent transaction will be "locked," meaning that no other transaction can modify those rows. Even rows you retrieve using a `SELECT` statement will be locked. You must commit or roll back the transaction to release the locks and allow other transactions to access the same rows. Use `SERIALIZABLE` only when you must ensure that your transaction is isolated from other transactions. You'll learn more about this later in the section "[Understanding SQL Server Locks](#)."

In addition, ADO.NET supports a number of transaction isolation levels, which are defined in the `System.Data.IsolationLevel` enumeration. [Table 14.4](#) shows the members of this enumeration.

**Table 14.4: IsolationLevel Enumeration Members**

ISOLATION LEVEL	DESCRIPTION
Chaos	Pending changes from more isolated transactions cannot be overwritten. SQL Server doesn't support this isolation level.
ReadCommitted	Phantoms and nonrepeatable reads are permitted, but dirty reads are not. This is the default.

ReadUncommitted	Phantoms, nonrepeatable reads, and dirty reads are permitted.
RepeatableRead	Phantoms are permitted, but nonrepeatable and dirty reads are not.
Serializable	Phantoms, nonrepeatable reads, and dirty reads are not permitted.
Unspecified	A different isolation level than the one specified is being used, but the level cannot be determined. SQL Server doesn't support this isolation level.

In the next two sections, you'll learn how to set the transaction isolation level using T-SQL and a `SqlTransaction` object.

## Setting the Transaction Isolation Level Using T-SQL

As well as learning to set the transaction isolation level using T-SQL, you'll see an example that shows the effect of setting different transaction isolation levels in SQL Server using the Query Analyzer tool.

To set the transaction isolation level in T-SQL, you use the `SET TRANSACTION ISOLATION LEVEL` command. The syntax for this command is as follows:

```
SET TRANSACTION ISOLATION LEVEL {
    READ COMMITTED |
    READ UNCOMMITTED |
    REPEATABLE READ |
    SERIALIZABLE
}
```

As you can see from the previous syntax, you can set the transaction isolation to any of the levels shown earlier in [Table 14.3](#).

The following example sets the transaction isolation level to `SERIALIZABLE`:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

**Note** The transaction isolation level is set for your session. Therefore, if you perform multiple transactions in a session, all your transactions will use the same level. If you want to change the level in your session, you simply execute another `SET TRANSACTION ISOLATION LEVEL` command with your new level. All subsequent transactions in your session will use the new level.

The following example sets the transaction isolation level to `READ COMMITTED`:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

Let's look at a complete example that sets the transaction isolation level using T-SQL. [Listing 14.3](#) shows an example T-SQL script that sets the transaction isolation level first to `SERIALIZABLE` and executes a transaction, and then sets the level to `READ COMMITTED` and executes another transaction.

### **[Listing 14.3: TransactionIsolation.sql](#)**

---

```
/*
 TransactionIsolation.sql illustrates how to set the
 transaction isolation level
*/
USE Northwind

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

BEGIN TRANSACTION
    SELECT CustomerID, CompanyName
    FROM Customers
```

## Setting the Transaction Isolation Level

```
WHERE CustomerID IN ( 'ALFKI' , 'J8COM' )

INSERT INTO Customers (
    CustomerID, CompanyName
) VALUES (
    'J8COM' , 'J8 Company'
)
UPDATE Customers
SET CompanyName = 'Widgets Inc.'
WHERE CustomerID = 'ALFKI'

SELECT CustomerID, CompanyName
FROM Customers
WHERE CustomerID IN ( 'ALFKI' , 'J8COM' )
COMMIT TRANSACTION

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

BEGIN TRANSACTION

    UPDATE Customers
    SET CompanyName = 'Alfreds Futterkiste'
    WHERE CustomerID = 'ALFKI'

    DELETE FROM Customers
    WHERE CustomerID = 'J8COM'

    SELECT CustomerID, CompanyName
    FROM Customers
    WHERE CustomerID IN ( 'ALFKI' , 'J8COM' )

    COMMIT TRANSACTION
```

---

[Figure 14.2](#) shows the `TransactionIsolation.sql` script being run in Query Analyzer. In the results pane in the lower half of Query Analyzer, the first two sets of rows are generated by the first transaction, and the final single row is generated by the second transaction.



**Figure 14.2:** Running the `TransactionIsolation.sql` script in Query Analyzer

## Setting the Transaction Isolation Level of a `SqlTransaction` Object

Along with setting the transaction isolation level of a `SqlTransaction` object, you'll see an example that shows the effect of setting different levels in a C# program.

You create a `SqlTransaction` object by calling the `BeginTransaction()` method of the `SqlConnection` object. This method is overloaded as follows:

```
SqlTransaction BeginTransaction()
SqlTransaction BeginTransaction(IsolationLevel myIsolationLevel)
SqlTransaction BeginTransaction(string transactionName)
SqlTransaction BeginTransaction(IsolationLevel myIsolationLevel, string
transactionName)
```

where

- **`myIsolationLevel`** specifies the isolation level of your transaction. This is a constant from the `System.Data.IsolationLevel` enumeration, for which members were shown earlier in [Table 14.4](#).
- **`transactionName`** specifies a string containing the name you want to assign to your transaction.

In the examples in this section, assume you have an open `SqlConnection` named `mySqlConnection` that is connected to the SQL Server Northwind database. The following example creates a `SqlTransaction` named `serializableTrans` by calling the `BeginTransaction()` method of `mySqlConnection`; notice that the `IsolationLevel` of `Serializable` is passed to `BeginTransaction()`:

```
SqlTransaction serializableTrans =
mySqlConnection.BeginTransaction(IsolationLevel.Serializable);
```

The next example creates a `SqlCommand` named `serializableCommand`, and sets its `Transaction` property to `serializableTrans`:

```
SqlCommand serializableCommand =
mySqlConnection.CreateCommand();
serializableCommand.Transaction = serializableTrans;
```

Any SQL statements performed using `serializableCommand` will now use `serializableTrans`, and will therefore be performed in a serializable transaction. The following example performs an `INSERT` statement that adds a row to the `Customers` table:

```
serializableCommand.CommandText =
    "INSERT INTO Customers (" +
    "CustomerID, CompanyName " +
    ") VALUES (" +
    "'J8COM', 'J8 Company' " +
    ")";
int numberOfRows = serializableCommand.ExecuteNonQuery();
```

The next example performs an `UPDATE` statement:

```
serializableCommand.CommandText =
    "UPDATE Customers " +
    "SET CompanyName = 'Widgets Inc.' " +
    "WHERE CustomerID = 'ALFKI'";
numberOfRows = serializableCommand.ExecuteNonQuery();
```

Finally, the following example commits the `INSERT` and `UPDATE` statements by calling the `Commit()` method of `serializableTrans`:

```
serializableTrans.Commit();
```

[Listing 14.4](#) shows a program that contains the following methods:

- **DisplayRows()** Selects and displays any rows from the `Customers` table with a `CustomerID` of `ALFKI` or `J8COM`.
- **PerformSerializableTransaction()** Performs the code shown earlier in this section to create a `SqlTransaction` object with an isolation level of `Serializable`, and uses it to perform an `INSERT` and `UPDATE` statement.
- **PerformReadCommittedTransaction()** Creates a `SqlTransaction` object with an isolation level of `ReadCommitted`, and uses it to perform `UPDATE` and `DELETE` statements.

#### **Listing 14.4: TransactionIsolation.cs**

---

```
/*
 TransactionIsolation.cs illustrates how to set the
 transaction isolation level
 */

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

class TransactionIsolation
{
    public static void DisplayRows(
        SqlCommand mySqlCommand
    )
    {
        mySqlCommand.CommandText =
            "SELECT CustomerID, CompanyName " +
            "FROM Customers " +
            "WHERE CustomerID IN ('ALFKI', 'J8COM')";
        SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();
        while (mySqlDataReader.Read())
        {
            Console.WriteLine("mySqlDataReader[\"CustomerID\"] = " +
                mySqlDataReader["CustomerID"]);
            Console.WriteLine("mySqlDataReader[\"CompanyName\"] = " +
```

## Setting the Transaction Isolation Level

```
        mySqlDataReader[ "CompanyName" ] );
    }
    mySqlDataReader.Close();
}

public static void PerformSerializableTransaction(
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn PerformSerializableTransaction()");

    // create a SqlTransaction object and start the transaction
    // by calling the BeginTransaction() method of the SqlConnection
    // object, passing the IsolationLevel of Serializable to the method
    SqlTransaction serializableTrans =
        mySqlConnection.BeginTransaction(IsolationLevel.Serializable);

    // create a SqlCommand and set its Transaction property
    // to serializableTrans
    SqlCommand serializableCommand =
        mySqlConnection.CreateCommand();
    serializableCommand.Transaction = serializableTrans;

    // call the DisplayRows() method to display rows from
    // the Customers table
    DisplayRows(serializableCommand);

    // insert a new row into the Customers table
    Console.WriteLine("Inserting new row into Customers table "+
        "with CustomerID of J8COM");
    serializableCommand.CommandText =
        "INSERT INTO Customers ("+
        "CustomerID, CompanyName "+
        ") VALUES ("+
        "'J8COM', 'J8 Company' "+
        ")";
    int numberOfRows = serializableCommand.ExecuteNonQuery();
    Console.WriteLine("Number of rows inserted = "+ numberOfRows);

    // update a row in the Customers table
    Console.WriteLine("Setting CompanyName to 'Widgets Inc.' for "+
        "row with CustomerID of ALFKI");
    serializableCommand.CommandText =
        "UPDATE Customers "+
        "SET CompanyName = 'Widgets Inc.' "+
        "WHERE CustomerID = 'ALFKI'";
    numberOfRows = serializableCommand.ExecuteNonQuery();
    Console.WriteLine("Number of rows updated = "+ numberOfRows);
    DisplayRows(serializableCommand);
    // commit the transaction
    serializableTrans.Commit();
}

public static void PerformReadCommittedTransaction(
    SqlConnection mySqlConnection
)
{
    Console.WriteLine("\nIn PerformReadCommittedTransaction()");

    // create a SqlTransaction object and start the transaction
```

## Setting the Transaction Isolation Level

```
// by calling the BeginTransaction() method of the SqlConnection
// object, passing the IsolationLevel of ReadCommitted to the method
// (ReadCommitted is actually the default)
SqlTransaction readCommittedTrans =
    mySqlConnection.BeginTransaction(IsolationLevel.ReadCommitted);

// create a SqlCommand and set its Transaction property
// to readCommittedTrans
SqlCommand readCommittedCommand =
    mySqlConnection.CreateCommand();
readCommittedCommand.Transaction = readCommittedTrans;

// update a row in the Customers table
Console.WriteLine("Setting CompanyName to 'Alfreds Futterkiste' "+
    "for row with CustomerID of ALFKI");
readCommittedCommand.CommandText =
    "UPDATE Customers "+
    "SET CompanyName = 'Alfreds Futterkiste' "+
    "WHERE CustomerID = 'ALFKI'";
int numberOfRows = readCommittedCommand.ExecuteNonQuery();
Console.WriteLine("Number of rows updated = " + numberOfRows);

// delete the new row from the Customers table
Console.WriteLine("Deleting row with CustomerID of J8COM");
readCommittedCommand.CommandText =
    "DELETE FROM Customers "+
    "WHERE CustomerID = 'J8COM'";
numberOfRows = readCommittedCommand.ExecuteNonQuery();
Console.WriteLine("Number of rows deleted = " + numberOfRows);

DisplayRows(readCommittedCommand);

// commit the transaction
readCommittedTrans.Commit();
}

public static void Main()
{
    SqlConnection mySqlConnection =
        new SqlConnection(
            "server=localhost;database=Northwind;uid=sa;pwd=sa"
        );
    mySqlConnection.Open();
    PerformSerializableTransaction(mySqlConnection);
    PerformReadCommittedTransaction(mySqlConnection);
    mySqlConnection.Close();
}
```

---

The output from this program is as follows:

```
In PerformSerializableTransaction()
mySqlDataReader["CustomerID"] = ALFKI
mySqlDataReader["CompanyName"] = Alfreds Futterkiste
Inserting new row into Customers table with CustomerID of J8COM
Number of rows inserted = 1
Setting CompanyName to 'Widgets Inc.' for row with CustomerID of ALFKI
Number of rows updated = 1
mySqlDataReader["CustomerID"] = ALFKI
mySqlDataReader["CompanyName"] = Widgets Inc.
```

## Setting the Transaction Isolation Level

```
mySqlDataReader[ "CustomerID" ] = J8COM
mySqlDataReader[ "CompanyName" ] = J8 Company

In PerformReadCommittedTransaction()
Setting CompanyName to 'Alfreds Futterkiste' for row with CustomerID of ALFKI
Number of rows updated = 1
Deleting row with CustomerID of J8COM
Number of rows deleted = 1
mySqlDataReader[ "CustomerID" ] = ALFKI
mySqlDataReader[ "CompanyName" ] = Alfreds Futterkiste
```

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Understanding SQL Server Locks

SQL Server uses *locks* to implement transaction isolation and to ensure the information stored in a database is consistent. Locks prevent one user from reading or changing a row that is being changed by another user. For example, when you update a row, a *row lock* is placed on that row to prevent another user from updating the row at the same time.

### Types of SQL Server Locks

SQL Server uses many types of locks, some of which are shown in [Table 14.5](#). This table shows the locks in ascending order of *locking granularity*, which refers to the size of the resource being locked. For example, a row lock has a finer granularity than a page lock.

**Table 14.5: SQL Server Lock Types**

LOCK TYPE	DESCRIPTION
Row (RID)	Placed on a row in a table. Stands for row identifier. Used to uniquely identify a row.
Key (KEY)	Placed on a row within an index. Used to protect key ranges in serializable transactions.
Page (PAG)	Placed on a <i>page</i> , which contains 8KB of row or index data.
Extent (EXT)	Placed on an <i>extent</i> , which is a contiguous group of 8 data or index pages.
Table (TAB)	Placed on a table and locks all the rows and indexes in that table.
Database (DB)	Used to lock the whole database when the database administrator puts it into single user mode for maintenance.

### SQL Server Locking Modes

SQL Server uses different *locking modes* that determine the level of locking placed on the resource. These locking modes are shown in [Table 14.6](#). You'll see these locking modes in the [next section](#).

**Table 14.6: SQL Server Locking Modes**

LOCKING MODE	DESCRIPTION
Shared (S)	Indicates that a transaction is going to read from the resource using a <code>SELECT</code> statement. Prevents other transactions from modifying the locked resource. A shared lock is released as soon as the data has been read-unless the transaction isolation level is set to <code>REPEATABLE READ</code> or <code>SERIALIZABLE</code> .
Update (U)	Indicates that a transaction intends to modify a resource using an <code>INSERT</code> , <code>UPDATE</code> , or <code>DELETE</code> statement. The lock must be <i>escalated</i> to an exclusive lock before the transaction actually performs the modification.
Exclusive (X)	Allows the transaction to modify the resource using an <code>INSERT</code> , <code>UPDATE</code> , or <code>DELETE</code> statement. No other transactions can read from or write to a resource on which an exclusive lock has been placed.
Intent shared (IS)	Indicates that the transaction intends to place a shared lock on some of the resources with a finer level of granularity within that resource. For example, placing an IS lock on a table indicates that the transaction intends to place a shared lock on some of the pages or rows within that table. No other transactions may place an exclusive lock on a resource that already has an IS lock on it.
Intent exclusive (IX)	Indicates that the transaction intends to place an exclusive lock on a resource with a finer level of granularity. No other transactions may place an exclusive lock on a resource that already has an IX lock on it.

Shared with intent exclusive (SIX)	Indicates that the transaction intends to read all of the resources that have a finer level of granularity <i>and</i> modify some of those resources. For example, placing a SIX lock on a table indicates that the transaction intends to read all the rows in that table and modify some of those rows. No other transactions may place an exclusive lock on a resource that already has a SIX lock on it.
Schema modification (Sch-M)	Indicates that a Data Definition Language (DDL) statement is going to be performed on a schema resource, for example, <code>DROP TABLE</code> . No other transactions may place a lock on a resource that already has a Sch-M lock on it.
Schema stability (Sch-S)	Indicates that a SQL statement that uses the resource is about to be performed, such as a <code>SELECT</code> statement for example. Other transactions <i>may place a lock</i> on a resource that already has a Sch-S lock on it; only a schema modification lock is prevented.
Bulk update (BU)	Indicates that a bulk copy operation to load rows into a table is to be performed. A bulk update lock allows other processes to bulk-copy data concurrently into the same table, but prevents other processes that are not bulk-copying data from accessing the table. For further information on bulk-copying data to a table, see the SQL Server Books Online documentation.

## Viewing SQL Server Lock Information

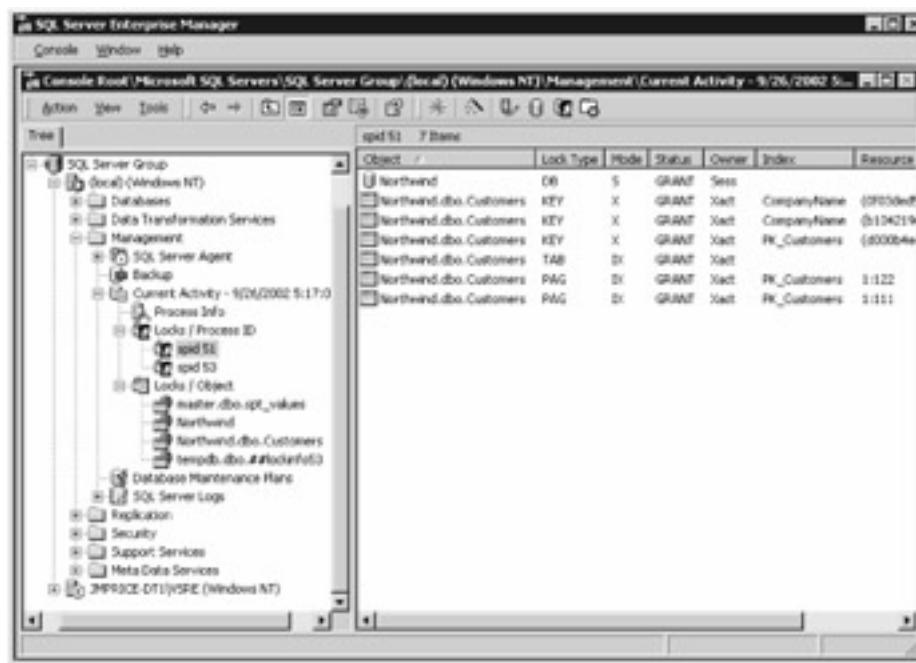
You can view the lock information in a database using SQL Server Enterprise Manager. You open the Management folder, open the Current Activity node, and then open either the Locks/Process ID node or the Locks/Object node. The Locks/Process ID node shows you the locks placed by each process; each process has a SPID number that is assigned by SQL Server to identify the process. The Locks/Object node shows you the locks placed on each resource by all processes.

**Tip** You can also view lock information by executing the `sp_lock` stored procedure, although Enterprise Manager organizes the information in a more readable format.

Assume you've started the following transaction (using Query Analyzer, for example) with the following T-SQL statements:

```
USE Northwind
BEGIN TRANSACTION
UPDATE Customers
SET CompanyName = 'Widgets Inc.'
WHERE CustomerID = 'ALFKI'
```

This places a shared lock on the Northwind database and a number of locks on the `Customers` table, which you can view using Enterprise Manager. [Figure 14.3](#) shows these locks using the Locks/ Process ID node of Enterprise Manager. The SPID of 51 corresponds to Query Analyzer where I ran the previous T-SQL statements. As you can see from this figure, a number of locks are placed by the previous T-SQL statements.



**Figure 14.3:** Viewing the locks using the Locks/ Process ID node of Enterprise Manager

To roll back the previous transaction, perform the following T-SQL statement:

```
ROLLBACK TRANSACTION
```

To release the locks, perform the following T-SQL statement:

```
COMMIT TRANSACTION
```

The information in the right pane of [Figure 14.3](#) shows the locks, and this information is divided into the following columns:

- **Object** The object being locked.
- **Lock Type** The type of lock, which corresponds to one of the types shown earlier in [Table 14.5](#).
- **Mode** The locking mode, which corresponds to one of the modes shown earlier in [Table 14.6](#).
- **Status** The lock status, which is either GRANT (lock was successfully granted), CNVT (lock was converted), or WAIT (waiting for lock).
- **Owner** The owner type of the lock, which is either Sess (session lock) or Xact (transaction lock).
- **Index** The name of the index being locked (if any).
- **Resource** The resource identifier of the object being locked (if any).

## Transaction Blocking

One transaction may *block* another transaction from obtaining a lock on a resource. For example, let's say you start a transaction using the following T-SQL, which is identical to the T-SQL in the [previous section](#):

```
USE Northwind
BEGIN TRANSACTION
UPDATE Customers
SET CompanyName = 'Widgets Inc.'
WHERE CustomerID = 'ALFKI'
```

As you saw in the [previous section](#), this places a number of locks on the *Customers* object.

If you then attempt to update the same row-without ending the previous transaction-using the following T-SQL statements:

```
USE Northwind
```

```
UPDATE Customers
SET CompanyName = 'Alfreds Futterkiste'
WHERE CustomerID = 'ALFKI'
```

then this UPDATE will wait until the first transaction has been committed or rolled back. [Figure 14.4](#) shows these two transactions being started in Query Analyzer. The first transaction, which is shown in the upper part of [Figure 14.4](#), is blocking the transaction on the bottom.

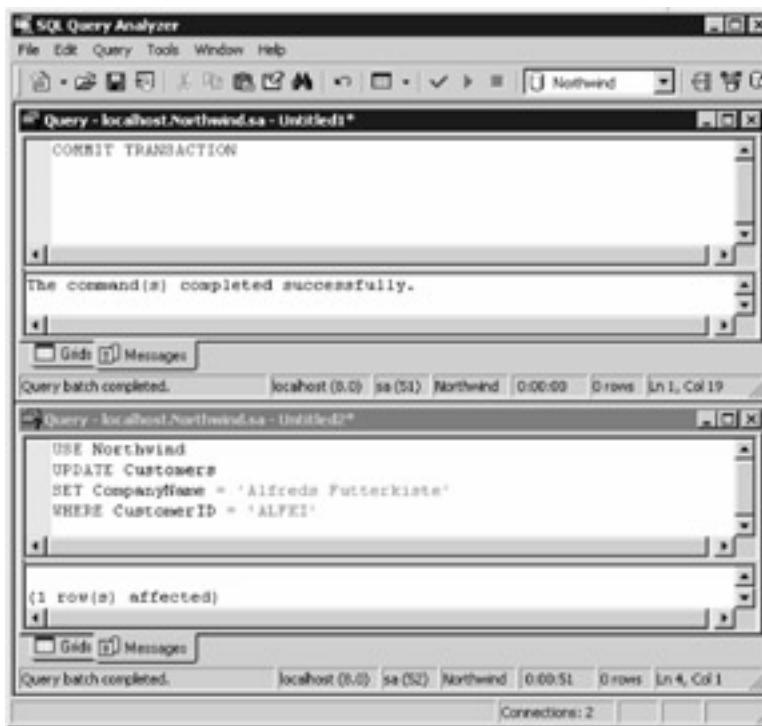


**Figure 14.4:** The transaction on the top part is blocking the transaction on the bottom.

To commit the previous transaction and release the locks for the first transaction, you may perform the following T-SQL statement:

```
COMMIT TRANSACTION
```

This allows the second UPDATE shown at the bottom of Query Analyzer to get the appropriate lock to update the row and proceed, as shown in [Figure 14.5](#).



**Figure 14.5:** Once the top transaction is committed, the bottom UPDATE proceeds.

## Setting the Lock Timeout

By default, a SQL statement will wait indefinitely to get a lock. You can change this by executing the `SET LOCK_TIMEOUT` command. For example, the following command sets the lock timeout to 1 second (1,000 milliseconds):

```
SET LOCK_TIMEOUT 1000
```

If a SQL statement has to wait longer than 1 second, SQL Server will return an error and cancel the SQL statement.

You can execute the `SET LOCK_TIMEOUT` command in C# code also. For example:

```
mySqlCommand.CommandText = "SET LOCK_TIMEOUT 1000";
mySqlCommand.ExecuteNonQuery();
```

You'll see the use of the `SET LOCK_TIMEOUT` command in the [next section](#).

## Blocking and Serializable/Repeatable Read Transactions

Serializable and repeatable read transactions lock the rows they are retrieving so that other transactions cannot update those rows. Serializable and repeatable read transactions do this so that the rows aren't changed after they've been read.

For example, if you select the row from the `Customers` table with a `CustomerID` of `ALFKI` using a serializable transaction, and then attempt to update that row using a second transaction, then the second transaction will be blocked. It is blocked because the serializable transaction locks the retrieved row and the second transaction is unable to get a lock on that row.

[Listing 14.5](#) shows an example of this. The second transaction sets the lock timeout to 1 second. This means the program throws a `SqlException` rather than simply hanging when the second transaction is unable to get a lock on the `ALFKI` row in the `Customers` table.

### **Listing 14.5: Block.cs**

---

```
/*
Block.cs illustrates how a serializable command locks
the rows it retrieves so that a second transaction
cannot get a lock to update one of these retrieved rows
that has already been locked
```

\*/

```

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

class Block
{
    public static void DisplayRows(
        SqlCommand mySqlCommand
    )
    {
        mySqlCommand.CommandText =
            "SELECT CustomerID, CompanyName " +
            "FROM Customers " +
            "WHERE CustomerID IN ('ALFKI', 'J8COM')";
        SqlDataReader mySqlDataReader = mySqlCommand.ExecuteReader();
        while (mySqlDataReader.Read())
        {
            Console.WriteLine("mySqlDataReader[\"CustomerID\"] = " +
                mySqlDataReader["CustomerID"]);
            Console.WriteLine("mySqlDataReader[\"CompanyName\"] = " +
                mySqlDataReader["CompanyName"]);
        }
        mySqlDataReader.Close();
    }

    public static void Main()
    {
        // create and open two SqlConnection objects
        SqlConnection serConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlConnection rcConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        serConnection.Open();
        rcConnection.Open();

        // create the first SqlTransaction object and start the transaction
        // by calling the BeginTransaction() method of the SqlConnection
        // object, passing the IsolationLevel of Serializable to the method
        SqlTransaction serializableTrans =

            serConnection.BeginTransaction(IsolationLevel.Serializable);

        // create a SqlCommand and set its Transaction property
        // to serializableTrans
        SqlCommand serializableCommand =
            serConnection.CreateCommand();
        serializableCommand.Transaction = serializableTrans;

        // call the DisplayRows() method to display rows from
        // the Customers table;
        // this causes the rows to be locked, if you comment
        // out the following line then the INSERT and UPDATE
        // performed later by the second transaction will succeed
        DisplayRows(serializableCommand); // *
    }
}

```

```

// create the second SqlTransaction object
SqlTransaction readCommittedTrans =
    rcConnection.BeginTransaction(IsolationLevel.ReadCommitted);

// create a SqlCommand and set its Transaction property
// to readCommittedTrans
SqlCommand readCommittedCommand =
    rcConnection.CreateCommand();
readCommittedCommand.Transaction = readCommittedTrans;
// set the lock timeout to 1 second using the
// SET LOCK_TIMEOUT command
readCommittedCommand.CommandText = "SET LOCK_TIMEOUT 1000";
readCommittedCommand.ExecuteNonQuery();

try
{
    // insert a new row into the Customers table
    Console.WriteLine("Inserting new row into Customers table "+
        "with CustomerID of J8COM");
    readCommittedCommand.CommandText =
        "INSERT INTO Customers (" +
        "CustomerID, CompanyName " +
        ") VALUES ( " +
        " 'J8COM', 'J8 Company' " +
        ")";
    int numberOfRows = readCommittedCommand.ExecuteNonQuery();
    Console.WriteLine("Number of rows inserted = " + numberOfRows);

    // update the ALFKI row in the Customers table
    Console.WriteLine("Setting CompanyName to 'Widgets Inc.' for "+
        "for row with CustomerID of ALFKI");
    readCommittedCommand.CommandText =
        "UPDATE Customers "+
        "SET CompanyName = 'Widgets Inc.' "+
        "WHERE CustomerID = 'ALFKI'";
    numberOfRows = readCommittedCommand.ExecuteNonQuery();
    Console.WriteLine("Number of rows updated = " + numberOfRows);

    // display the new rows and rollback the changes
    DisplayRows(readCommittedCommand);
    Console.WriteLine("Rolling back changes");
    readCommittedTrans.Rollback();
}
catch (SqlException e)
{
    Console.WriteLine(e);
}
finally
{
    serConnection.Close();
    rcConnection.Close();
}
}
}

```

---

**Warning**

If you compile and run this program as is, then it will throw a *SqlException*. This is intentional, as it shows you that the attempt to get a lock timed out. If you comment out the first call to the *DisplayRows()* method in this program [marked with an asterisk (\*)], then the program doesn't throw a *SqlException*. This is because commenting out the first call to *DisplayRows()* stops the serializable transaction from retrieving and therefore locking the rows. The second transaction is then able to get the lock on the ALFKI row.

The output from this program is as follows (notice it throws a *SqlException* when the lock timeout period is exceeded):

```
mySqlDataReader[ "CustomerID" ] = ALFKI
mySqlDataReader[ "CompanyName" ] = Alfreds Futterkiste
Inserting new row into Customers table with CustomerID of J8COM
System.Data.SqlClient.SqlException: Lock request time out period exceeded.
at System.Data.SqlClient.SqlConnection.OnError(SqlException exception,
TdsParserState state)
at System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception,
TdsParserState state)
at System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning()
at System.Data.SqlClient.TdsParser.Run(RunBehavior run, SqlCommand cmdHandler,
SqlDataReader dataStream)
at System.Data.SqlClient.SqlCommand.ExecuteNonQuery()
at Block.Main()
```

Try commenting out the first call to *DisplayRows()* in the program, and then recompile and run it again. This time the second transaction will be able to get the lock on the row and proceed.

## Deadlocks

A **deadlock** occurs when two transactions are waiting for locks that the other transaction currently has. Consider the following two transactions:

Transaction 1 (T1):

```
BEGIN TRANSACTION
UPDATE Customers
SET CompanyName = 'Widgets Inc.'
WHERE CustomerID = 'ALFKI'
UPDATE Products
SET ProductName = 'Widget'
WHERE ProductID = 1
COMMIT TRANSACTION
```

Transaction 2 (T2):

```
BEGIN TRANSACTION
UPDATE Products
SET ProductName = ' Chai '
WHERE ProductID = 1
UPDATE Customers
SET CompanyName = ' Alfreds Futterkiste '
WHERE CustomerID = 'ALFKI'
COMMIT TRANSACTION
```

Notice that T1 and T2 both update the same rows in the *Customers* and *Products* table. If T1 and T2 are executed in series at different times, T1 is executed and completed, followed by T2, then there's no problem. However, if T1 and T2 are executed at the same time with their *UPDATE* statements intermeshed, then a deadlock occurs. Let's take a look at an example of this using the following steps:

1. T1 is started.
2. T2 is started.

3. T1 locks the `Customers` row and updates the row.
4. T2 locks the `Products` row and updates the row.
5. T2 waits for the lock on the `Products` row, which is currently held by T1.
6. T1 waits for the lock the `Customers` row, which is currently held by T2.

In step 5, T2 is waiting for a lock held by T1. In step 6, T1 is waiting for a lock held by T2. Thus, a deadlock occurs since both transactions are waiting for each other. Both transactions hold mutually required locks. SQL Server will detect the deadlock and roll back one of the transactions. SQL Server rolls back the transaction that is the least expensive to undo, and also returns an error indicating that a deadlock occurred.

You can also choose the transaction that is to be rolled back using the T-SQL `SET DEADLOCK_PRIORITY` command, which uses the following syntax:

```
SET DEADLOCK_PRIORITY { LOW | NORMAL | @variable }
```

where

- **LOW** indicates the transaction has a low priority and is the one to roll back in the event of a deadlock.
- **NORMAL** indicates that the default rule is to be applied, meaning the least expensive transaction is rolled back.
- **@variable** is a T-SQL character variable that you set to 3 for `LOW` or 6 for `NORMAL`.

For example, the following command sets `DEADLOCK_PRIORITY` to `LOW`:

```
SET DEADLOCK_PRIORITY LOW
```

You can also execute the `SET DEADLOCK_PRIORITY` command in C# code. For example:

```
t2Command.CommandText = "SET DEADLOCK_PRIORITY LOW";
t2Command.ExecuteNonQuery();
```

**Tip** You can reduce the risk of a deadlock occurring in your program by keeping your transactions as short as possible; that way, the locks are held on the database objects for as short a period as possible. You should also access tables in the same order when executing multiple transactions at the same time; that way, you reduce the risk of transactions holding mutually required locks.

[Listing 14.6](#) shows a program that illustrates the T1 and T2 transaction deadlock scenario described earlier. Each `UPDATE` is executed using a separate thread to simulate the interleaved updates shown earlier in the six steps.

### **Listing 14.6: Deadlock.cs**

---

```
/*
Deadlock.cs illustrates how two transactions can
deadlock each other
*/
using System;
using System.Data;
using System.Data.SqlClient;
using System.Threading;

class Deadlock
{
    // create two SqlConnection objects
    public static SqlConnection t1Connection =
        new SqlConnection(
            "server=localhost;database=Northwind;uid=sa;pwd=sa"
        );
}
```

```
public static SqlConnection t2Connection =
    new SqlConnection(
        "server=localhost;database=Northwind;uid=sa;pwd=sa"
    );
// declare two SqlTransaction objects
public static SqlTransaction t1Trans;
public static SqlTransaction t2Trans;

// declare two SqlCommand objects
public static SqlCommand t1Command;
public static SqlCommand t2Command;

public static void UpdateCustomerT1()
{
    // update the row with a CustomerID of ALFKI
    // in the Customers table using t1Command
    Console.WriteLine("Setting CompanyName to 'Widgets Inc.' "+
        "for row with CustomerID of ALFKI using t1Command");
    t1Command.CommandText =
        "UPDATE Customers "+
        "SET CompanyName = 'Widgets Inc.' "+
        "WHERE CustomerID = 'ALFKI'";
    int numberOfRows = t1Command.ExecuteNonQuery();
    Console.WriteLine("Number of rows updated = "+ numberOfRows);
}

public static void UpdateProductT2()
{
    // update the row with a ProductID of 1
    // in the Products table using t2Command
    Console.WriteLine("Setting ProductName to 'Widget' "+
        "for the row with ProductID of 1 using t2Command");
    t2Command.CommandText =
        "UPDATE Products "+
        "SET ProductName = 'Widget' "+
        "WHERE ProductID = 1";
    int numberOfRows = t2Command.ExecuteNonQuery();
    Console.WriteLine("Number of rows updated = "+ numberOfRows);
}

public static void UpdateProductT1()
{
    // update the row with a ProductID of 1
    // in the Products table using t1Command
    Console.WriteLine("Setting ProductName to 'Chai' "+
        "for the row with ProductID of 1 using t1Command");
    t1Command.CommandText =
        "UPDATE Products "+
        "SET ProductName = 'Chai' "+
        "WHERE ProductID = 1";
    int numberOfRows = t1Command.ExecuteNonQuery();
    Console.WriteLine("Number of rows updated = "+ numberOfRows);
}

public static void UpdateCustomerT2()
{
    // update the row with a CustomerID of ALFKI
    // in the Customers table using t2Command
    Console.WriteLine("Setting CompanyName to 'Alfreds Futterkiste' "+
        "for row with CustomerID of ALFKI using t2Command");
```

```

t2Command.CommandText =
    "UPDATE Customers "+ 
    "SET CompanyName = 'Alfreds Futterkiste' "+
    "WHERE CustomerID = 'ALFKI'";
int numberOfRows = t2Command.ExecuteNonQuery();
Console.WriteLine("Number of rows updated = " + numberOfRows);
}

public static void Main()
{
    // open the first connection, begin the first transaction,
    // and set the lock timeout to 5 seconds
    t1Connection.Open();
    t1Trans = t1Connection.BeginTransaction();
    t1Command = t1Connection.CreateCommand();
    t1Command.Transaction = t1Trans;
    t1Command.CommandText = "SET LOCK_TIMEOUT 5000";
    t1Command.ExecuteNonQuery();

    // open the second connection, begin the second transaction,
    // and set the lock timeout to 5 seconds
    t2Connection.Open();
    t2Trans = t2Connection.BeginTransaction();
    t2Command = t2Connection.CreateCommand();
    t2Command.Transaction = t2Trans;
    t2Command.CommandText = "SET LOCK_TIMEOUT 5000";
    t2Command.ExecuteNonQuery();

    // set DEADLOCK_PRIORITY to LOW for the second transaction
    // so that it is the transaction that is rolled back
    t2Command.CommandText = "SET DEADLOCK_PRIORITY LOW";
    t2Command.ExecuteNonQuery();

    // create four threads that will perform the interleaved updates
    Thread updateCustThreadT1 = new Thread(new ThreadStart(UpdateCustomerT1));
    Thread updateProdThreadT2 = new Thread(new ThreadStart(UpdateProductT2));
    Thread updateProdThreadT1 = new Thread(new ThreadStart(UpdateProductT1));
    Thread updateCustThreadT2 = new Thread(new ThreadStart(UpdateCustomerT2));

    // start the threads to actually perform the interleaved updates
    updateCustThreadT1.Start();
    updateProdThreadT2.Start();
    updateProdThreadT1.Start();
    updateCustThreadT2.Start();
}
}

```

---

**Note** You can think of a thread as a separate process in your program, and each thread appears to execute in parallel with the other threads. For a detailed discussion of threads, see the book *Mastering Visual C# .NET* by Jason Price and Mike Gunderloy (Sybex, 2002).

The program shown in [Listing 14.6](#) contains the following methods:

- **UpdateCustomerT1()** Updates the row with a CustomerID of ALFKI in the Customers table using the first transaction. Specifically, it sets the CompanyName to Widgets Inc.
- **UpdateProductT2()** Updates the row with a ProductID of 1 in the Products table using the second transaction. Specifically, it sets the ProductName to Widget.

- **UpdateProductT1()** Updates the row with a ProductID of 1 in the Products table using the first transaction. Specifically, it sets the ProductName to Chai.
- **UpdateCustomerT2()** Updates the row with a CustomerID of ALFKI in the Customers table using the second transaction. Specifically, it sets the CompanyName to Alfreds Futterkiste.

These methods will be called by the threads to perform the interleaved updates.

**Note** This program indicates that the second transaction is to be rolled back when deadlock occurs using the *SET DEADLOCK\_PRIORITY LOW* command.

The output from this program is as follows:

```
Setting CompanyName to 'Widgets Inc.' for row
  with CustomerID of ALFKI using t1Command
Number of rows updated = 1
Setting ProductName to 'Widget' for the row
  with ProductID of 1 using t2Command
Number of rows updated = 1
Setting ProductName to 'Chai' for the row
  with ProductID of 1 using t1Command
Setting CompanyName to 'Alfreds Futterkiste' for row
  with CustomerID of ALFKI using t2Command
```

Unhandled Exception:

```
System.Data.SqlClient.SqlException: Transaction (Process ID 53) was deadlocked on
{lock} resources with another process and has been chosen as the deadlock victim.
Rerun the transaction.
at System.Data.SqlClient.SqlConnection.OnError(SqlException exception,
  TdsParserState state)
at System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception,
  TdsParserState state)
at System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning()
at System.Data.SqlClient.TdsParser.Run(RunBehavior run, SqlCommand cmdHandler,
  SqlDataReader dataStream)
at System.Data.SqlClient.SqlCommand.ExecuteNonQuery()
at Deadlock.UpdateCustomerT2()
Number of rows updated = 1
```

**Team LiB****◀ PREVIOUS****NEXT ▶**

## Summary

Today's databases can handle many users and programs accessing the database at the same time, each potentially running their own transactions in the database. The database software must be able to satisfy the needs of all these concurrent transactions, as well as maintain the integrity of rows stored in the database tables. You can control the level of isolation that exists between your transactions and other transactions that might be running in the database.

In this chapter, you delved into advanced transaction control using SQL Server and ADO.NET. Specifically, you saw how to set a savepoint, roll back a transaction to that savepoint, and set the transaction isolation level. You also learned about SQL Server locks and how transactions can block and deadlock each other.

In the [next chapter](#), you'll learn about XML.

**Team LiB****◀ PREVIOUS****NEXT ▶**

**Team LiB**[◀ PREVIOUS](#)[NEXT ▶](#)

## Chapter 15: Introducing Web Applications-ASP.NET

### Overview

Active Server Pages for .NET (ASP.NET) enables you to create dynamic Web pages with content that can change at runtime and to develop applications that are accessed using a Web browser. For example, you could develop an e-commerce application that allows users to order products over the Web, or a stock-trading application that allows users to place trades for shares in companies.

ASP.NET is conceptually similar to its rival JavaServer Pages (JSP) in that you request a page from a server using a Web browser, and the server responds by running the ASP.NET page. The server then sends back HTML that is displayed in your browser.

In this chapter, you'll learn the basics of ASP.NET, and you'll see how to use Visual Studio .NET (VS .NET) to create ASP.NET applications using C# as the programming language. Featured in this chapter:

- Creating ASP.NET Web applications
- The Web form controls
- Using `DataGridView` and `DataSource` controls to access a database
- Maintaining state in a Web application
- Creating a simple shopping-cart application

**Team LiB**[◀ PREVIOUS](#)[NEXT ▶](#)

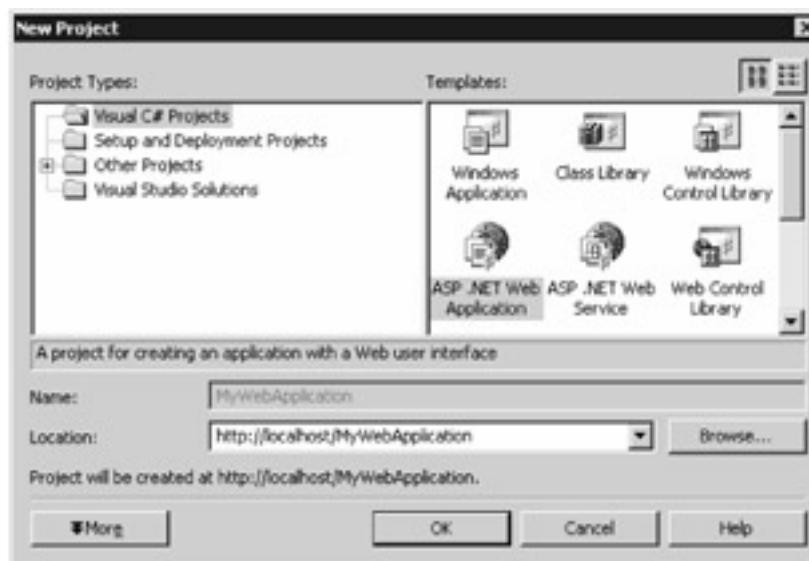
## Creating a Simple ASP.NET Web Application Using VS .NET

In this section, you'll see how to create a simple ASP.NET Web application that contains a text box and a button using VS .NET. When you press the button, a string of text will appear in your text box. You'll learn how to deploy this application to Microsoft's Internet Information Server (IIS). You'll also see how to run the example Web application from Internet Explorer.

**Note** IIS is software that allows you to run ASP.NET Web applications and display HTML pages. To deploy the ASP.NET applications shown in this chapter, you'll need access to a computer that runs IIS, along with the FrontPage Server Extensions. These extensions allow you to deploy an ASP.NET Web application from Visual Studio .NET. You can find full information on installing IIS and the FrontPage Server Extensions in the Windows online help documentation; to access this documentation, select Start ➤ Help.

Perform the following steps:

1. Start Visual Studio .NET (VS .NET) and select File ➤ New Project. Select Visual C# Projects from the Project Types area on the left of the New Project dialog box, and select ASP .NET Web Application from the Templates area on the right. Enter **http://localhost/MyWeb-Application** in the Location field, as shown in [Figure 15.1](#).



**Figure 15.1:** Creating an ASP.NET Web application in Visual Studio .NET

**Note** The name *localhost* represents your local computer on which you are developing your Web application. If you're using IIS that is running on a computer other than your local computer, you should replace *localhost* with the name of the remote computer.

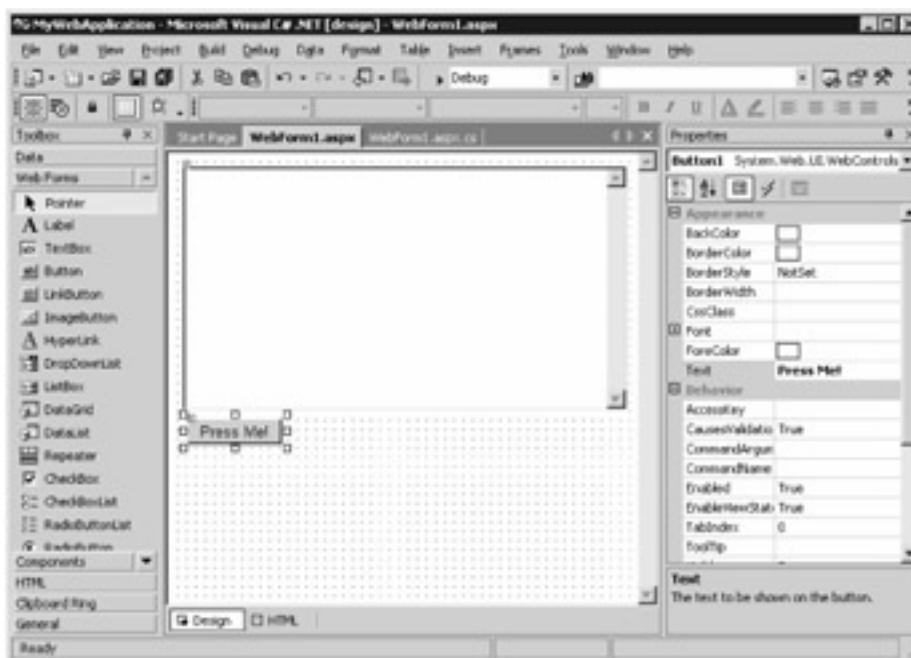
2. Click the OK button to continue. VS .NET will create a new directory named **MyWebApplication** in the **wwwroot** directory; this is the directory where IIS stores published Web pages and applications. After you click the OK button, you'll see the new application being sent to IIS.

Once your application has been deployed to IIS, VS .NET will display a blank Web form. You can think of the Web form as the canvas on which you can place controls, such as text boxes and buttons. When you later run your form, you'll see that the page displayed by the Web browser is laid out in a similar manner to your form.

3. Add a **TextBox** control to your form. The default value for the **ID** property of your **TextBox** control is **TextBox1**.

**Note** You use the **ID** property when referencing a Web control in C# code. You'll see an example of code that does this shortly.

4. Set the **TextMode** property for **TextBox1** to **MultiLine**; this allows the text to be displayed on more than one line. Next, add a **Button** control to your form. The default **ID** for your **Button** control is **Button1**. Set the **Text** property for **Button1** to **Press Me!** [Figure 15.2](#) shows the form with the **TextBox** and **Button** controls.



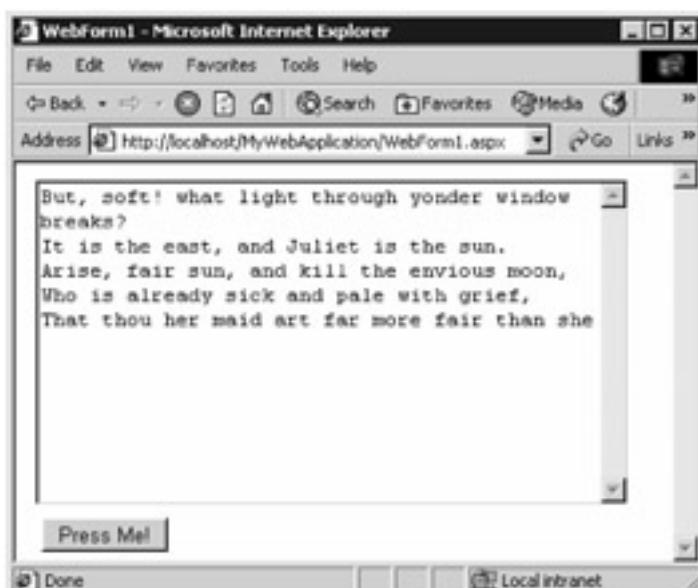
**Figure 15.2:** Adding TextBox and Button controls to the form

5. Next, you'll add a line of code to the `Button1_Click()` method. This method is executed when `Button1` is pressed in your running form. The statement you'll add to `Button1_Click()` will set the `Text` property of `TextBox1` to a string. This string will contain a line from Shakespeare's *Romeo and Juliet*. To add the code, double-click `Button1` and enter the following code in the `Button1_Click()` method:

```
TextBox1.Text =
    "But, soft! what light through yonder window breaks?\n" +
    "It is the east, and Juliet is the sun.\n" +
    "Arise, fair sun, and kill the envious moon,\n" +
    "Who is already sick and pale with grief,\n" +
    "That thou her maid art far more fair than she";
```

**Note** If you're a Shakespeare fan, you'll recognize these lines from the magnificent balcony scene in which Romeo professes his true love for Juliet.

6. You're now ready to run your form. Select `Debug > Start Without Debugging`, or press `Ctrl+F5` on the keyboard to run your form (see [Figure 15.3](#)).



**Figure 15.3:** The running form

Now that you've created and run the form, let's examine the code generated by VS .NET. There are two main parts to the code:

- The `WebForm1.aspx` file, which contains HTML and ASP.NET code.
- The `WebForm1.aspx.cs` file, which contains C# code that supports the Web form. You can think of this C# code as *running behind* the form, and for this reason the `WebForm1.aspx.cs` file is known as the *code-behind file*.

**Note** The extension `.aspx` identifies ASP.NET files.

You'll examine the details of the `WebForm1.aspx` and `WebForm1.aspx.cs` files in the following sections.

## The `WebForm1.aspx` File

You can view the HTML containing the ASP.NET tags for your form by clicking the HTML link at the bottom of the form designer. Click the HTML link to view the code for your form. [Listing 15.1](#) shows the contents of the `WebForm1.aspx` file.

### **Listing 15.1: WebForm1.aspx**

---

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
AutoEventWireup="false"
Inherits="MyWebApplication.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
<HEAD>
<title>WebForm1</title>
<meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
<meta content="C#" name="CODE_LANGUAGE">
<meta content="JavaScript" name="vs_defaultClientScript">
<meta content="http://schemas.microsoft.com/intellisense/ie5"
name="vs_targetSchema">
</HEAD>
<body MS_POSITIONING="GridLayout">
<form id="Form1" method="post" runat="server">
<asp:TextBox id="TextBox1" style="Z-INDEX: 101; LEFT: 13px;
POSITION: absolute; TOP: 11px" runat="server"
Width="386px" Height="212px"
TextMode="MultiLine"></asp:TextBox>
<asp:Button id="Button1" style="Z-INDEX: 102; LEFT: 17px;
POSITION: absolute; TOP: 231px" runat="server" Width="82px" Height="22px"
Text="Press Me!"></asp:Button>
</form>
</body>
</HTML>
```

---

**Note** The exact values for the positions and sizes of the controls in your own code might differ slightly from those shown in [Listing 15.1](#).

Let's examine the lines in this file. The first lines are

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
AutoEventWireup="false"
Inherits="MyWebApplication.WebForm1" %>
```

The `language` attribute indicates that the file uses the C# language. The `Codebehind` attribute specifies the code-behind file that supports the form, and in this case, the code-behind file is `WebForm1.aspx.cs`. The `AutoEventWireUp` attribute indicates whether the ASP.NET framework automatically calls the `Page_Init()` and `Page_Load()` event handler methods. These methods are defined in the `WebForm1.aspx.cs`; you'll learn more about these event handler methods shortly. The `Inherits` attribute

specifies the name of the class in the `WebForm1.aspx.cs` file from which the form inherits.

The next few lines are standard HTML that specifies the header and some meta-information describing the file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
<HEAD>
  <title>WebForm1</title>
  <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
  <meta content="C#" name="CODE_LANGUAGE">
  <meta content="JavaScript" name="vs_defaultClientScript">
  <meta content="http://schemas.microsoft.com/intellisense/ie5"
    name="vs_targetSchema">
</HEAD>
```

The next line starts the body of the file:

```
<body MS_POSITIONING="GridLayout">
```

The `MS_POSITIONING` attribute indicates that the form controls are laid out in a grid. The alternative to `GridLayout` is `LinearLayout`, which specifies that the form controls are to be laid out one after another in the browser.

The next line starts a form:

```
<form id="Form1" method="post" runat="server">
```

The `ID` attribute specifies that the name of the form is `Form1`. The `method` attribute indicates that the form uses an HTTP post request to send information to the server. The `runat` attribute specifies that the form is executed on the server.

The next lines contain the details of the `TextBox` control that you added to your form:

```
<asp:TextBox id="TextBox1" style="Z-INDEX: 101; LEFT: 13px;
  POSITION: absolute; TOP: 11px" runat="server"
  Width="386px" Height="212px"
  TextMode="MultiLine"></asp:TextBox>
```

The next lines contain the details of the `Button` control that you added to your form:

```
<asp:Button id="Button1" style="Z-INDEX: 102; LEFT: 17px;
  POSITION: absolute; TOP: 231px" runat="server"
  Width="82px" Height="22px" Text="Press Me!"></asp:Button>
```

The remaining lines in the `WebForm1.aspx` file end the form, the body, and the file:

```
  </form>
</body>
</HTML>
```

## The `WebForm1.aspx.cs` File

The `WebForm1.aspx.cs` file contains the code behind your form. You can view this code by selecting `View > Code`, or you can press F7 on your keyboard.

[Listing 15.2](#) shows the contents of the `WebForm1.aspx.cs` file.

### **Listing 15.2: `WebForm1.aspx.cs`**

---

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
```

```
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace MyWebApplication
{
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.TextBox TextBox1;

        protected System.Web.UI.WebControls.Button Button1;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            //
            // CODEGEN: This call is required by the ASP.NET Web Form Designer.
            //
            InitializeComponent();
            base.OnInit(e);
        }

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.Button1.Click += new System.EventHandler(this.Button1_Click);
            this.Load += new System.EventHandler(this.Page_Load);

        }
        #endregion

        private void Button1_Click(object sender, System.EventArgs e)
        {
            TextBox1.Text =
                "But, soft! what light through yonder window breaks?\n" +
                "It is the east, and Juliet is the sun.\n" +
                "Arise, fair sun, and kill the envious moon,\n" +
                "Who is already sick and pale with grief,\n" +
                "That thou her maid art far more fair than she";
        }
    }
}
```

---

As you can see, the `WebForm1` class is derived from the `System.Web.UI.Page` class. In fact, when you run your form, .NET

actually creates an object of the `Page` class that represents your form.

The `WebForm1` class declares two protected objects named `TextBox1` and `Button1`, which represent the `TextBox` and `Button` controls you added to your form.

The `Page_Load()` event handler method is called when the `Page_Load` event is raised. The `Page_Load` event is raised whenever the Web form is loaded by a browser. Typically, you'll place any initialization code in the `Page_Load()` method. For example, if you wanted to access a database, you would open the database connection in the `Page_Load()` method.

The `OnInit()` and `InitializeComponent()` methods are placed within `#region` and `#endregion` preprocessor directives. These directives enclose an area of code that may then be collapsed in VS .NET's code editor, leaving only the text that immediately follows `#region` visible.

The `OnInit()` method is called when the form is initialized. This method calls the `InitializeComponent()` method and adds the button `Click` and the form `Load` events to the `System.EventHandler` object. This informs the system that these two events are to be handled by the `Button1_Click()` and `Page_Load()` methods, respectively.

The `Button1_Click()` method is the method you modified earlier with code that sets the `Text` property of your `TextBox1` control to a string containing the quote from *Romeo and Juliet*.

In the [next section](#), you'll be introduced to some of the other controls you can add to a Web form.

**Team LiB**

◀ PREVIOUS

NEXT ▶

## The Web Form Controls

In this section, you'll see a summary of the various Web form controls that you can pick from the Toolbox's Web Forms Section. [Table 15.1](#) summarizes the controls.

**Table 15.1: WEB FORM CONTROLS**

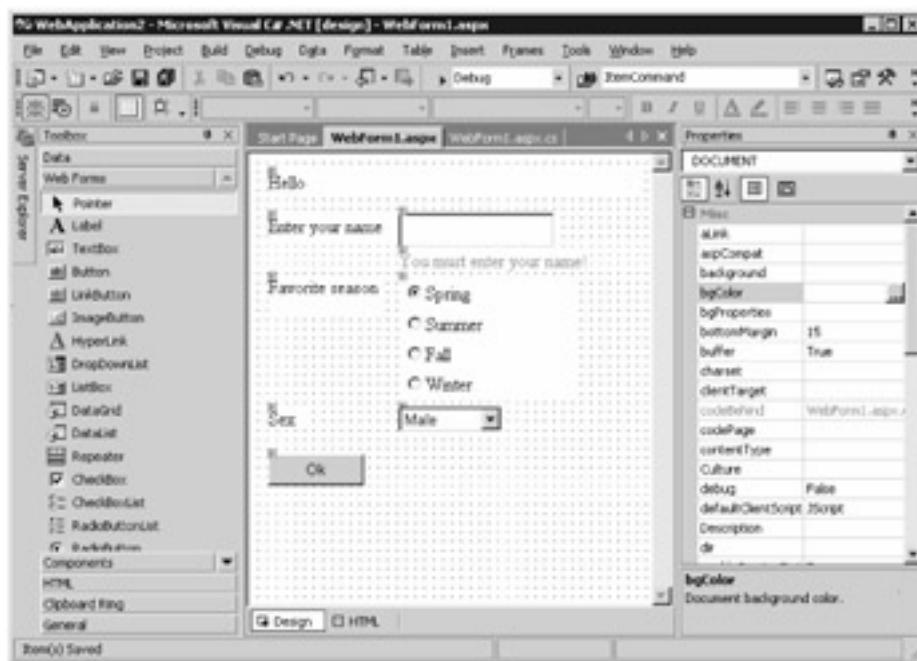
CONTROL	DESCRIPTION
Label	Displays text. You set the text that you want to display using the <code>Text</code> property.
TextBox	A box containing text that the user of your form may edit at runtime. The <code>TextMode</code> property may be set to <code>SingleLine</code> (text appears on one line), <code>MultiLine</code> (text appears over multiple lines), and <code>Password</code> (text appears as asterisk characters). The <code>Text</code> property contains the <code>TextBox</code> text.
Button	A clickable button. The <code>Text</code> property determines the text shown on the button.
LinkButton	Similar to a <code>Button</code> , except that a <code>LinkButton</code> appears as a hypertext link. You set the link using the <code>Text</code> property.
ImageButton	Similar to a <code>Button</code> , except that an <code>ImageButton</code> shows an image. You set the image using the <code>ImageUrl</code> property.
HyperLink	A hyperlink. You set the hyperlink using the <code>NavigateUrl</code> property.
DropDownList	A list of options that drops down when clicked. You set the list of options using the <code>Items</code> property. The user can select only one option from the <code>DropDownList</code> when the form is run.
ListBox	A list of options. You set the list of options using the <code>Items</code> property. The user can select multiple options from the <code>ListBox</code> if the <code>SelectionMode</code> property is set to <code>Multiple</code> . The other value is <code>Single</code> , in which case the user can select only one option.
GridView	A grid containing data retrieved from a data source, such as a database. You set the data source using the <code>DataSource</code> property.
DataSource	A list containing data retrieved from a data source. You set the data source using the <code>DataSource</code> property.
Repeater	A list containing data retrieved from a data source that you set using the <code>DataSource</code> property. Each item in the list may be displayed using a template. A template defines the content and layout of the items in the list.
CheckBox	A check box contains a Boolean <code>true/false</code> value that is set to <code>true</code> by the user if they check the box. The <code>Checked</code> property indicates the Boolean value currently set in the check box.
CheckBoxList	A multiple-selection check box. You set the list of check boxes using the <code>Items</code> property.
RadioButton	A radio button contains a Boolean <code>true/false</code> value that is set to <code>true</code> by the user if they press the button. The <code>Checked</code> property indicates the Boolean value currently set in the radio button.
RadioButtonList	A group of radio buttons. You set the list of radio buttons using the <code>Items</code> property.
Image	Displays an image that you set using the <code>ImageUrl</code> property.

Panel	A container for other controls.
PlaceHolder	A container for controls that you can create at runtime; these are known as <i>dynamic controls</i> .
Calendar	Displays a calendar for a month and allows a user to select a date and navigate to the previous and next month. You use the <code>SelectedDate</code> property to get or set the selected date, and you use the <code>VisibleDate</code> property to get or set the month currently displayed.
AdRotator	Displays banner advertisements. Details on the advertisements, such as the image, URL when clicked, and frequency of display, are set in an XML file that you set using the <code>AdvertisementFile</code> property.
Table	Displays a table of rows, which you set using the <code>Rows</code> property.
RequiredFieldValidator	Used to ensure that the user has specified some input for a control. You set the control to validate using the <code>ControlToValidate</code> property. You'll see an example that uses a validation control shortly.
CompareValidator	Used to compare an entry made by a user in one control with another control or a constant value. You set the control to validate using the <code>ControlToValidate</code> property (this control contains the value entered by the user). You set the control to compare against using the <code>ControlToCompare</code> property or the <code>ValueToCompare</code> property. You set the operator for the comparison using the <code>Operator</code> property.
RangeValidator	Used to ensure that the user has entered a value within a specified range in a control. You set the control to validate using the <code>ControlToValidate</code> property, and the range of values using the <code>MinimumValue</code> and <code>MaximumValue</code> properties.
RegularExpressionValidator	Used to ensure that the user has entered a value that satisfies a specified regular expression. You set the control to validate using the <code>ControlToValidate</code> property, and the regular expression using the <code>ValidationExpression</code> property.
CustomValidator	Used to perform your own custom validation for the value entered by a user. You set the control to validate using the <code>ControlToValidate</code> property, and your function to use in the validation using the <code>ClientValidationFunction</code> property.
ValidationSummary	Used to display a summary of all validation errors on the Web form and/or a message box. You set whether you want to show the errors on your Web form using the <code>ShowSummary</code> property and whether you want to show the errors in a message box using the <code>ShowMessageBox</code> property.
XML	Displays the contents of an XML file. You set the XML file to display using the <code>DocumentSource</code> property.
Literal	Displays static text. You set the text to display using the <code>Text</code> property.

You'll see how to use some of these controls in the rest of this chapter.

## Building a More Complex Application

In this section, you'll see a more complex Web form that uses `Label`, `TextBox`, `RadioButtonList`, `DropDownList`, and `RequiredFieldValidator` controls. The form will prompt the user for their name (a required field), favorite season (spring, summer, fall, or winter), and gender (male or female). The form will also feature a `Button` control, which when pressed will set the `Text` property of one of the `Label` controls to a string containing the user's name, gender, and favorite season. [Figure 15.4](#) shows how your final form will appear.



**Figure 15.4:** The appearance of the final form

Perform the following steps:

1. To create the new project, select File > New Project in VS .NET. Select Visual C# Projects from the Project Types area on the left of the New Project dialog box, and select ASP .NET Web Application from the Templates area on the right. Enter **http://localhost/MyWeb-Application2** in the Location field. VS .NET will display a blank form to which you can add controls.
2. Now, add the four `Label` controls listed in [Table 15.2](#) to your blank form. This table shows the `ID` and `Text` property to set for each of your `Label` controls.

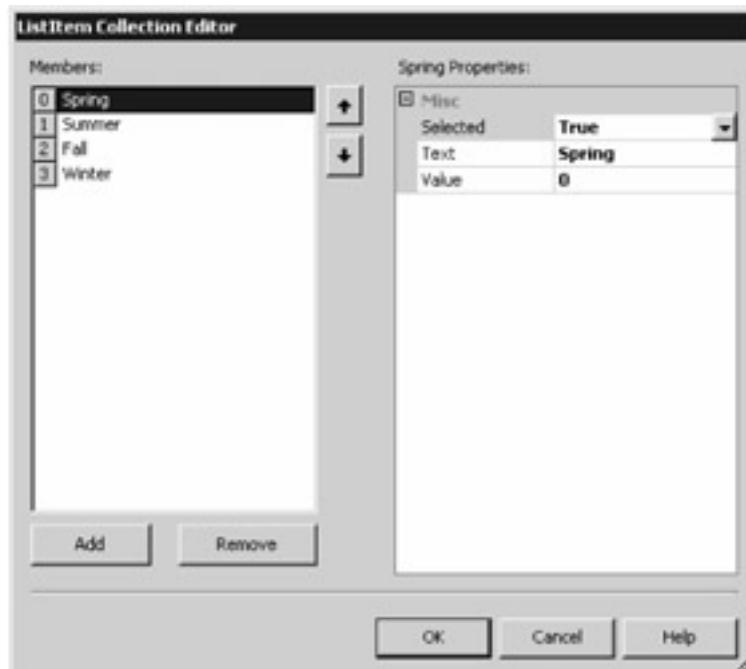
**Table 15.2: Label CONTROLS**

ID PROPERTY	Text PROPERTY
HelloLabel	Hello
NameLabel	Enter your name
SeasonLabel	Favorite season
SexLabel	Sex

3. Next, add a `TextBox` control to the right of `NameLabel`. Set the `ID` property for your `TextBox` control to `NameTextBox`. The user will enter their name in `NameTextBox` when the form is run.
4. We want the user to have to enter their name; if they don't, we want to display a message prompting them to do so. To achieve this, you use a `RequiredFieldValidator` control. Add a `RequiredFieldValidator` control below `NameTextBox`. Set the `ID` property for your `RequiredFieldValidator` control to `NameRequiredFieldValidator`. Set the `Text` property to `You must enter your name!`. Finally, set the `ControlToValidate` property to

NameTextBox.

5. Next, add a RadioButtonList control to the right of SeasonLabel1. The user will select their favorite season from this control. Set the ID property for your RadioButtonList control to SeasonRadioButtonList. To add radio buttons to SeasonRadioButtonList, click the ellipsis (...) button in the Items property. This displays the ListItem Collection Editor, which you use to add, modify, or remove items in the Items collection for the control. When the form is run, any items you add to the collection are displayed as radio buttons. [Figure 15.5](#) shows the ListItem Collection Editor with the required entries for your form.



**Figure 15.5:** The ListItem Collection Editor

6. The Selected property indicates whether the item is initially selected in the running form. The Text property contains the text displayed with the item. The Value property is the returned value when the item is selected.
7. Now click the Add button to add the first item to your RadioButtonList control. Set the Selected property for the item to True—this causes the radio button to be initially selected. Set the Text property for the item to Spring; this is the text displayed in the radio button. Set the Value property to 0; this is the actual value selected. [Table 15.3](#) shows the Selected, Text, and Value properties for this radio button, along with the three other radio buttons to add to your RadioButtonList control.

**Table 15.3: RadioButtonList ITEMS**

Selected PROPERTY	Text PROPERTY	Value PROPERTY
True	Spring	0
False	Summer	1
False	Fall	2
False	Winter	3

8. Next, add a DropDownList control to your form. This control will allow a user to select their gender (male or female). Set the ID property for your DropDownList control to SexDropDown-List. You add items to a DropDownList control using the ListItem Collection Editor, which you access using the ellipsis button through the Items property. Open the ListItem Collection Editor and add the items shown in [Table 15.4](#).

**Table 15.4: DropDownList ITEMS**

Selected PROPERTY	Text PROPERTY	Value PROPERTY
True	Male	0
False	Female	1

9. Finally, add a Button control to your form. Set the ID property for your Button control to OkButton, and set the Text property to Ok. Double-click OkButton to edit the code for the OkButton\_Click() method, and add the following lines of code to this method:

```
HelloLabel.Text =
    "Hello " + NameTextBox.Text +
    ", you are " + SexDropDownList.SelectedItem.Text +
    "and your favorite season is " +
    SeasonRadioButtonList.SelectedItem.Text;
```

As you can see, this line sets the Text property for the HelloLabel control to a string containing the user's entry in the NameTextBox, SexDropDownList, and SeasonRadioButton controls.

Run your completed form by pressing Ctrl+F5. Press the OK button without entering a name, and you'll see the message "You must enter your name!", as shown in [Figure 15.6](#). This message comes from the NameRequiredFieldValidator control.



**Figure 15.6:** Message from the NameRequired-FieldValidator control

When you've finished running your form, close it and return to the VS .NET form designer. You can view the HTML containing the ASP.NET tags for your form by clicking the HTML link at the bottom of the form designer. Click the HTML link to view the code for your form. [Listing 15.3](#) shows the WebForm1.aspx file for the form. You'll notice that this file contains the various controls that were added to the form.

### **Listing 15.3: THE WebForm1.aspx FILE**

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
  AutoEventWireup="false"
  Inherits="WebApplication2.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>WebForm1</title>
```

```
<meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
<meta name="CODE_LANGUAGE" Content="C#">
<meta name="vs_defaultClientScript" content="JavaScript">
<meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
</HEAD>
<body MS_POSITIONING="GridLayout">
  <form id="Form1" method="post" runat="server">
    <asp:Label id="HelloLabel" style="Z-INDEX: 101; LEFT: 17px;
      POSITION: absolute; TOP: 16px" runat="server"
      Width="322px" Height="23px">Hello</asp:Label>
    <asp:Label id="NameLabel" style="Z-INDEX: 102; LEFT: 17px;
      POSITION: absolute; TOP: 54px" runat="server"
      Width="114px" Height="22px">Enter your name</asp:Label>
    <asp:Label id="SeasonLabel" style="Z-INDEX: 103; LEFT: 17px;
      POSITION: absolute; TOP: 107px" runat="server"
      Width="101px" Height="32px">Favorite season</asp:Label>
    <asp:Label id="SexLabel" style="Z-INDEX: 104; LEFT: 17px;
      POSITION: absolute; TOP: 221px" runat="server"
      Width="33px" Height="15px">Sex</asp:Label>
    <asp:TextBox id="NameTextBox" style="Z-INDEX: 105; LEFT: 130px;
      POSITION: absolute; TOP: 51px" runat="server"
      Width="135px" Height="30px"></asp:TextBox>
    <asp:RequiredFieldValidator id="NameRequiredFieldValidator"
      style="Z-INDEX: 106; LEFT: 130px; POSITION: absolute;
      TOP: 84px" runat="server" ErrorMessage="RequiredFieldValidator"
      ControlToValidate="NameTextBox">You must enter your name!
    </asp:RequiredFieldValidator>
    <asp:RadioButtonList id="SeasonRadioButtonList"
      style="Z-INDEX: 107; LEFT: 130px; POSITION: absolute;
      TOP: 107px" runat="server" Width="152px" Height="107px">
      <asp:ListItem Value="0" Selected="True">Spring</asp:ListItem>
      <asp:ListItem Value="1">Summer</asp:ListItem>
      <asp:ListItem Value="2">Fall</asp:ListItem>
      <asp:ListItem Value="3">Winter</asp:ListItem>
    </asp:RadioButtonList>
    <asp:DropDownList id="SexDropDownList" style="Z-INDEX: 108;
      LEFT: 130px; POSITION: absolute; TOP: 220px" runat="server"
      Width="90px" Height="27px">
      <asp:ListItem Value="0" Selected="True">Male</asp:ListItem>
      <asp:ListItem Value="1">Female</asp:ListItem>
    </asp:DropDownList>
    <asp:Button id="OkButton" style="Z-INDEX: 109; LEFT: 17px;
      POSITION: absolute; TOP: 261px" runat="server"
      Width="83px" Height="27px" Text="Ok"></asp:Button>
  </form>
</body>
</HTML>
```

---

The WebForm1.aspx.cs file contains the code behind your form. You can view this code by selecting View > Code, or you can press F7 on your keyboard.

## Using a *DataGrid* Control to Access a Database

A *DataGrid* allows you to access rows in a database table. In the following sections, you'll learn how to create an ASP.NET Web application that uses a *DataGrid* control to access the rows in a database table. The *DataGrid* you'll create will display the rows from the *Products* table of the Northwind database.

### Creating the Web Application

Perform the following steps:

1. To create the new project, select File > New Project in VS .NET. Select Visual C# Projects from the Project Types area on the left of the New Project dialog box, and select ASP.NET Web Application from the Templates area on the right. Enter **http://localhost/DataGrid-WebApplication** in the Location field. Click OK to continue. Your new project will contain a blank form.
2. Next, you'll add a *DataGrid* control to your form. To do this, select the *DataGrid* from the Toolbox and drag it to your form. [Figure 15.7](#) shows the form with the *DataGrid*.



**Figure 15.7:** Form with a *DataGrid*

3. Next, you'll add a *SqlConnection* object and a *SqlDataAdapter* object to your form. To add these objects, select the *Products* table in Server Explorer and drag it to your form. (Adding a *SqlConnection* object to a form was discussed in [Chapter 6](#), "Introducing Windows Applications and ADO.NET," and in [Chapter 7](#), "Connecting to a Database.")

**Note** To display Server Explorer, select View > Server Explorer, or press Ctrl+Alt+S on your keyboard.

4. After you drag the *Products* table to your form, VS .NET creates a *SqlConnection* object named *sqlConnection1* and a *SqlDataAdapter* object named *sqlDataAdapter1*. Click your *sqlConnection1* object to display the properties for this object in the Properties window. To enable *sqlConnection1* to access the database, you need to set the password for the connection. To do this, you need to add a substring containing *pwd* to the *ConnectionString* property of *sqlConnection1*. Add *pwd=sa;* to the *ConnectionString* property.

**Note** If you don't have the password for the *sa* user, you'll need to get it from your database administrator.

5. Next, you'll modify the SQL SELECT statement used to retrieve the rows from the *Products* table. Click the *sqlDataAdapter1* object to display the properties for this object. Click the addition icon to the left of the *SelectCommand* property to display the dynamic properties. One of the dynamic properties is the *CommandText* property, which contains the

SELECT statement.

6. Click **CommandText** and then click the ellipsis button to display the Query Builder. You use Query Builder to define SQL statements. You can type in the SQL statement, or you can build it visually. Uncheck all the columns **except** the following: **ProductID**, **ProductName**, **QuantityPerUnit**, and **UnitPrice**. This results in the SQL SELECT statement being set to the following:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice
FROM Products
```

7. Click the OK button to save your SELECT statement and close Query Builder.

Next, you need to create a **DataSet** object. You use a **DataSet** object to store a local copy of the information stored in the database. A **DataSet** object can represent database structures such as tables, rows, and columns. In the example in this section, you'll use a **DataSet** object to store the rows from the **Products** table.

1. Click an area of your form outside the **DataGrid**. Next, click the Generate Dataset link near the bottom of the Properties window. This displays the Generate Dataset dialog box. Select the New radio button and make sure the text field to the right of this radio button contains **DataSet1**. Also, make sure the Add This Dataset To The Designer checkbox is checked. Click the OK button to continue. This adds a new **DataSet** object named **dataSet11** to your form.
2. Next, you'll need to set the **DataSource** property of your **DataGrid** to your **DataSet** object. This sets the source of the data for your **DataGrid** and allows the rows from your **DataSet** to be displayed in your **DataGrid**. To set the **DataSource** property, click your **DataGrid** object and set the **DataSource** property to **dataSet11**. Also, set the **DataMember** property to **Products**; this is the table with rows that are to be displayed by your **DataGrid**.
3. Next, you'll need to add code to populate **sqlDataAdapter1** with the rows retrieved by your **SELECT** statement. Typically, the best place to place this code is in the **Page\_Load( )** method of your form. The **Page\_Load( )** method is called when the Web page containing your form is initially loaded or refreshed. The **IsPostBack** property of a page is **false** the first time the page is loaded and **true** when the submit button of a form is pressed. For performance, you'll generally want to retrieve rows only when the **IsPostBack** property is **false**; otherwise you might needlessly reload the rows from the database. To view the code for your form, open the code for your form by selecting **View > Code** or by pressing **F7** on your keyboard. Set your **Page\_Load( )** method to the following:

```
private void Page_Load(object sender, System.EventArgs e)
{
    // Put user code to initialize the page here
    if (!this.IsPostBack)
    {
        sqlDataAdapter1.Fill(dataSet11, "Products");
        this.DataBind();
    }
}
```

The **Fill( )** method retrieves the rows from the **Products** table and populates **dataSet11** with those rows. The **.DataBind( )** method then fills the **Products** **DataTable** in **dataSet11** with the rows retrieved from the **Products** table. This causes the rows to be displayed in the **DataGrid** of your form.

To run your form, select **Debug > Start Without Debugging**, or press **Ctrl+F5** on your keyboard (see [Figure 15.8](#)).

The screenshot shows a Microsoft Internet Explorer window titled "WebForm1 - Microsoft Internet Explorer". The address bar displays "http://localhost/DataGridWebApplication/WebForm1.aspx". The main content area contains a DataGrid control with the following data:

ProductID	ProductName	QuantityPerUnit	UnitPrice
1	Chai	10 boxes x 20 bags	18
2	Chang	24 - 12 oz bottles	19
3	Aniseed Syrup	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	36 boxes	21.35
6	Grandma's Boysenberry Spread	12 - 8 oz jars	25
7	Uncle Bob's Organic Dried Pears	12 - 1 lb pkgs.	30
8	Northwoods Cranberry Sauce	12 - 12 oz jars	40
9	Mishi Kobe Niku	18 - 500 g pkgs.	97
10	Ibura	12 - 200 ml jars	31
11	Queso Cabrales	1 kg pkg.	21

**Figure 15.8:** The running form

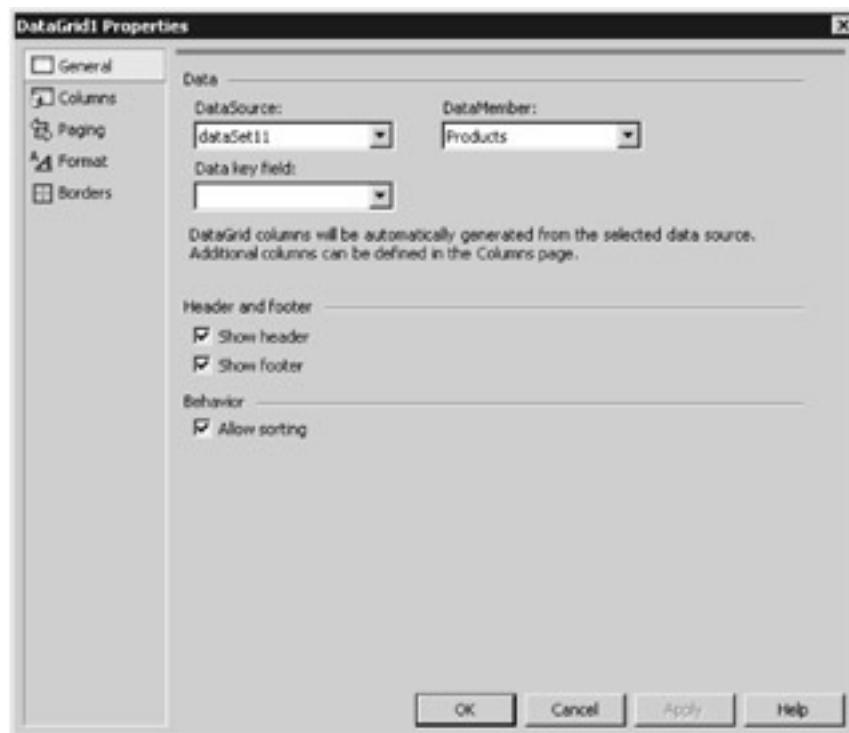
As you can see, a vertical scroll bar is displayed because of the number of rows retrieved from the `Products` table. In the [next section](#), you'll learn how to customize your `DataGrid`. You'll see how you can control the number of rows displayed in your `DataGrid` so that no scroll bar appears, as well as control other aspects of your `DataGrid`.

## Customizing the *DataGrid*

You customize your `DataGrid` by first selecting the `DataGrid` control and then clicking the Property Builder link at the bottom of the Properties window. This displays the Properties dialog box for your `DataGrid`. The Properties dialog box is divided into five areas: General, Columns, Paging, Format, and Borders.

### General Properties

You use the General properties to set the data source for your `DataGrid` and whether you want a header and footer to be displayed, among other properties. Set your General properties as shown in [Figure 15.9](#).



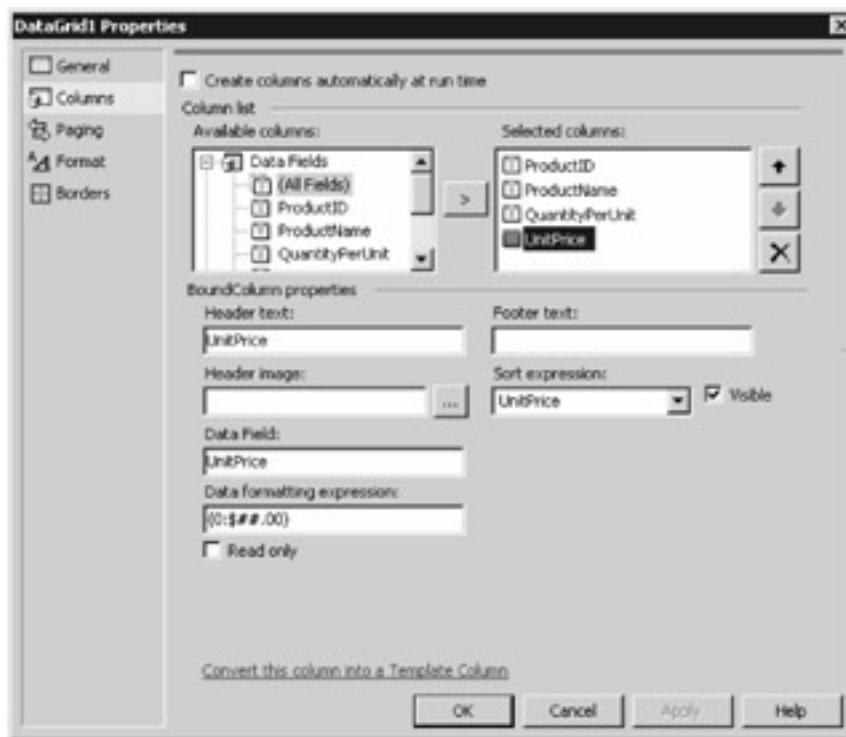
**Figure 15.9:** The General properties

The General properties are as follows:

- **DataSource** The `DataSource` is the source of the data for your `DataGrid`. In this example, the `DataSource` is `dataSet11`.
- **DataMember** The `DataMember` is the name of the table to which your `DataGrid` is bound. In this example, the `DataMember` is `Products`.
- **Data Key Field** The Data Key Field is the name of a column or expression that is associated with each row in your `DataGrid` but isn't actually shown. You typically use it to specify the primary key.
- **Header and Footer** The header displays the name of the columns at the top of the `DataGrid`. Select Show Header and Show Footer.
- **Behavior** You can sort columns in the header of your `DataGrid`. Select Allow Sorting so that your columns can be sorted.

## Columns Properties

You use the Columns properties to select the columns to be displayed in your `DataGrid` and the header and footer text to be displayed for each column, among other properties. Click the Columns link of the Properties dialog box and set your Columns properties as shown in [Figure 15.10](#).



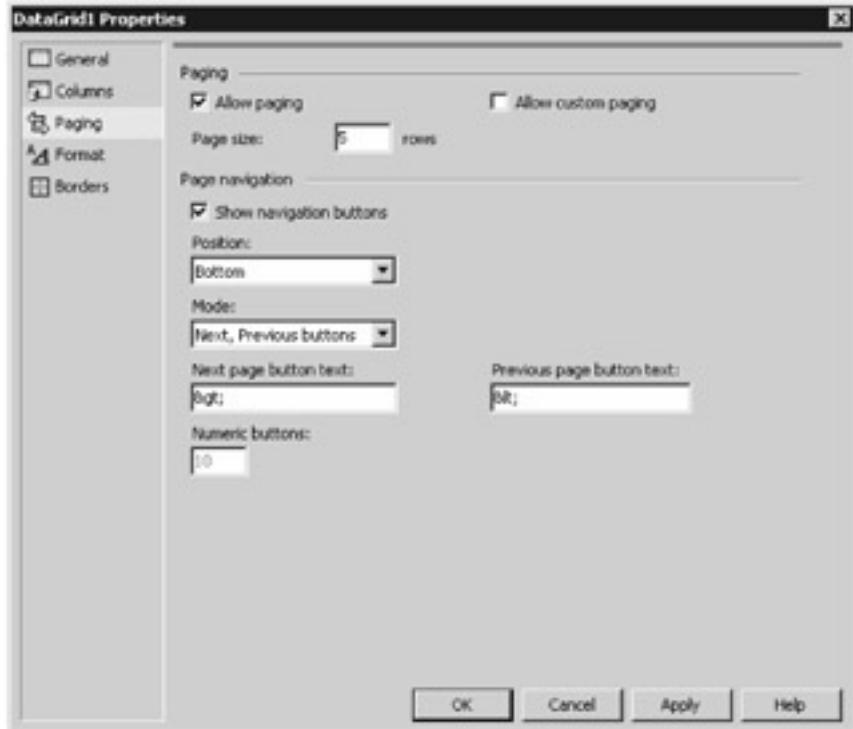
**Figure 15.10:** The Columns properties

The Columns properties are as follows:

- **Create Columns Automatically At Run Time** The Create Columns Automatically At Run Time check box specifies whether to automatically include all the columns for the DataSet in your DataGrid. When this check box is unselected, you can then set the other properties for each column individually. Unselect this check box.
- **Column List** The Column List allows you to select columns from your DataSet for display in your DataGrid. You select columns from the Available Columns area on the left and add them to Selected Columns area on the right using the button containing the right arrow. Select (All Fields) from Available Columns, and add them to the Selected Columns.
- **BoundColumn Properties** The BoundColumn properties allow you to set the properties for each column. You select the column you want to set in the Selected Columns area, and then you set the properties for that column. The fields you can set for each column are as follows:
  - **Header Text** The text you want to display in the header for a column.
  - **Footer Text** The text you want to display in the footer for a column.
  - **Header Image** The image you want to display in the header for a column.
  - **Sort Expression** The column or expression you want to use to sort the column by. Select `UnitPrice` as the Sort expression.
  - **Data Field** The name of column.
  - **Data Formatting Expression** Allows you to format a column value. You can use a formatting expression to format dates and numbers, among others. For example, `{0:$##.00}` formats a number, adds a dollar sign at the front, and displays two digits after the decimal point; thus, 19 is formatted as \$19 . 00. Set the formatting expression for the `UnitPrice` column to `{0:$##.00}`.

## Paging Properties

Next, click on the Paging link of the Properties dialog box. Normally, all the rows retrieved by a SELECT statement are displayed on a single page for the DataGrid. You can use the Paging properties to split all the rows into separate pages, with a fixed number of rows on each page in your DataGrid. You can then select the buttons to navigate between these pages of rows. You'll set your page size to five rows with Next and Previous buttons to navigate between the pages of rows. Set your Paging properties as shown in [Figure 15.11](#).



**Figure 15.11:** The Paging properties

The Paging properties are as follows:

- **Allow Paging** Controls whether paging is enabled. Check the Allow Paging box.
- **Page Size** Controls the number of rows displayed on each page. Set your Page Size to 5.
- **Show Navigation Buttons** The Show Navigation Buttons check box controls whether navigation buttons are displayed. These buttons allow you to navigate between pages of rows. Check the Show Navigation Buttons box.
- **Position** Allows you to set the position of the navigation buttons. Set the Position to Bottom.
- **Mode** Controls the type of navigation buttons displayed. You can use Next and Previous buttons or page numbers to navigate between pages. Set the Mode to Next, Previous Buttons.
- **Next Page Button Text** Sets the text displayed on the Next page button. Leave this as &gt; so that a greater-than character (>) is displayed.
- **Previous Page Button Text** Sets the text displayed on the Previous page button. Leave this as &lt; so that a less-than character (<) is displayed.
- **Numeric Buttons** Control whether numbers are displayed for each page when you set the Mode to Page Numbers. For example, 1 navigates to the first page, 2 to the second page, and so on.

In addition to enabling paging, you'll also need to add some code to your `DataGrid` to make navigation work, and you'll do this shortly.

## Format Properties

Next, click the Format link of the Properties dialog box. You use the Format properties to control how each element on your `DataGrid` appears. You can set features such as the color of your `DataGrid`, as well as the font. You can also set the display properties of each column. You'll set the foreground and background color to black and white, respectively. You'll also set the font of the text displayed in your `DataGrid` to Arial. Set your Format properties as shown in [Figure 15.12](#).



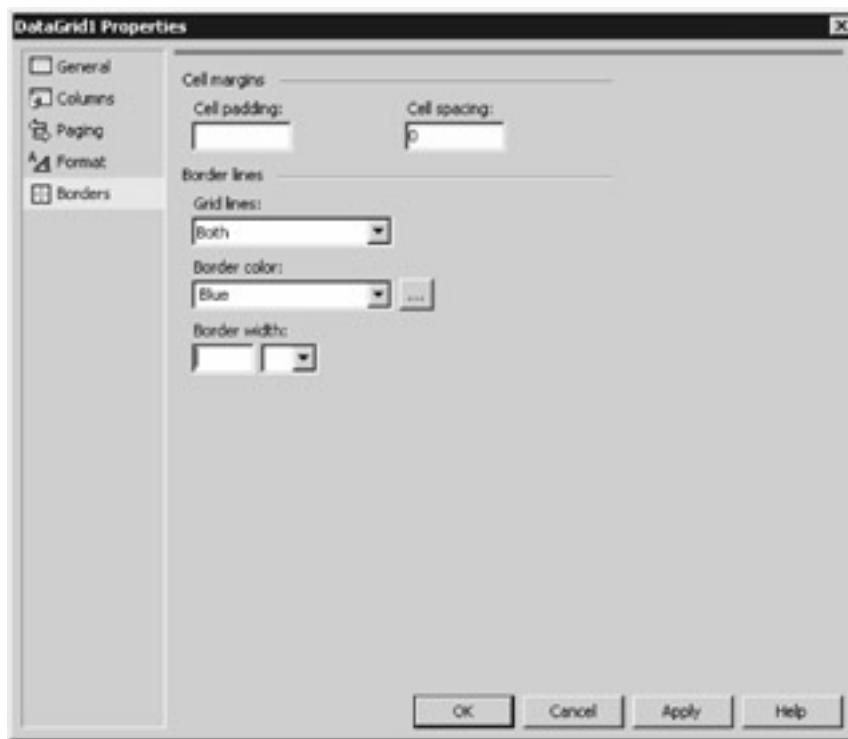
**Figure 15.12:** The Format properties

The Format properties are as follows:

- **Forecolor** The Forecolor option specifies the text color. Set the Forecolor to Black.
- **Back Color** The Back Color option specifies the color behind the text. Set the Back Color to White.
- **Font Name** The Font Name option specifies the font used to display the text. Set the Font Name to Arial.
- **Font Size** The Font Size option controls the size of the font used to display the text.
- **Bold, Italic, Underline, Strikeout, Overline** The Bold, Italic, Underline, Strikeout, and Overline options control the character formatting for the text.
- **Horizontal Alignment** The Horizontal Alignment option specifies the position of the text in the cell.

## Borders Properties

Next, click on the Borders link of the Properties dialog box. You use the Borders properties to control the padding, spacing, and appearance of the grid lines in your DataGrid. You'll set the border color of the grid lines in your DataGrid to blue. Set your Borders properties as shown in [Figure 15.13](#).



**Figure 15.13:** The Borders properties

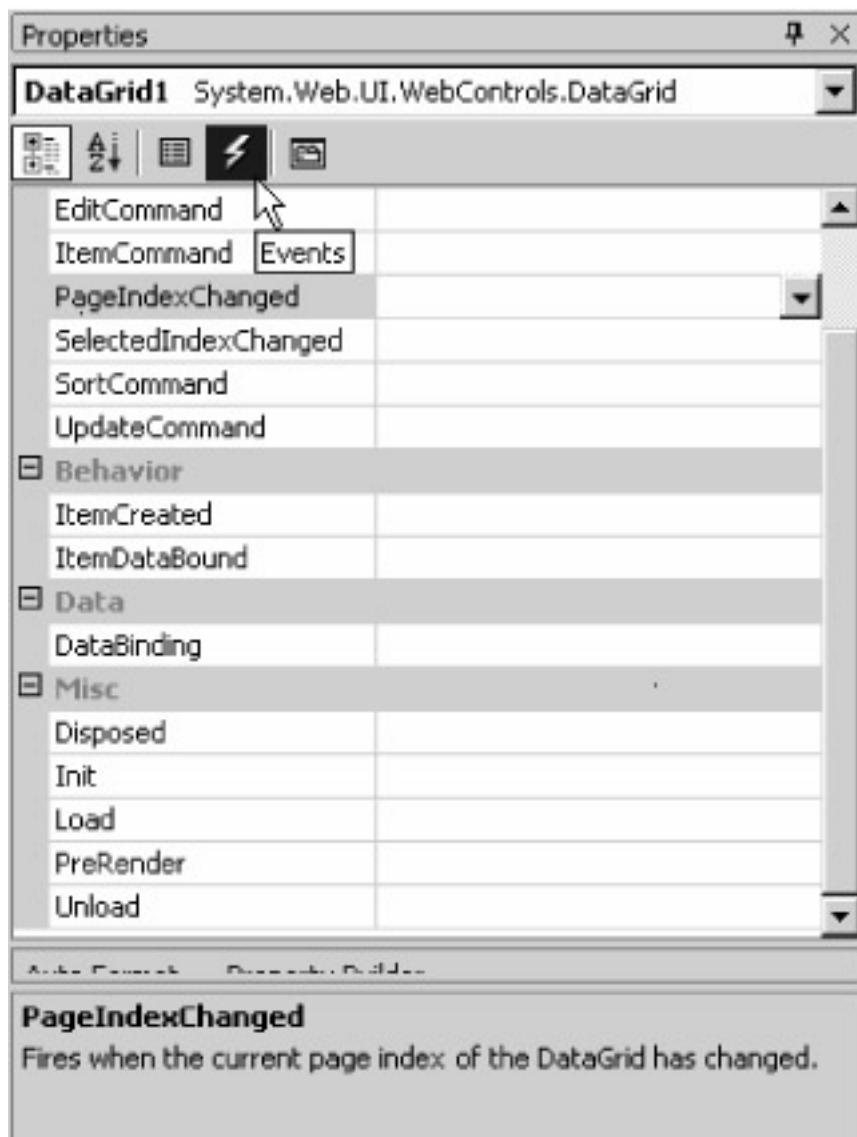
The Borders properties are as follows:

- **Cell Padding** Controls the amount of space (in pixels) between the edge of a cell and the cell contents in your DataGrid.
- **Cell Spacing** Controls the amount of space (in pixels) between each element in your DataGrid.
- **Grid Lines** Specifies the direction of the grid lines in your DataGrid.
- **Border Color** Specifies the color of the grid lines in your DataGrid. Set this to blue.
- **Border Width** Controls the width and units of the grid lines in your DataGrid.

Once you've set your properties, click the OK button to continue. Next, you'll code the `PageIndexChanged()` event handler to allow navigation of the rows in your DataGrid.

## Coding the `PageIndexChanged()` Event Handler

As mentioned earlier, in addition to enabling paging in the Paging properties window, you'll also need to add some code to your DataGrid, specifically, to the `PageIndexChanged()` event handler method. This method is called whenever you change the page in the DataGrid on your running Web page. Before you add the required code, you first select your DataGrid, and then you click the Events button in the Properties window to display the events for your DataGrid, as shown in [Figure 15.14](#).



**Figure 15.14:** Displaying the DataGrid events

Double-click the `PageIndexChanged` event and set your `DataGrid1_PageIndexChanged()` method as follows:

```
private void DataGrid1_PageIndexChanged(
    object source,
    System.Web.UI.WebControls.DataGridPageChangedEventArgs e)
{
    DataGrid1.CurrentPageIndex = e.NewPageIndex;
    sqlDataAdapter1.Fill(dataSet11, "Products");
    DataGrid1.DataBind();
}
```

The first statement inside the method body is as follows:

```
DataGrid1.CurrentPageIndex = e.NewPageIndex;
```

This statement sets the current page displayed in `DataGrid1` to the new page that is selected using the navigation buttons in the running form. You set the current page for `DataGrid1` using the `CurrentPageIndex` property, and you get the new page from the `NewPageIndex` property of the `DataGridPageChangedEventArgs` object. By setting `DataGrid1.CurrentPageIndex` equal to `e.NewPageIndex`, the navigation to the new page of rows is performed.

The second statement is as follows:

```
sqlDataAdapter1.Fill(dataSet11, "Products");
```

This statement calls the `Fill()` method of `sqlDataAdapter1` to populate `dataSet11` with the next set of rows from the `Products` table.

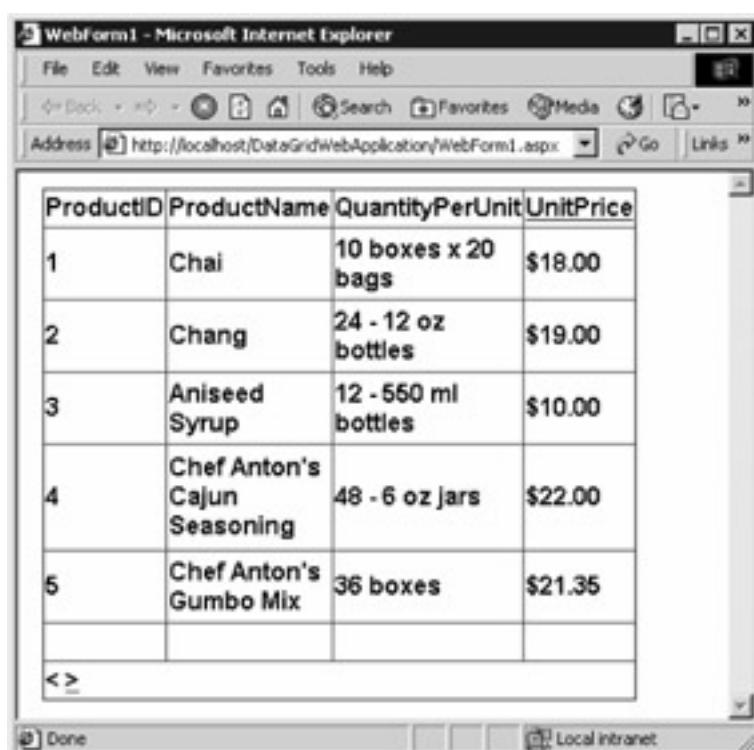
The third statement is as follows:

```
DataGrid1.DataBind();
```

This statement calls the `.DataBind()` method of `DataGrid1`, causing the new set of rows to be displayed.

**Note** With VS .NET, you can also go to the code view and use the top drop-down menus to create the signature for events. This applies to any events that you add. Of course, double-clicking on events to get the "default" event is easier, but there are other events for each control.

Run your form by pressing **Ctrl+F5** on your keyboard. [Figure 15.15](#) shows the running form.



The screenshot shows a Microsoft Internet Explorer window titled "WebForm1 - Microsoft Internet Explorer". The address bar displays "http://localhost/DataGridWebApplication/WebForm1.aspx". The main content area contains a DataGrid control with the following data:

ProductID	ProductName	QuantityPerUnit	UnitPrice
1	Chai	10 boxes x 20 bags	\$18.00
2	Chang	24 - 12 oz bottles	\$19.00
3	Aniseed Syrup	12 - 550 ml bottles	\$10.00
4	Chef Anton's Cajun Seasoning	48 - 6 oz jars	\$22.00
5	Chef Anton's Gumbo Mix	36 boxes	\$21.35
<>			

**Figure 15.15:** The running form

Use the navigation buttons to move between pages of rows. Once you've finished running your form, close it and return to the VS .NET form designer. Click the HTML link to view the code for your form. [Listing 15.4](#) shows the `WebForm1.aspx` file for the form. You'll notice that this file contains a `DataGrid` control with the appropriate columns.

#### **Listing 15.4: THE WebForm1.aspx FILE**

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
AutoEventWireup="false"
Inherits="DataGridWebApplication.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
<HEAD>
<title>WebForm1</title>
<meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
<meta content="C#" name="CODE_LANGUAGE">
<meta content="JavaScript" name="vs_defaultClientScript">
<meta content="http://schemas.microsoft.com/intellisense/ie5"
name="vs_targetSchema">
</HEAD>
<body MS_POSITIONING="GridLayout">
<form id="Form1" method="post" runat="server">
<asp:datagrid id=DataGrid1 style="Z-INDEX: 101; LEFT: 16px;
```

```
POSITION: absolute; TOP: 11px" runat="server"
AutoGenerateColumns="False" BorderColor="Blue" Font-Bold="True"
Font-Names="Arial" ForeColor="Black" BackColor="White"
AllowPaging="True" PageSize="5" ShowFooter="True"
DataMember="Products" AllowSorting="True"
DataSource="<%# dataSet11 %>" Height="333px" Width="352px">
<Columns>
<asp:BoundColumn
DataField="ProductID" HeaderText="ProductID">
</asp:BoundColumn>
<asp:BoundColumn DataField="ProductName"
HeaderText="ProductName">
</asp:BoundColumn>
<asp:BoundColumn DataField="QuantityPerUnit"
HeaderText="QuantityPerUnit">
</asp:BoundColumn>
<asp:BoundColumn DataField="UnitPrice"
SortExpression="UnitPrice" HeaderText="UnitPrice"
DataFormatString="{0:$##.00}">
</asp:BoundColumn>
</Columns>
</asp:datagrid></form>
</body>
</HTML>
```

---

In the [next section](#), you'll learn how to use a DataList control to access a database.

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Using a *DataList* Control to Access a Database

In this section, you'll learn how to use a *DataList* control to access the rows in the *Products* table.

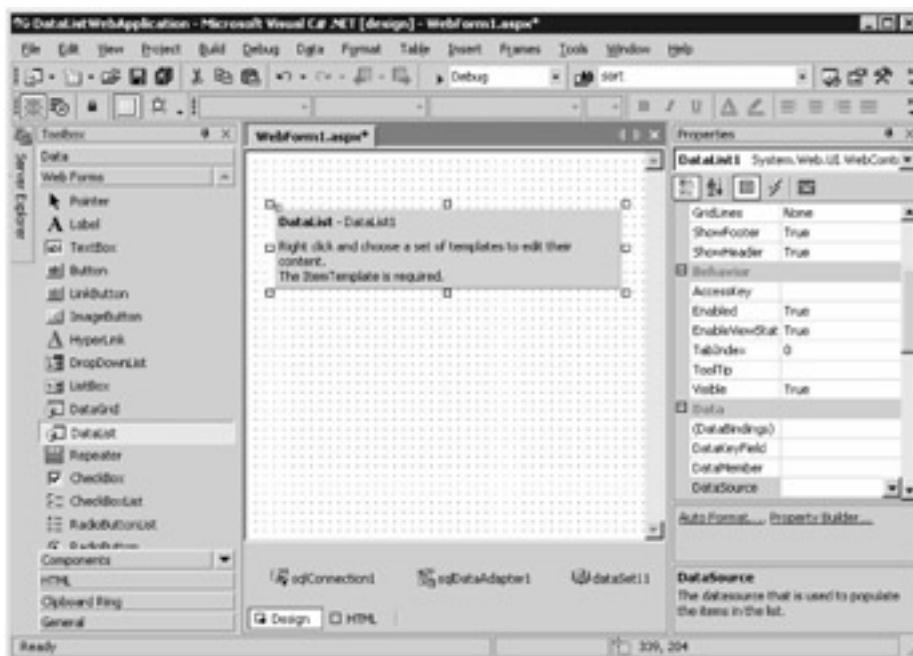
**Tip** A *DataList* offers you a lot more flexibility in the presentation of column values than that offered by a *DataGridView*.

Perform the following steps:

1. To create the new project, select File > New Project in VS .NET. Select Visual C# Projects from the Project Types area on the left of the New Project dialog box, and select ASP .NET Web Application from the Templates area on the right. Enter **http://localhost/DataList-WebApplication** in the Location field. Click OK to continue. Your new project will contain a blank form.
2. Next, you'll add a *SqlConnection* object and a *SqlDataAdapter* object to your form. Select the *Products* table in Server Explorer and drag it to your form.
3. After you drag the *Products* table to your form, VS .NET creates a *SqlConnection* object named *sqlConnection1* and a *SqlDataAdapter* object named *sqlDataAdapter1*.
4. Click your *sqlConnection1* object to display the properties for this object in the Properties window. To enable *sqlConnection1* to access the database, you need to set the password for the connection. Add *pwd=sa*; to the *ConnectionString* property.
5. Next, you'll modify the SQL *SELECT* statement used to retrieve the rows from the *Products* table. Click your *sqlDataAdapter1* object to display the properties for this object. Click the addition icon to the left of the *SelectCommand* property to display the dynamic properties; one of the dynamic properties is the *CommandText* property, which contains the *SELECT* statement. Next, click *CommandText* and then click the ellipsis button to display the Query Builder. You can type in the SQL statement, or you can build it up visually. Uncheck all the columns except *ProductID*, *ProductName*, *QuantityPerUnit*, and *UnitPrice*. This results in the SQL *SELECT* statement being set to the following:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice  
FROM Products
```

6. Click OK to continue.
7. Next, you need to create a *DataSet* object. Click an area of your form. Next, click the Generate-Dataset link near the bottom of the Properties window. This displays the Generate Dataset dialog box. Select the New radio button and make sure the field to the right of this radio button contains *DataSet1*. Click the OK button to continue. This adds a new *DataSet* object named *dataSet1* to your form.
8. Next, you'll add a *DataList* control to your form. To do this, select the *DataList* from the Toolbox and drag it to your form. [Figure 15.16](#) shows the form with the new *DataList*.



**Figure 15.16:** Form with a DataList

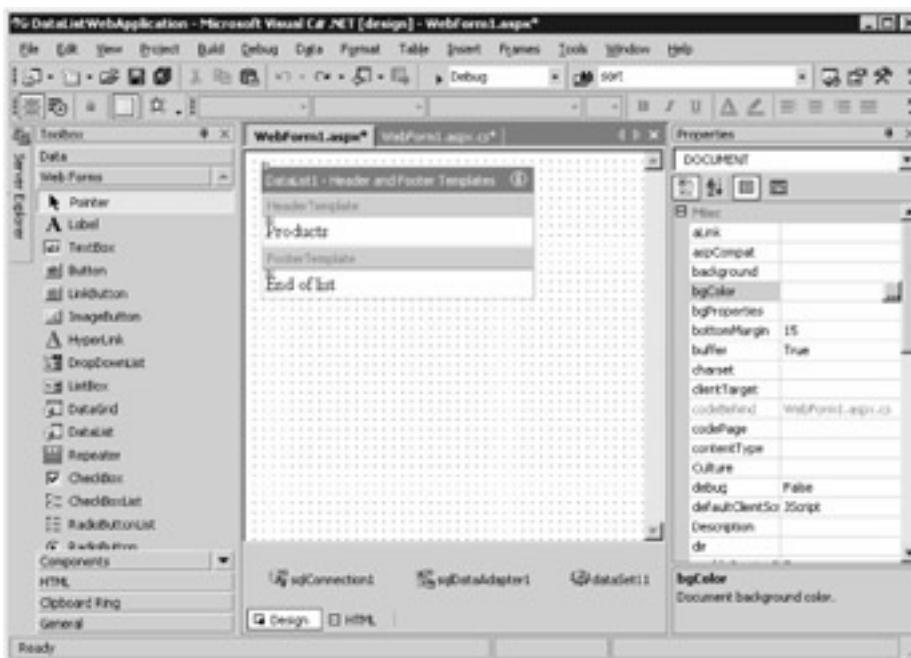
9. Next, you'll need to set the `DataSource` property of your `DataList` to your `DataSet` object created earlier. This sets the source of the data for your `DataList` and allows the rows from your `DataSet` to be displayed in your `DataList`. To set the `DataSource` property, click your `DataList` object and set its `DataSource` property to `dataSet11`. Also, set the `DataMember` property of your `DataList` to `Products`; this is the table with rows that are displayed by the `DataList`.

A `DataList` uses templates that define how its contents are laid out, and your next task is to set up those templates.

**Tip** It is the `DataList` templates that give you the flexibility for laying out controls that display column values.

You'll edit the template that defines the header and footer for the `DataList`, along with the template that defines the actual items displayed within your `DataList`:

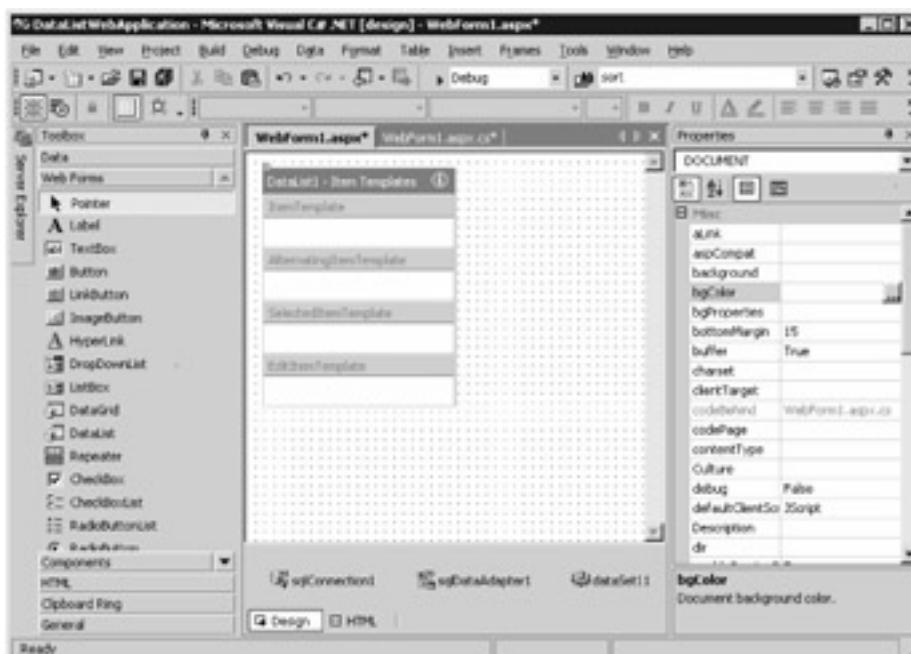
1. To edit the header and footer template, right-click your `DataList` and select `Edit Template > Header And Footer Templates`.
2. You can add controls to the areas within the `HeaderTemplate` and `FooterTemplate` areas. Any controls you add will be displayed at the start and end of the `DataList`, respectively. Add a `Label` in `HeaderTemplate`; you do this by dragging a `Label` control from the `Toolbox` to the empty area below `HeaderTemplate`. Set the `Text` property for this `Label` to `Products`. Also, add a `Label` in the `FooterTemplate` area and set its `Text` property to `End of list`. [Figure 15.17](#) shows the modified header and footer templates with the `Label` controls.



**Figure 15.17:** The modified header and footer templates with `Label` controls

**Note** You can end editing a template at any time by right-clicking your `DataList` and selecting End Template Editing.

3. Next, you'll edit the item template and add `Label` controls to display the `ProductID`, `ProductName`, `QuantityPerUnit`, and `UnitPrice` columns. Right-click your `DataList` and select Edit Template > Item Templates. [Figure 15.18](#) shows the Item Templates editor.



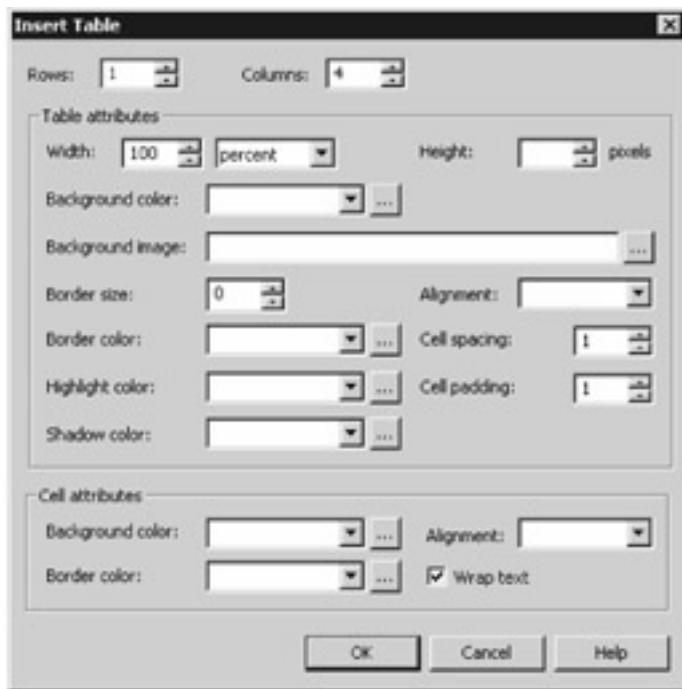
**Figure 15.18:** The Item Templates editor

As you can see from [Figure 15.18](#), the Item Templates editor is divided into the following four areas:

- **ItemTemplate** Contains controls that you typically use to display column values.
- **AlternatingItemTemplate** Contains controls that are shown after the controls in the `ItemTemplate`.
- **SelectedItemTemplate** Contains controls that are shown when you select an item.
- **EditItemTemplate** Contains controls that are shown when you edit an item.

You'll add a table to the `ItemTemplate` area, and then you'll add four `Label` controls in the cells of your table. The four `Label` controls will display the values for the `ProductID`, `ProductName`, `QuantityPerUnit`, and `UnitPrice` columns. To add a table:

1. Click anywhere in the `ItemTemplate` area and select Table > Insert Table. Set the properties for the table as shown in [Figure 15.19](#).



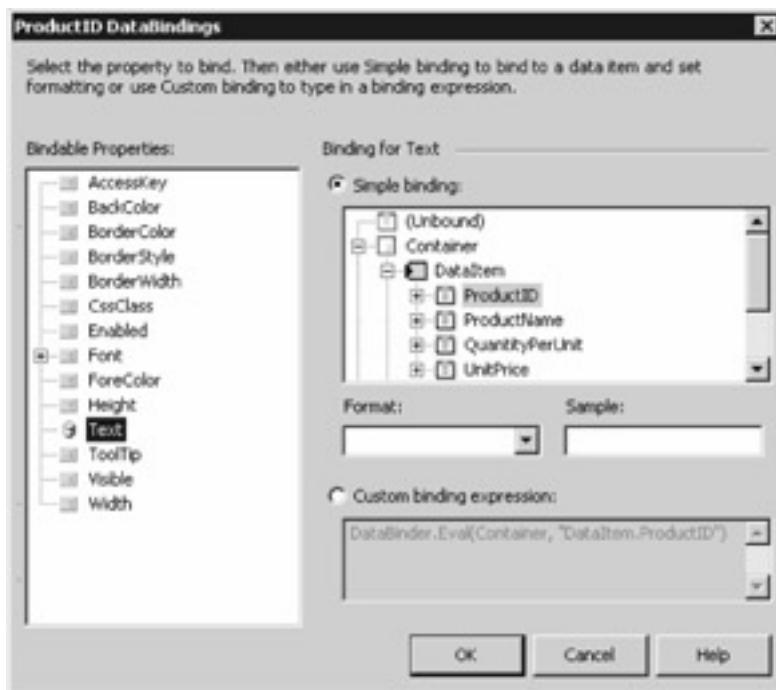
**Figure 15.19:** Setting the properties of the table

2. Next, drag a `Label` to the first cell in the table. You'll use this first `Label` to display the `ProductID` column. Set the `ID` property of your `Label` to `ProductID`, as shown in [Figure 15.20](#).



**Figure 15.20:** Adding the Label

3. To get the `Label` to display the `ProductID` column, you'll need to bind it to that column. To do this, click the ellipsis button in the `DataBindings` property. You'll then see the `DataBindings` dialog box. Open the Container node by clicking the addition icon, and then open the `DataItem` node; finally, select the `ProductID` column, as shown in [Figure 15.21](#).



**Figure 15.21:** Binding the Label to the Product ID column

4. Next, add three more `Label` controls in the remaining cells of your table. Set the `ID` property for your three `Label` controls to `ProductName`, `QuantityPerUnit`, and `UnitPrice`, respectively. Also, bind each of your `Label` controls to the `ProductName`, `QuantityPerUnit`, and `UnitPrice` columns, respectively.

**Warning** As you add your `Label` controls to the cells, you'll notice that the remaining cells shrink. Watch out for that as it can make adding the other `Label` controls a little tricky.

5. Next, you'll modify the HTML for your form to make the table a little easier to read. You'll change the `width` and `border` attributes of the `TABLE` tag and setting the `width` attribute of the `TD` tags.

**Note** The `TABLE` tag defines a table, and the `TD` tag defines an element in a row.

6. To view the HTML code for your form, click the `HTML` link under the form designer to view the code for your form. Set the `width` and `border` attributes of your `TABLE` tag to 320 and 1, respectively, and set the `width` attributes of the four `TD` tags to 20, 100, 100, and 100, respectively. The following HTML shows these changes:

```
<TABLE id="Table5" cellSpacing="1" cellPadding="1" width="320" border="1">
<TR>
  <TD width="20">
    <asp:Label id=ProductID runat="server" Text='
      <%# DataBinder.Eval(Container, "DataItem.ProductID") %>'>
  </asp:Label></TD>
  <TD width="100">
    <asp:Label id=ProductName runat="server" Text='
      <%# DataBinder.Eval(Container, "DataItem.ProductName") %>'>
  </asp:Label></TD>
  <TD width="100">
    <asp:Label id=QuantityPerUnit runat="server" Text='
      <%# DataBinder.Eval(Container, "DataItem.QuantityPerUnit") %>'>
  </asp:Label></TD>
  <TD width="100">
    <asp:Label id=UnitPrice runat="server" Text='
      <%# DataBinder.Eval(Container, "DataItem.UnitPrice") %>'>
  </asp:Label></TD>
</TR>
</TABLE>
```

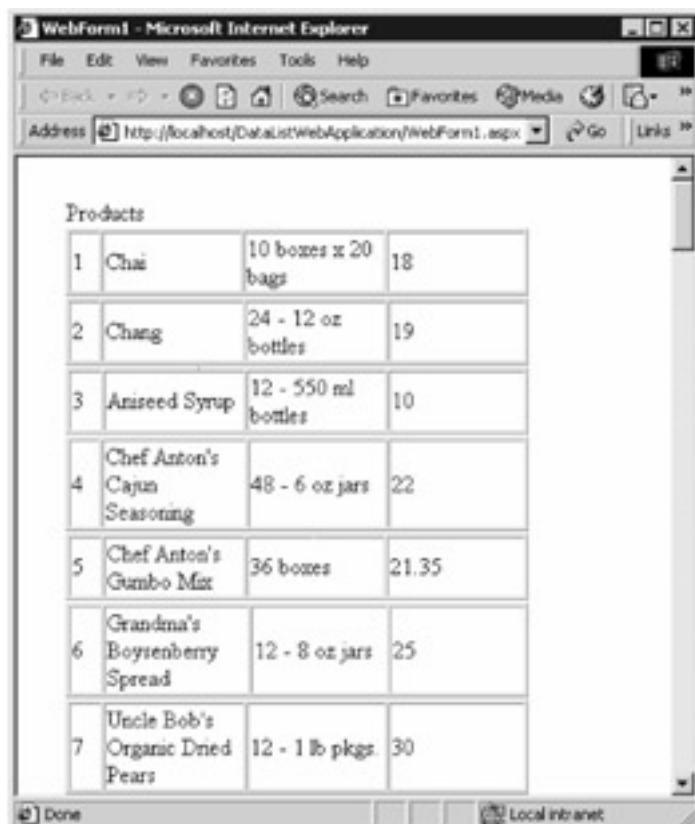
**Note** The *ID* attribute of your *TABLE* tag might differ from that shown in the previous code. Don't worry about changing the *ID* attribute for your *TABLE* tag.

7. Next, you'll need to add code to populate `sqlDataAdapter1` with the rows retrieved by your `SELECT` statement. Typically, the best place to place this code is in the `Page_Load()` method of your form. The `Page_Load()` method is called when the Web page containing your form is initially loaded or refreshed. Open the code for your form by selecting `View > Code`, or press F7 on your keyboard. Set your `Page_Load()` method to the following:

```
private void Page_Load(object sender, System.EventArgs e)
{
    // Put user code to initialize the page here
    if (!this.IsPostBack)
    {
        sqlDataAdapter1.Fill(dataSet11, "Products");
        this.DataBind();
    }
}
```

The `Fill()` method retrieves the rows from the `Products` table and populates the `dataSet11` object with those rows. The `DataBind()` method then causes the rows to be displayed in the `DataList` of your form.

To run your form, select `Debug > Start Without Debugging`, or press `Ctrl+F5` on your keyboard. [Figure 15.22](#) shows the running form.



**Figure 15.22:** The running form

Once you've finished running your form, close it and return to the form designer. Click the `HTML` link to view the code for your form. [Listing 15.5](#) shows the `WebForm1.aspx` file for the form. You'll notice that this file contains a `DataList` control with the appropriate columns.

### **Listing 15.5: THE WebForm1.aspx FILE**

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
AutoEventWireup="false"
Inherits="DataListWebApplication.WebForm1" %>
```

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>WebForm1</title>
    <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
    <meta content="C#" name="CODE_LANGUAGE">
    <meta content="JavaScript" name="vs_defaultClientScript">
    <meta content="http://schemas.microsoft.com/intellisense/ie5"
      name="vs_targetSchema">
  </HEAD>
  <body MS_POSITIONING="GridLayout">
    <form id="Form1" method="post" runat="server">
      <asp:datalist id=DataList1 style="Z-INDEX: 101; LEFT: 33px;
        POSITION: absolute; TOP: 28px" runat="server"
        DataMember="Products" Height="140" Width="297" DataSource=""
        <%# dataSet1 %>">
        <HeaderTemplate>
          <asp:Label id="Label1" runat="server">Products</asp:Label>
        </HeaderTemplate>
        <FooterTemplate>
          <asp:Label id="Label2" runat="server">End of list</asp:Label>
        </FooterTemplate>
        <ItemTemplate>
          <TABLE id="Table5" cellSpacing="1" cellPadding="1" width="320"
            border="1">
            <TR>
              <TD width="20">
                <asp:Label id=ProductID runat="server" Text='
                  <%# DataBinder.Eval(Container, "DataItem.ProductID") %>'>
              </asp:Label></TD>
              <TD width="100">
                <asp:Label id=ProductName runat="server" Text='
                  <%# DataBinder.Eval(Container,
                    "DataItem.ProductName") %>'>
              </asp:Label></TD>
              <TD width="100">
                <asp:Label id=QuantityPerUnit runat="server" Text='
                  <%# DataBinder.Eval(Container, "DataItem.QuantityPerUnit") %>'>
              </asp:Label></TD>
              <TD width="100">
                <asp:Label id=UnitPrice runat="server" Text='
                  <%# DataBinder.Eval(Container, "DataItem.UnitPrice") %>'>
              </asp:Label></TD>
            </TR>
          </TABLE>
        </ItemTemplate>
      </asp:datalist></form>
    </body>
  </HTML>

```

## Maintaining State in a Web Application

The Hypertext Transport Protocol (HTTP) doesn't maintain state between pages served by your Web server during each round-trip. This means that any information you provided in a form is forgotten when you get a new page. If you're simply receiving static HTML Web pages, then this isn't a problem. If you're placing an order for a product, however, then the server needs to remember what you ordered.

To get the Web server to remember what you did during the last round-trip, you can store information on the server or on the client computer the browser is running on.

Storing information on the client means you don't use up any resources on the server to store that information, and your Web application can potentially handle many more users. Storing information on the server gives you more control of the stored information, but since this consumes server resources, you need to be careful not to store too much; otherwise your Web application won't be able to handle many users.

### Storing Information on the Client

To store information on the client, you can use cookies or the `Page` object's `ViewState` property. Let's take a look at how you use cookies and the `ViewState` property.

#### Storing Information using Cookies

A *cookie* is a name and value pair that is stored in a small file that resides on the hard drive of the client computer. You use the name to identify the value being stored; both the name and value are `string` objects.

**Warning** Cookies are potentially problematic because the user can configure their browser to prevent cookies from being stored. Also, a browser stores only a limited number of cookies: 300 in total and no more than 20 per Web server. You should therefore use cookies sparingly—if at all.

The following example creates an `int` variable named `myInt` that is set to 1 and creates an `HttpCookie` object that stores `myInt` under the name `count`:

```
int myInt = 1;
HttpCookie myHttpCookie = new HttpCookie("count", myInt.ToString());
```

Because a cookie stores the value as a `string`, you use the `ToString()` method to convert `myInt` to a `string` before storing it in `myHttpCookie`.

To store the cookie on the client, you call the `AppendCookie()` method of the `Page` object's `Response`:

```
Response.AppendCookie(myHttpCookie);
```

The `Response` object is the HTTP response sent by the Web server to the browser. When this code is run, it causes the browser to store the cookie on the client computer's hard disk in the directory specified in the settings for the browser.

You can retrieve the `count` value from the `Cookies` collection of the `Request` object:

```
myInt = Int32.Parse(Request.Cookies["count"].Value);
```

The `Request` object is sent by the browser to the Web server and contains the cookie previously set. Because the `count` value is stored as a `string`, you use the static `Parse()` method of the `Int32` structure to convert the `string` to an `int`.

[Listing 15.6](#) shows an example ASP.NET application that uses a cookie to keep track of the number of times the page has been viewed.

#### Listing 15.6: CookieTest.aspx

```
<!--
CookieTest.aspx illustrates the use of a cookie to
store information on the client
-->
```

```

<html>

<head>
<script language="C#" runat="server">

    void Page_Load(Object sender, EventArgs e)
    {
        int myInt;

        // check if count is null
        if (Request.Cookies["count"] == null)
        {
            // count is null, so initialize myInt to 1
            myInt = 1;

            // create an HttpCookie object
            HttpCookie myHttpCookie = new HttpCookie("count", myInt.ToString());

            // add HttpCookie object to Response
            Response.AppendCookie(myHttpCookie);
        }
        else
        {
            // retrieve count and increment myInt by 1
            myInt = Int32.Parse(Request.Cookies["count"].Value) + 1;
        }
        // set count value to myInt
        Response.Cookies["count"].Value = myInt.ToString();

        // display myInt in myLabel
        myLabel.Text = "This page has been viewed " + myInt.ToString() +
            " times.";
    }
}

</script>
</head>

<body>
<asp:Label id="myLabel" runat="server"/>
<form runat="server">
<asp:Button text="Press the Button!" runat="server"/>
</form>
</body>

</html>

```

---

**Note** Notice that you can embed C# code directly into an `.aspx` file. The `CookieTest.aspx` file was created using Microsoft Notepad.

To run `CookieText.aspx`, simply copy this file into your `Inetpub\wwwroot` directory and point your browser to `http://localhost/CookieTest.aspx`. [Figure 15.23](#) shows the page generated by `CookieTest.aspx`-assuming that the button on the page has been repeatedly pressed.



**Figure 15.23:** The running `CookieTest.aspx` page

## Storing Information using the `ViewState` Property

You use the `Page` object's `ViewState` property to access a `StateBag` object, which stores a collection of name and value pairs on the client computer. You use the name to identify the value being stored. The name is a `string` and the value is an `object`. Unlike a cookie, a user cannot prevent values from being stored using the `ViewState` property. One use for the `ViewState` property would be to store a user's name.

**Tip** Since the values are sent back and forth between the client and the server, you should store only a small amount of information using the `ViewState` property. This is still a better solution than using cookies because the user can always prevent cookies from being stored.

The following example stores `myInt` under the name `count`:

```
int myInt = 1;
ViewState["count"] = myInt;
```

You can then retrieve the `count` value using the following code:

```
myInt = (int) ViewState["count"];
```

Because a value is stored as an `object`, you must cast it to the specific type you want to use. In this example, the `count` value is cast to an `int`.

[Listing 15.7](#) shows an example ASP.NET page that uses the `ViewState` property to keep track of the number of times the page has been viewed.

### Listing 15.7: `ViewStateTest.aspx`

```
<!--
  ViewStateTest.aspx illustrates the use of ViewState to
  store information on the client
-->

<html>

<head>
<script language="C#" runat="server">

  void Page_Load(Object sender, EventArgs e)
  {
    int myInt;
```

```

// check if count is null
if (ViewState["count"] == null)
{
    // count is null, so initialize myInt to 1
    myInt = 1;
}
else
{
    // retrieve count and increment myInt by 1
    myInt = (int) ViewState["count"] + 1;
}

// set count value to myInt
ViewState["count"] = myInt;
// display myInt in myLabel
myLabel.Text = "This page has been viewed " + myInt.ToString() +
    " times.";
}

</script>
</head>

<body>
<asp:Label id="myLabel" runat="server"/>
<form runat="server">
<asp:Button text="Press the Button!" runat="server"/>
</form>
</body>

</html>

```

---

## Storing Information on the Server

To store information on the server, you can use the `Page` object's `Session`, `Application`, or `Cache` object. These objects all store information in the form of name and value pairs, where the name is a `string` and the value is an `object`. You can also store information in the database itself, which is the best solution if you need to store a lot of information about a user or the application. Finally, you can of course always store information in static variables or objects.

You'll learn about the `Session`, `Application`, and `Cache` objects in the next sections. I'll also discuss storing information about a Web application in the database.

### Storing Information Using a Session Object

A `Session` object allows you to store separate information for each user. The information stored in the `Session` object remains on the server up to a default time of 20 minutes, after which the information is thrown away. One use for the `Session` object might be to store the user's name.

**Tip** Because each `Session` object stores information for a single user, store the absolute minimum information for each user. Otherwise, your Web server could be swamped with `Session` objects and run out of memory, and your application wouldn't support large numbers of users.

The information is stored in name and value pairs, where the name is a `string` and the value is an `object`. The following example stores `myInt` under the name `count`:

```

int myInt = 1;
Session["count"] = myInt;

```

You can then retrieve the `count` value using the following code:

```
myInt = (int) Session["count"];
```

Because a value is stored as an `object`, you must cast it to the specific type you want to use. In this example, the `count` value is cast to an `int`.

[Listing 15.8](#) shows an example ASP.NET page that uses the `Session` object to keep track of the number of times the page has been viewed. This information is specific to each user, and therefore shows the total number of times the page has been viewed by the current user.

### **Listing 15.8: SessionObjectTest.aspx**

---

```
<!--
  SessionObjectTest.aspx illustrates the use of the
  Session object to store information on the server.
  This information is specific for each user.
-->

<html>
<head>
<script language="C#" runat="server">

  void Page_Load(Object sender, EventArgs e)
  {
    int myInt;

    // check if count is null
    if (Session["count"] == null)
    {
      // count is null, so initialize myInt to 1
      myInt = 1;
    }
    else
    {
      // retrieve count and increment myInt by 1
      myInt = (int) Session["count"] + 1;
    }

    // set count value to myInt
    Session["count"] = myInt;

    // display myInt in myLabel
    myLabel.Text = "This page has been viewed " + myInt.ToString() +
      " times.";
  }

</script>
</head>
<body>
<asp:Label id="myLabel" runat="server"/>
<form runat="server">
<asp:Button text="Press the Button!" runat="server"/>
</form>
</body>
</html>
```

---

## Storing Information using the *Application* Object

The `Application` object allows you to store information that is shared for all users. One use for the `Application` object might be to store a `DataSet` object containing a product catalog. The information is stored in name and value pairs, where the name is a `string` and the value is an `object`.

The following example stores `myInt` under the name `count`:

```
int myInt = 1;
Application["count"] = myInt;
```

You can then retrieve the `count` value using the following code:

```
myInt = (int) Application["count"];
```

[Listing 15.9](#) shows an example ASP.NET page that uses the `Application` object to keep track of the number of times the page has been viewed. This information is shared by all users, and therefore shows the total number of times the page has been viewed by all users.

### **Listing 15.9: ApplicationObjectTest.aspx**

---

```
<!--
  ApplicationObjectTest.aspx illustrates the use of the
  Application object to store information on the server.
  This information is shared for all users.
-->

<html>

<head>
<script language="C#" runat="server">

  void Page_Load(Object sender, EventArgs e)
  {
    int myInt;

    // check if count is null
    if (Application["count"] == null)
    {
      // count is null, so initialize myInt to 1
      myInt = 1;
    }
    else
    {
      // retrieve count and increment myInt by 1
      myInt = (int) Application["count"] + 1;
    }

    // set count value to myInt
    Application["count"] = myInt;

    // display myInt in myLabel
    myLabel.Text = "This page has been viewed " + myInt.ToString() +
      " times.";
  }

</script>
</head>

<body>
<asp:Label id="myLabel" runat="server" />
<form runat="server">
```

```
<asp:Button text="Press the Button!" runat="server"/>
</form>
</body>

</html>
```

---

## Storing Information using the *Cache* Object

Like the `Application` object, the `Cache` object is also shared for all users, but it provides more functionality than the `Application` object. For example, you can control when the stored information is removed. For more information about the `Cache` object, consult the .NET online documentation as described in [Chapter 1](#), "Introduction to Database Programming with ADO.NET." Look up "Cache class" in the index of the online documentation.

## Storing Information using the Database

If you have a large amount of information to store about a user, store it in the database rather than the `Session` object. For example, if you're building a Web site that a user can order products from, store their shopping cart in the database.

Team LiB

◀ PREVIOUS

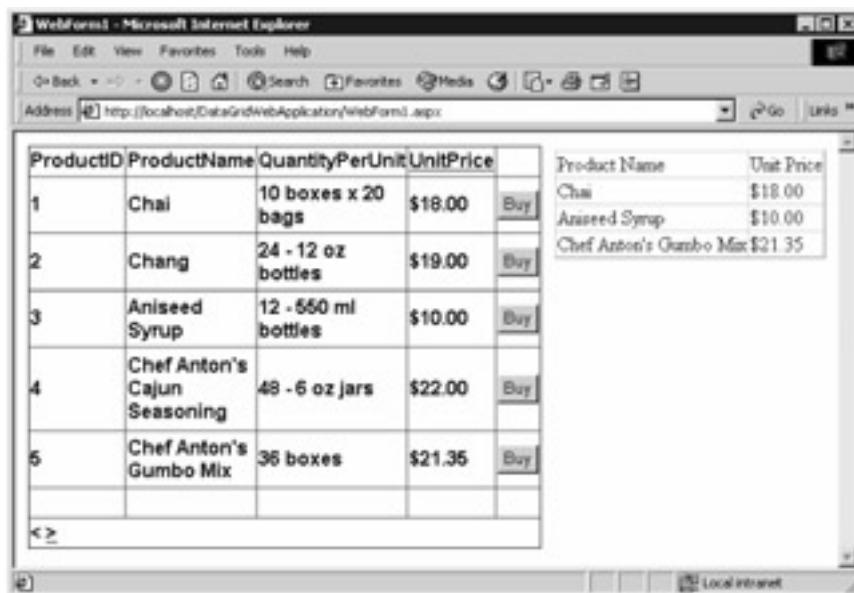
NEXT ▶

## Creating a Simple Shopping Cart Application

In this section, you'll modify your `DataGridWebApplication` you created earlier to turn it into a simple shopping cart. You'll store the shopping cart in a `Session` object.

**Note** As mentioned in the [previous section](#), in a real application you'll probably want to store the shopping cart in the database rather than in a `Session` object.

[Figure 15.24](#) shows the final running form that you'll build. You use the `Buy` button to add a product to the shopping cart shown on the right of the form. As you can see from [Figure 15.24](#), I've added three products to the shopping cart by pressing the `Buy` button for each product in the grid on the left.



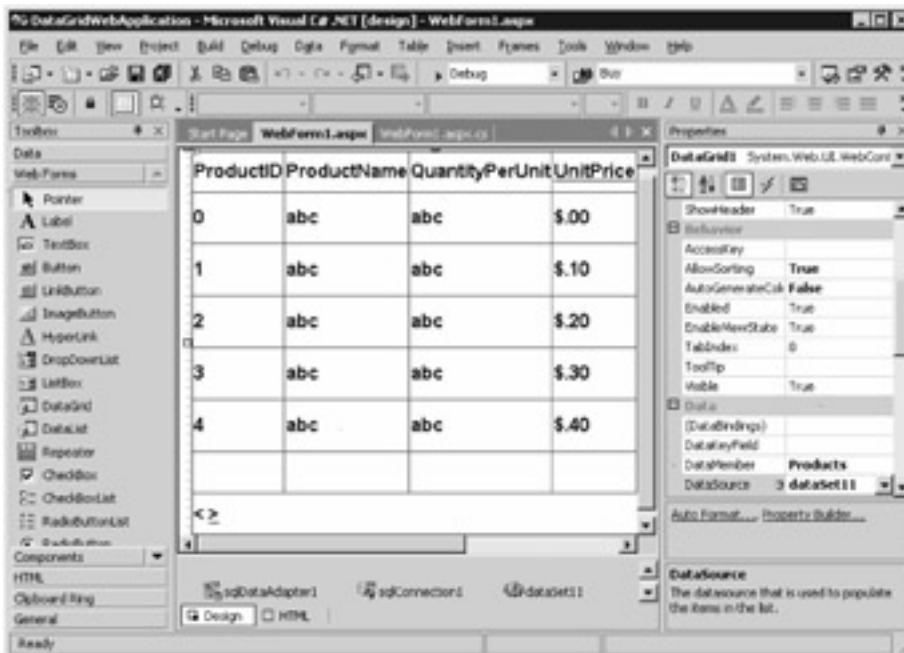
**Figure 15.24:** The running form

You can either follow the steps shown in the following sections to add the `Buy` button and the shopping cart to your form, or you can replace the `ASP.NET` code in your form with the code in the `WebForm1.aspx` file contained in the `VS .NET Projects\DataGridWebApplication` directory. You replace the code in your form by selecting and deleting the existing code in your form and pasting in the code from the `WebForm1.aspx` file. You'll also need to replace the code behind your form with the code in the `WebForm1.aspx.cs` file contained in the `VS .NET Projects\DataGridWebApplication` directory.

### Adding the Buy Button

In this section, you'll add the `Buy` button to the `DataGrid1` object of the `DataGridWebApplication`. Perform the following steps:

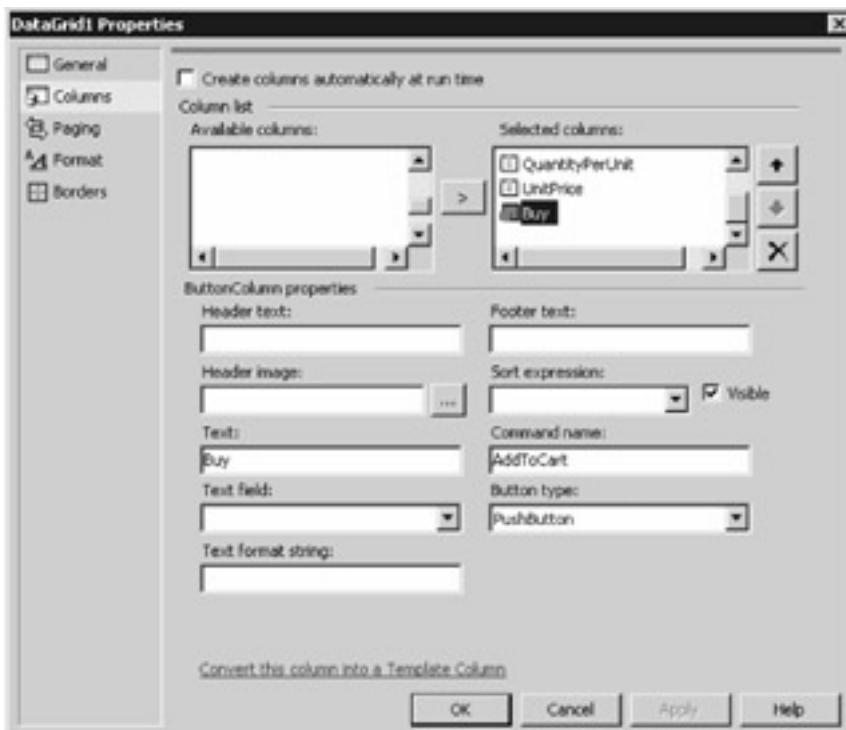
1. Open `DataGridWebApplication` by selecting `File > Open > Project`, double-click the `DataGridWebApplication` folder, and double-click the `DataGridWebApplication.sln` file.
2. Open the `WebForm1.aspx` file in Design mode by double-clicking this file in the Solution Explorer window. Click on the `DataGrid1` control in the form. [Figure 15.25](#) shows the properties for `DataGrid1`.

**Figure 15.25:** DataGrid1 properties

3. Next, click the Property Builder link near the bottom of the Properties window. Do the following to add the Buy button:

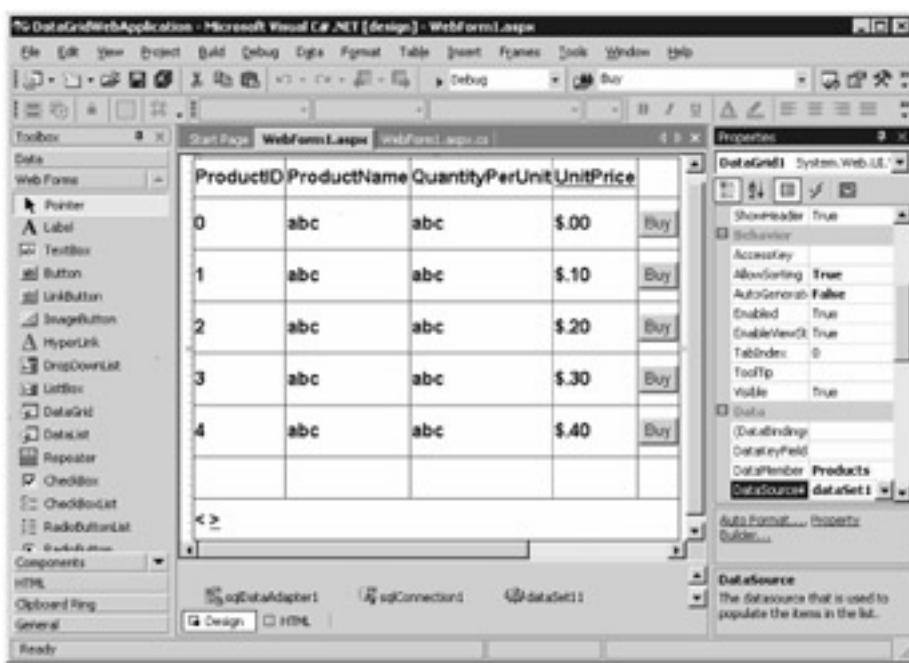
1. Click Columns on the left of the DataGrid1 Properties dialog box.
2. Expand the Button Column node of the Available Columns section.
3. Add a Select button to the Selected Columns area.
4. Set Text to Buy. This is the text that is shown on the button.
5. Set Command Name to AddToCart. This is the method that is called when the button is pressed. (You'll create this method later.)
6. Set Button Type to PushButton.

Figure 15.26 shows the final properties of the Buy button.



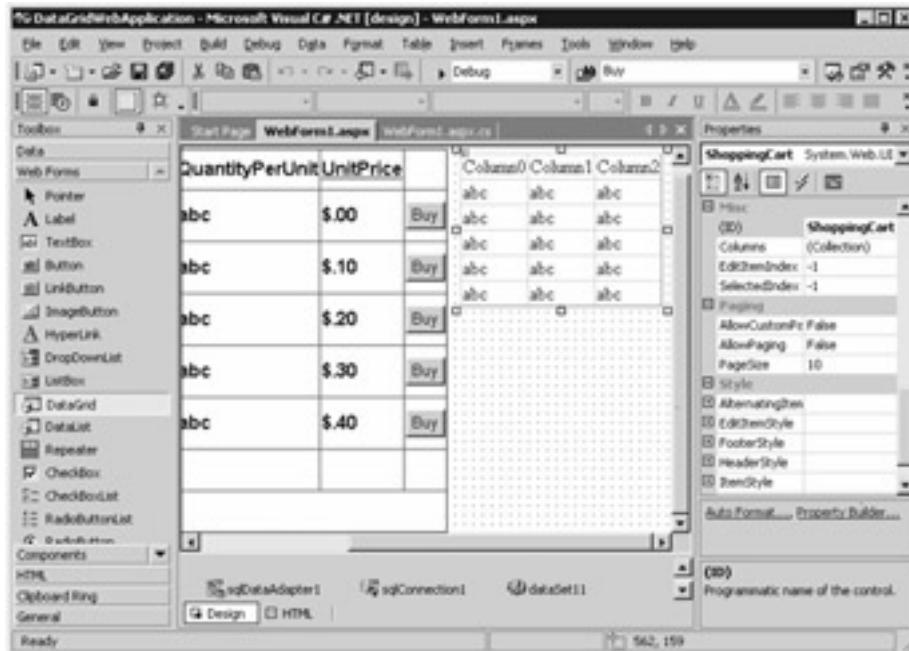
**Figure 15.26:** Buy button properties

4. Click OK to add the button to DataGrid1. [Figure 15.27](#) shows DataGrid1 with the newly added Buy button.

**Figure 15.27:** DataGrid1 with Buy button

## Adding the Shopping Cart

In this section, you'll add a **DataGridView** to store the shopping cart. Drag a **DataGridView** control from the Toolbox to the right of DataGrid1 on your form. Set the ID of this new **DataGridView** to **ShoppingCart**, as shown in [Figure 15.28](#).

**Figure 15.28:** ShoppingCart DataGridView

**Note** You might have to close all windows except the form designer so that you have enough screen space to add the new **DataGridView** to your form.

## Adding Code to the **WebForm1.aspx.cs** File

Your next task is to add some additional code to the `WebForm1.aspx.cs` file to support the shopping cart. As mentioned earlier, either you can follow the steps shown in this section or you can replace the code behind your form with the code in the `WebForm1.aspx.cs` file contained in the `VS .NET Projects\GridViewWebApplication` directory. You replace the code in your form by selecting and deleting the existing code in your form and pasting in the code from the `WebForm1.aspx.cs` file.

Perform the following steps if you want to modify the code yourself:

1. Select View > Code, or press F7 on your keyboard to view the code. Add a `DataTable` object named `Cart` and a `DataView` object named `CartView` to the `WebForm1` class, as shown in the following code:

```
public class WebForm1 : System.Web.UI.Page
{
    protected DataTable Cart;
    protected DataView CartView;
```

Your `Cart` object is used to store the shopping cart and will be populated with the products selected using the Buy button. Your `CartView` object is used to view the shopping cart.

2. Next, set your `Page_Load()` method to the following code; notice that this method creates a `DataTable` to store the shopping cart, and that this `DataTable` is stored in the `Session` object:

```
private void Page_Load(object sender, System.EventArgs e)
{
    // Put user code to initialize the page here

    // populate the Session object with the shopping cart
    if (Session["ShoppingCart"] == null)
    {
        Cart = new DataTable();
        Cart.Columns.Add(new DataColumn("Product Name", typeof(string)));
        Cart.Columns.Add(new DataColumn("Unit Price", typeof(string)));
        Session["ShoppingCart"] = Cart;
    }
    else
    {
        Cart = (DataTable) Session["ShoppingCart"];
    }
    CartView = new DataView(Cart);
    ShoppingCart.DataSource = CartView;
    ShoppingCart.DataBind();

    if (!this.IsPostBack)
    {
        // populate dataSet11 with the rows from the Products DataTable
        sqlDataAdapter1.Fill(dataSet11, "Products");
        this.DataBind();
    }
}
```

3. Next, you need to add the following `AddToCart()` method to your `WebForm1` class. This method is called when the user presses the Buy button. Notice this method creates a `DataRow` object and populates it with `TableCell` objects to store the product name and unit price in the `Shopping-Cart` `DataTable` that you previously added to your form.

```
protected void AddToCart(Object sender, DataGridCommandEventArgs e)
{
    DataRow product = Cart.NewRow();

    // e.Item is the row of the table where the command is raised.
```

```
// For bound columns the value is stored in the Text property of TableCell
TableCell productNameCell = e.Item.Cells[1];
TableCell unitPriceCell = e.Item.Cells[3];
string productName = productNameCell.Text;
string unitPrice = unitPriceCell.Text;

if (((Button)e.CommandSource).CommandName == "AddToCart")
{
    product[0] = productName;
    product[1] = unitPrice;
    Cart.Rows.Add(product);
}
ShoppingCart.DataBind();
}
```

4. The only thing left to do is to run your form. To do this, select Debug > Start Without Debugging, or press Ctrl+F5 on your keyboard.

Click the Buy button for different products in the grid to add them to your shopping cart.

**Team LiB**

 PREVIOUS  NEXT 

## Summary

HTML creates static Web pages with content that doesn't change. If you want the information to be dynamic, however, then you can use ASP.NET. It enables you to create Web pages with content that can change at runtime, and to develop applications that are accessed using a Web browser. In this chapter, you saw how to use Visual Studio .NET and the C# programming language to create some simple ASP.NET Web applications. This chapter gave you a brief introduction to the large subject of ASP.NET. For thorough coverage of the topic, see Russell Jones's *Mastering ASP .NET with C#*(Sybex, 2002).

There are two main parts to an ASP.NET form: the `.aspx` file, which contains HTML and ASP.NET code, and the `.aspx.cs` file, which contains C# code that supports the Web form. You can think of this C# code as *running behind* the form, and for this reason the `.aspx.cs` file is known as the *code-behind file*. You can view the HTML containing the ASP.NET code for your form by clicking the HTML link at the bottom of the form designer. You can view the code-behind file by selecting View > Code, or you can press F7 on your keyboard.

A `DataGridView` allows you to access rows in a database table. In the following sections, you'll learn how to create an ASP.NET Web application that uses a `DataGridView` control to access the rows in a database table. You customize your `DataGridView` by first selecting the `DataGridView` control and then clicking the Property Builder link at the bottom of the Properties window. This displays the Properties dialog box for your `DataGridView`. The Properties dialog box is divided into five areas: General, Columns, Paging, Format, and Borders. A `DataSource` offers you a lot more flexibility in the presentation of column values than that offered by a `DataGridView`, such as the ability to add headers and footers to the data.

The Hypertext Transport Protocol (HTTP) doesn't maintain state between pages served by your Web server during each round-trip. This means that any information you provided in a form is forgotten when you get a new page. To get the Web server to remember what you did during the last round-trip, you can store information on the server or on the client computer on which the browser is running. Storing information on the client means you don't use up any resources on the server to store that information and your Web application can handle many more users. Storing information on the server gives you more control of the stored information, but since this consumes server resources, you need to be careful not to store too much; otherwise your Web application won't be able to handle many users.

# Chapter 16: Using SQL Server's XML Support

## Overview

XML has become the *lingua franca* of the Web because it is such an ideal format for exchanging information. Already, many companies exchange information with each other using XML sent over the Web.

In this chapter, you'll learn about SQL Server's extensive support for XML. You'll also see how to store XML in a C# program using the  `XmlDocument` and  `XmlDataDocument` objects.

Featured in this chapter:

- Using the SQL Server FOR XML clause
- Introducing XPath and XSLT
- Accessing SQL Server using HTTP
- Using the SQL Server OPENXML( ) function
- Using an  `XmlDocument` object to store an XML document
- Using an  `XmlDataDocument` object to store an XML document

## Using the SQL Server *FOR XML* Clause

With a standard SQL SELECT statement, you submit your SELECT statement to the database for execution and get results back in the form of rows. SQL Server extends the SELECT statement to allow you to query the database and get results back as XML. To do this, you add a FOR XML clause to the end of your SELECT statement. The FOR XML clause specifies that SQL Server is to return results as XML.

The FOR XML clause has the following syntax:

```
FOR XML {RAW | AUTO | EXPLICIT}
[ , XMLDATA]
[ , ELEMENTS]
[ , BINARY BASE64]
```

The RAW, AUTO, and EXPLICIT keywords indicate the XML *mode*. [Table 16.1](#) shows a description of the keywords used in the FOR XML clause. In the next sections, you'll examine some examples of the use of the FOR XML clause.

**Table 16.1: FOR XML KEYWORDS**

KEYWORD	DESCRIPTION
RAW	Specifies that each row in your result set is returned as an XML <row> element. The column values for each row in the result set become attributes of the <row> element.
AUTO	Specifies that each row in the result set is returned as an XML element. The name of the table is used as the name of the tag in the row elements.
EXPLICIT	Indicates your SELECT statement specifies a parent-child relationship. This relationship is then used by SQL Server to generate XML with the appropriate nested hierarchy.
XMLDATA	Specifies that the XML schema is to be included in the returned XML.
ELEMENTS	Specifies that the column values are returned as subelements of the row; otherwise the columns are returned as attributes of the row. You can use this option only with the AUTO mode.
BINARY BASE64	Specifies that any binary data returned by your SELECT statement is encoded in base 64. If you want to retrieve binary data using either the RAW or EXPLICIT mode, then you must use the BINARY BASE64 option.

### Using the RAW Mode

You use the RAW mode to specify that each row in the result set returned by your SELECT statement is returned as an XML <row> element. The column values for each row in the result set become attributes of the <row> element.

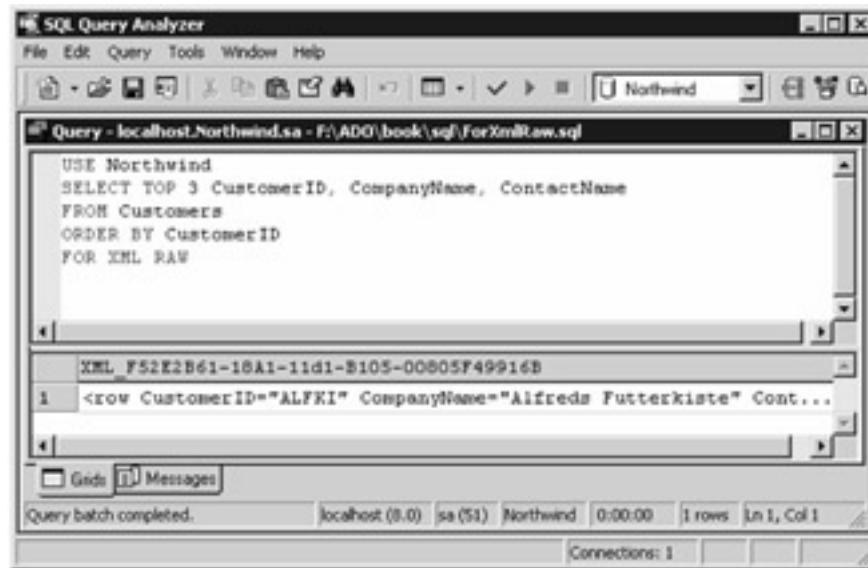
[Listing 16.1](#) shows an example SELECT statement that retrieves the top three rows from the Customers table. The results of the SELECT are returned as XML using the FOR XML RAW clause.

**Listing 16.1: FORXMLRAW.SQL**

```
USE Northwind
SELECT TOP 3 CustomerID, CompanyName, ContactName
FROM Customers
ORDER BY CustomerID
FOR XML RAW
```

**Note** This *SELECT* statement is contained in a T-SQL script named *ForXmlRaw.sql*, which is located in the *sql* directory for this chapter.

You can load the *ForXmlRaw.sql* T-SQL script into Query Analyzer by selecting File > Open from the menu. You then run the script by selecting Query > Execute, or by pressing the F5 key. [Figure 16.1](#) shows the result of running the script in Query Analyzer. You'll notice that the XML is shown on one line, and that the line is truncated.



**Figure 16.1:** Running a *SELECT* statement containing a *FOR XML RAW* clause in Query Analyzer

**Note** By default, the maximum number of characters displayed by Query Analyzer per column is 256. Any results longer than 256 characters will be truncated. For the examples in this section, you'll need to increase the maximum number of characters to 8,192. To do this, you select Tools > Options in Query Analyzer and set the Maximum Characters Per Column field to 8,192.

Here's the XML line returned by the example, which I copied from Query Analyzer and added some return characters to make it easier to read:

```
<row
CustomerID="ALFKI"
CompanyName="Alfreds Futterkiste"
ContactName="Maria Anders"/>
<row
CustomerID="ANATR"
CompanyName="Ana Trujillo Emparedados y helados"
ContactName="Ana Trujillo"/>
<row
CustomerID="ANTON"
CompanyName="Antonio Moreno Taquería"
ContactName="Antonio Moreno"/>
```

Notice that each customer is placed within a *<row>* tag. Also, the column values appear as attributes within each row; for example, in the first row, the *CustomerID* attribute is *ALFKI*.

## Using the *AUTO* Mode

You use the *AUTO* mode to specify that each row in the result set is returned as an XML element. The name of the table is used as the name of the tag in the row elements.

[Listing 16.2](#) shows an example *SELECT* statement that retrieves the top three rows from the *Customers* table. The results are returned as XML using the *FOR XML AUTO* clause.

### **Listing 16.2: FORXMLAUTO.SQL**

```
USE Northwind
SELECT TOP 3 CustomerID, CompanyName, ContactName
FROM Customers
ORDER BY CustomerID
FOR XML AUTO
```

---

The XML returned by this example is as follows:

```
<Customers
CustomerID="ALFKI"
CompanyName="Alfreds Futterkiste"
ContactName="Maria Anders"/>
<Customers
CustomerID="ANATR"
CompanyName="Ana Trujillo Emparedados y helados"
ContactName="Ana Trujillo"/>
<Customers
CustomerID="ANTON"
CompanyName="Antonio Moreno Taquería"
ContactName="Antonio Moreno"/>
```

Notice that each customer appears within a `<Customer>` tag instead of a generic `<row>` tag, as was the case in the previous `RAW` mode example.

## Using the **EXPLICIT** Mode

You use the `EXPLICIT` mode to indicate that your `SELECT` statement specifies a parent-child relationship. This relationship is then used by SQL Server to generate XML with the appropriate nested hierarchy.

When using the `EXPLICIT` mode, you must provide at least two `SELECT` statements. The first `SELECT` specifies the parent row (or rows), and the second specifies the child rows. The rows retrieved by the two `SELECT` statements are related through special columns named `Tag` and `Parent`. `Tag` specifies the numeric position of the element, and `Parent` specifies the `Tag` number of the parent element (if any).

Let's consider an example that uses two `SELECT` statements. The first `SELECT` retrieves the `CustomerID`, `CompanyName`, and `ContactName` for the row from the `Customers` table that has a `CustomerID` of `ALFKI`. The second `SELECT` additionally retrieves the `OrderID` and `OrderDate` from the row in the `Orders` table that also has a `CustomerID` of `ALFKI`. The first `SELECT` statement is as follows:

```
SELECT
    1 AS Tag,
    0 AS Parent,
    CustomerID AS [Customer!1!CustomerID],
    CompanyName AS [Customer!1!CompanyName],
    ContactName AS [Customer!1!ContactName],
    NULL AS [Order!2!OrderID!element],
    NULL AS [Order!2!OrderDate!element]
FROM Customers
WHERE CustomerID = 'ALFKI'
```

The `Tag` column specifies the numeric position of the row in the XML hierarchy. The `Parent` column identifies the parent, which is 0 in the previous `SELECT` statement; that's because this row is the parent, or root, in the XML hierarchy.

**Note** You can also use a `Tag` value of `NULL` to indicate the root.

The `CustomerID`, `CompanyName`, and `ContactName` columns in the previous `SELECT` are supplied an alias using the `AS` keyword, followed by a string that uses the following format:

[*elementName!* *tag!* *attributeName!* *directive*]

where

- ***elementName*** specifies the name of the row element in the returned XML.
- ***tag*** specifies the Tag number.
- ***attributeName*** specifies the name of the column elements in the returned XML.
- ***directive (optional)*** specifies how the element is to be treated in the XML. The directives are shown in [Table 16.2](#).

**Table 16.2: DIRECTIVES**

DIRECTIVE	DESCRIPTION
element	Indicates that the column value appears as a contained row element within the outer row element, rather than an embedded attribute of the outer row element. The element directive may be combined with ID, IDREF, or IDREFS.
hide	Indicates that the column value doesn't appear in the returned XML.
xml	Similar to the element directive except that the column value isn't coded as an entity in the returned XML. This means that the special characters &, ', >, <, and " are left as is. These characters would otherwise be coded as &amp;, &apos;, &gt;, &lt;, and &quot; respectively. The xml directive may be combined with hide.
xmltext	Indicates that the column value is contained in a single tag. To use the xmltext directive, your column type must be varchar, nvarchar, char, nchar, text, or ntext.
cdata	Indicates that the column value is contained within a CDATA section. CDATA sections are used to escape blocks of text containing special characters that would otherwise be interpreted as markup; these characters include &, ', >, <, and ". To use the cdata directive, your column type must be varchar, nvarchar, text, or ntext.
ID	Indicates that the column value is an ID attribute. An IDREF and IDREFS attribute can point to an ID attribute, allowing you to create links within the XML.
IDREF	Indicates that the column value is an IDREF attribute.
IDREFS	Indicates that the column value is an IDREFS attribute.

Let's consider an example: CustomerID AS [Customer!1!CustomerID] specifies that the CustomerID column value will appear within the Customer row element with the attribute name of CustomerID.

After the ContactName in the previous SELECT clause, appear two NULL columns; these are used as placeholders for the OrderID and OrderDate columns that are retrieved by the second SELECT statement, which you'll see next. These two columns use the element directive, which indicates that the column values are to appear as contained elements within the Customer row element.

The second SELECT statement retrieves the rows from the Orders table that has a CustomerID of ALFKI:

```

SELECT
  2 AS Tag,
  1 AS Parent,
  C.CustomerID,
  C.CompanyName,
  C.ContactName,
  O.OrderID,
  O.OrderDate
FROM Customers C, Orders O
WHERE C.CustomerID = O.CustomerID
AND C.CustomerID = 'ALFKI'
  
```

Notice that the `Parent` column is set to 1, which indicates that the parent is the row previously retrieved by the first `SELECT` statement shown earlier.

[Listing 16.3](#) shows a complete example that uses the two `SELECT` statements shown in this section.

### **Listing 16.3: FORXMLEXPLICIT.SQL**

---

```
USE Northwind

SELECT
    1 AS Tag,
    0 AS Parent,
    CustomerID AS [Customer!1!CustomerID],
    CompanyName AS [Customer!1!CompanyName],
    ContactName AS [Customer!1!ContactName],
    NULL AS [Order!2!OrderID!element],
    NULL AS [Order!2!OrderDate!element]
FROM Customers
WHERE CustomerID = 'ALFKI'

UNION ALL

SELECT
    2 AS Tag,
    1 AS Parent,
    C.CustomerID,
    C.CompanyName,
    C.ContactName,
    O.OrderID,
    O.OrderDate
FROM Customers C, Orders O
WHERE C.CustomerID = O.CustomerID
AND C.CustomerID = 'ALFKI'

FOR XML EXPLICIT
```

---

**Note** The `UNION ALL` clause causes the results retrieved by the two `SELECT` statements to be merged into one result set.

The combined result set produced by the `UNION ALL` clause is then converted to XML by the `FOR XML EXPLICIT` clause.

The XML returned by the example is as follows:

```
<Customer
CustomerID="ALFKI"
CompanyName="Alfreds Futterkiste"
ContactName="Maria Anders">
<Order>
<OrderID>10643</OrderID>
<OrderDate>1997-08-25T00:00:00</OrderDate>
</Order>
<Order>
<OrderID>10692</OrderID>
<OrderDate>1997-10-03T00:00:00</OrderDate>
</Order>
<Order>
<OrderID>10702</OrderID>
<OrderDate>1997-10-13T00:00:00</OrderDate>
```

```

</Order>
<Order>
<OrderID>10835</OrderID>
<OrderDate>1998-01-15T00:00:00</OrderDate>
</Order>
<Order>
<OrderID>10952</OrderID>
<OrderDate>1998-03-16T00:00:00</OrderDate>
</Order>
<Order>
<OrderID>11011</OrderID>
<OrderDate>1998-04-09T00:00:00</OrderDate>
</Order>
</Customer>

```

Notice that the `OrderID` and `OrderDate` elements appear as row elements contained in the outer `Order` element. That's because the element directive was specified for the `OrderID` and `OrderDate` elements in the first `SELECT` statement. If the element directive is omitted from the `OrderID` and `OrderDate` elements, then the returned XML is as follows:

```

<Customer
  CustomerID="ALFKI"
  CompanyName="Alfreds Futterkiste"
  ContactName="Maria Anders">
<Order OrderID="10643" OrderDate="1997-08-25T00:00:00" />
<Order OrderID="10692" OrderDate="1997-10-03T00:00:00" />
<Order OrderID="10702" OrderDate="1997-10-13T00:00:00" />
<Order OrderID="10835" OrderDate="1998-01-15T00:00:00" />
<Order OrderID="10952" OrderDate="1998-03-16T00:00:00" />
<Order OrderID="11011" OrderDate="1998-04-09T00:00:00" />
</Customer>

```

Notice that the `OrderID` and `OrderDate` elements are embedded as attributes of the outer `Order` element.

## Using the `XMLELEMENT` Option

You use the `XMLELEMENT` option to specify that the XML schema document type definition (DTD) is to be included in the returned XML. The XML schema contains the name and type of the column attributes.

[Listing 16.4](#) shows an example that uses the `XMLELEMENT` option to return the XML schema along with the `ProductID`, `ProductName`, and `UnitPrice` columns for the top two rows from the `Products` table.

---

### Listing 16.4: FORXMLAUTOXMLELEMENT.SQL

---

```

USE Northwind
SELECT TOP 2 ProductID, ProductName, UnitPrice
FROM Products
ORDER BY ProductID
FOR XML AUTO, XMLELEMENT

```

---

**Note** In this example, I use columns from the `Products` table rather than the `Customers` table because the `Customers` table contains only string column values, and I want you to see some of the different types returned in an XML schema. The `Products` table contains column values that consist of both strings and numbers.

The `ProductID` column is of the SQL Server `int` type, `ProductName` is of the `nvarchar` type, and `UnitPrice` is of the `money` type. The XML returned by this example is as follows:

```

<Schema name="Schema3"
  xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">

```

```

<ElementType name="Products" content="empty" model="closed">
<AttributeType name="ProductID" dt:type="i4"/>
<AttributeType name="ProductName" dt:type="string"/>
<AttributeType name="UnitPrice" dt:type="fixed.14.4"/>
<attribute type="ProductID"/>
<attribute type="ProductName"/>
<attribute type="UnitPrice"/>
</ElementType>
</Schema>
<Products
  xmlns="x-schema:#Schema3"
  ProductID="1"
  ProductName="Chai"
  UnitPrice="18.0000"/>
<Products
  xmlns="x-schema:#Schema3"
  ProductID="2"
  ProductName="Chang"
  UnitPrice="19.0000"/>

```

Notice the different XML types of the `ProductID`, `ProductName`, and `UnitPrice` attributes specified in the `AttributeType` tag near the start of the previous XML.

For more information, see *XML Schemas* by Chelsea Valentine, Lucinda Dykes, and Ed Tittel (Sybex, 2002).

## Using the *ELEMENTS* Option

You use the `ELEMENTS` option to indicate that the column values are returned as subelements of the row; otherwise the column values are returned as attributes of the row.

**Tip** You can use the `ELEMENTS` option only with the `AUTO mode`.

[Listing 16.5](#) shows an example that uses the `ELEMENTS` option when retrieving the top two rows from the `Customers` table.

---

### Listing 16.5: FORXMLAUTOELEMENTS.SQL

---

```

USE Northwind
SELECT TOP 2 CustomerID, CompanyName, ContactName
FROM Customers
ORDER BY CustomerID
FOR XML AUTO, ELEMENTS

```

---

The XML returned by this example is as follows:

```

<Customers>
<CustomerID>ALFKI</CustomerID>
<CompanyName>Alfreds Futterkiste</CompanyName>
<ContactName>Maria Anders</ContactName>
</Customers>
<Customers>
<CustomerID>ANATR</CustomerID>
<CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
<ContactName>Ana Trujillo</ContactName>
</Customers>

```

Notice that the column values are returned as subelements within the `Customers` rows.

## Using the *BINARY BASE64* Option

You use the `BINARY BASE64` option to specify that any binary data returned by your `SELECT` statement is encoded in base 64.

**Note** If you want to retrieve binary data using either the `RAW` or `EXPLICIT` mode, then you must use the `BINARY BASE64` option.

In the examples in this section, I'll use the `Employees` table of the Northwind database. This table contains details of the employees that work for the fictional Northwind Company and contains a column named `Photo`. The `Photo` column is of the SQL Server `image` type and contains binary data with an image of the employee.

[Figure 16.2](#) shows a `SELECT` statement run in Query Analyzer that retrieves the `EmployeeID` (the primary key), `FirstName`, `LastName`, and `Photo` columns from the `Employees` table. Notice that the binary data is retrieved as hexadecimal digits (base 16).

The screenshot shows the SQL Query Analyzer interface. The query window contains the following code:

```
USE Northwind
SELECT EmployeeID, FirstName, LastName, Photo
FROM Employees
```

The results grid displays the following data:

	EmployeeID	FirstName	LastName	Photo
1	1	Mary	Isidor	0x151C2F000230000001000E0014D01100FF7FFFF742697469617020...
2	2	Andrew	Fuller	0x151C2F000230000001000E0014D02100FF7FFFF742697469617020...
3	3	Janet	Leverling	0x151C2F000230000001000E0014D02100FF7FFFF742697469617020...
4	4	Margaret	Penwick	0x151C2F000230000001000E0014D02100FF7FFFF742697469617020...
5	5	Steven	Bachman	0x151C2F000230000001000E0014D02100FF7FFFF742697469617020...
6	6	Michael	Sayles	0x151C2F000230000001000E0014D02100FF7FFFF742697469617020...
7	7	Robert	King	0x151C2F000230000001000E0014D02100FF7FFFF742697469617020...
8	8	Laura	Callahan	0x151C2F000230000001000E0014D02100FF7FFFF742697469617020...
9	9	Anne	Dodsworth	0x151C2F000230000001000E0014D02100FF7FFFF742697469617020...

The status bar at the bottom indicates "Query batch completed." and shows connection information: "localhost (8.0) (sa) (51) Northwind 0:00:00 9 rows [Int, Col 1]".

**Figure 16.2:** Retrieving rows from the `Employees` table

In `AUTO` mode, binary data is returned by default as a reference to the data rather than the actual data itself. The following example retrieves the `EmployeeID` and `Photo` columns for the top two rows from the `Employees` table using the `AUTO` mode:

```
USE Northwind
SELECT TOP 2 EmployeeID, Photo
FROM Employees
ORDER BY EmployeeID
FOR XML AUTO
```

This example returns the following XML:

```
<Employees
  EmployeeID="1"
  Photo="dbobject/Employees[@EmployeeID='1']/@Photo" />
<Employees
  EmployeeID="2"
  Photo="dbobject/Employees[@EmployeeID='2']/@Photo" />
```

The reference to the binary data contained in the `Photo` column is actually an XPath expression. (You'll learn about XPath in the [next section](#).)

To get the binary data itself, rather than the reference to it, you need to use the `BINARY BASE64` option. [Listing 16.6](#) shows an example that uses the `BINARY BASE64` option when retrieving the `EmployeeID` and `Photo` columns for the top two rows from the `Employees` table.

### **Listing 16.6: FORXMLAUTOBINARYBASE64.SQL**

```
USE Northwind
```

```
SELECT TOP 2 EmployeeID, Photo  
FROM Employees  
ORDER BY EmployeeID  
FOR XML AUTO, BINARY BASE64
```

---

The XML returned by this example is as follows:

```
<Employees  
EmployeeID="1"  
Photo="FRwvAAIAAA..." />  
<Employees  
EmployeeID="2"  
Photo="FRwvAAIAAA..." />
```

**Note** I've shown only the first 10 digits of binary data. To view the binary data in Query Analyzer, you'll need to set the Default results target to Results To Text in the Options dialog box. You select Tools > Options from the menu to get to this dialog box.

Team LiB

◀ PREVIOUS

NEXT ▶

## Introducing XPath

The Extensible Markup Language Path (XPath) is a language that allows you to search and navigate an XML document, and you can use XPath with SQL Server. In this section, you'll explore the structure of an XML document and how to navigate and search an XML document using XPath. In later sections, you'll see how to use XPath with SQL Server.

### XML Document Structure

An XML document file is divided into nodes, with the topmost node being referred to as the *root node*. The easiest way to understand how the structure works is to consider an example XML document; [Listing 16.7](#) shows an XML document contained in the file `Customers.xml`.

#### **Listing 16.7: CUSTOMERS.XML**

```
<?xml version="1.0"?>

<NorthwindCustomers>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkiste</CompanyName>
    <PostalCode>12209</PostalCode>
    <Country>Germany</Country>
    <Phone>030-0074321</Phone>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
    <PostalCode>05021</PostalCode>
    <Country>Mexico</Country>
    <Phone>(5) 555-4729</Phone>
  </Customers>
</NorthwindCustomers>
```

**Note** You'll find all the XML files in the `xml` directory for this chapter.

**Note** The line `<?xml version="1.0"?>` indicates that `Customers.xml` is an XML file that uses the 1.0 standard.

[Figure 16.3](#) shows a visual representation of the `Customers.xml` document structure.



**Figure 16.3:** Customers.xml document structure

As you can see from [Figure 16.3](#), an XML document is structured like an inverted tree. `NorthwindCustomers` is the *root node*. The two `Customers` nodes beneath the root node are known as a *node set*. The `CustomerID`, `CompanyName`, `PostalCode`, `Country`, and `Phone` are known as *elements*. Each `Customers` node and its `CustomerID`, `CompanyName`, `PostalCode`, `Country`, and `Phone` elements are known as a *node subtree*. A node located beneath another node is known as a *child node*, and the node above is known as the *parent node*; for example, the `NorthwindCustomers` node is the parent node of the child `Customers` nodes.

You can view an XML file using Microsoft Internet Explorer, as shown in [Figure 16.4](#).



**Figure 16.4:** Viewing `Customers.xml` in Internet Explorer

**Tip** To open the XML file, right-click `Customers.xml` in Windows Explorer and select Open With ➤ Internet Explorer from the pop-up menu.

## XPath Expressions

To search or navigate an XML document file you supply an *expression* to XPath. These expressions work within a *context*, which is the current node being accessed within the XML file. The most commonly used ways of specifying the context are shown in [Table 16.3](#).

**Table 16.3: SPECIFYING THE CONTEXT**

CHARACTERS	DESCRIPTION
/	Specifies the root node as the context.
. /	Specifies the current node as the context.
... /	Specifies the parent node as the context.
//	Specifies the whole XML document as the context.
. //	Specifies the whole XML document starting at the current node as the context.

Let's take a look at some example XPath expressions. The following example returns the `Customers` nodes:

`/NorthwindCustomers/Customers`

As you can see from this example, you specify the path down the tree structure to specify the nodes, separating each node with a forward slash (/) character.

You can also get all the `Customers` nodes using the following example, which uses `//` to specify the whole XML document as the context:

```
/Customers
```

The next example returns the `Customers` nodes and all their elements:

```
/NorthwindCustomers/Customers/*
```

**Note** The asterisk (\*) specifies all the elements.

The next example returns just the `CustomerID` element of the `Customers` nodes:

```
/NorthwindCustomers/Customers/CustomerID
```

You can find elements in a node by specifying a search within square brackets [ ]. The following example returns all the elements of the customer with a `CustomerID` of `ALFKI`:

```
/NorthwindCustomers/Customers[CustomerID="ALFKI"]/*
```

The following example returns the `CompanyName` of the customer with a `CustomerID` of `ALFKI`:

```
/NorthwindCustomers/Customers[CustomerID="ALFKI"]/CompanyName
```

You can also use square brackets to indicate the index of a node, starting at index 1. The following example returns the first `Customers` node:

```
/NorthwindCustomers/Customers[1]
```

You can use the `last()` function to get the last node. The following example returns the last `Customers` node:

```
/NorthwindCustomers/Customers[last()]
```

If your XML file contains embedded attributes rather than elements to hold values, then your XPath search expression is slightly different. [Listing 16.8](#) shows an XML file named `CustomersWithAttributes.xml` that uses attributes.

### **Listing 16.8: CUSTOMERSWITHATTRIBUTES.XML**

---

```
<?xml version="1.0"?>

<NorthwindCustomers>
  <Customers
    CustomerID="ALFKI"
    CompanyName="Alfreds Futterkiste"
    PostalCode="12209"
    Country="Germany"
    Phone="030-0074321"
  />
  <Customers
    CustomerID="ANATR"
    CompanyName="Ana Trujillo Emparedados y helados"
    PostalCode="05021"
    Country="Mexico"
    Phone="(5) 555-4729"
  />
</NorthwindCustomers>
```

---

To access an attribute you place an at (@) character at the start of the attribute name. The following example returns the `CustomerID` attribute of the `Customers` nodes:

```
/NorthwindCustomers/Customers/@CustomerID
```

The next example returns all the attributes of the customer with a CustomerID of ALFKI:

```
/NorthwindCustomers/Customers[@CustomerID="ALFKI"]/*
```

The following example returns the CompanyName of the customer with a CustomerID of ALFKI:

```
/NorthwindCustomers/Customers[@CustomerID="ALFKI"]/@CompanyName
```

**Note**

I've only touched on XPath expressions in this section. You can use many other mathematical operators, Boolean expressions, and much more. You can learn more about XPath in the SQL Server Books Online documentation and at the World Wide Web Consortium's (WC3) Web site at [www.w3.org](http://www.w3.org); just look for XPath in the table of contents.

Team LiB

◀ PREVIOUS

NEXT ▶

## Introducing XSLT

XML is a great way to represent data in a portable format, but XML doesn't contain information on how to format that data for display. The Extensible Stylesheet Language Transformation (XSLT) allows you to control the formatting of XML data, and may be used to transform XML data to a format suitable for displaying it as a document.

An XSL stylesheet—also known as an XSLT file—is a template that contains the rules that describe how the data in the XML file is to be formatted for viewing.

The XML and XSLT files are processed together by an XSLT processor. The rules defined in the XSLT file are applied to the data in the XML file, and the final result is output by the XSLT processor. Microsoft Internet Explorer contains an XSLT processor, and you'll see examples in this section that display the results of processing an XML and XSLT file in Internet Explorer.

**Note** Internet Explorer actually comes with a default XSLT file, which causes XML files to use different colors for the parts of the XML document and to be displayed with + and - icons to expand and collapse nested XML data.

[Listing 16.9](#) shows an example XSLT file named `CustomersStylesheet.xsl`, which you'll find in the `xml` directory. Later, you'll see how to apply this XSLT file to an XML file containing customer data.

### Listing 16.9: CUSTOMERSSTYLESHEET.XSL

---

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:template match="/">

<HTML>
<HEAD>
  <TITLE>Customers</TITLE>
</HEAD>
<BODY>
  <xsl:for-each select="/NorthwindCustomers/Customers">
    <p>
      <b>Customer:</b>
      <br><xsl:value-of select="CustomerID"/></br>
      <br><xsl:value-of select="CompanyName"/></br>
      <br><xsl:value-of select="PostalCode"/></br>
      <br><xsl:value-of select="Country"/></br>
      <br><xsl:value-of select="Phone"/></br>
    </p>
  </xsl:for-each>
</BODY>
</HTML>

</xsl:template>
</xsl:stylesheet>
```

---

As you can see, the `CustomersStylesheet.xsl` file contains HTML tags and `xsl` tags. The `xsl` tags are instructions that indicate how XML is to be transformed. You can reference this XSLT file in an XML file; the rules in the XSLT file are then applied to the data in the XML file. You'll see how to do that later in this section.

Let's take a closer look at the lines in the `CustomersStylesheet.xsl` file. This file starts with the following line, which indicates that the file uses XML version 1.0:

```
<?xml version="1.0"?>
```

The next lines are `xsl` tags:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:template match="/">
```

The first line uses the `xsl:stylesheet` tag and specifies that the <http://www.w3.org/1999/XSL/Transform> namespace is to be used. The second line uses the `xsl:template` tag and sets the match attribute to `/`, which specifies that the entire XML document is to be selected and used by the XSLT processor, starting at the root node.

**Note** You can set the match attribute to any XPath expression. For example, if you set match to `//Customers`, then all the `Customers` nodes would be selected.

The next lines are HTML tags, which start the HTML part of the file, define a header, and start the body:

```
<HTML>
<HEAD>
  <TITLE>Customers</TITLE>
</HEAD>
<BODY>
```

The next lines are the real meat of the XSLT file and use the `xsl:for-each` tag to iterate over the `Customers` nodes:

```
<xsl:for-each select="/NorthwindCustomers/Customers">
  <p>
    <b>Customer:</b>
    <br><xsl:value-of select="CustomerID"/><br>
    <br><xsl:value-of select="CompanyName"/><br>
    <br><xsl:value-of select="PostalCode"/><br>
    <br><xsl:value-of select="Country"/><br>
    <br><xsl:value-of select="Phone"/><br>
  </p>
</xsl:for-each>
```

The `xsl:value-of` tag is used to retrieve the element values from the XML file. For example, `<xsl:value-of select="CustomerID"/>` retrieves the `CustomerID` element.

The remaining lines close up the HTML and the `xsl` parts of the file:

```
</BODY>
</HTML>

</xsl:template>
</xsl:stylesheet>
```

The XSLT file contains only the rules to transform XML data; we still need to provide the XML data itself. [Listing 16.10](#) shows an XML file named `CustomersUsingStylesheet.xml`, which contains XML data for two customers.

### **Listing 16.10: CUSTOMERSUSINGSTYLESHEET.XML**

---

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="CustomersStylesheet.xsl"?>
<NorthwindCustomers>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkiste</CompanyName>
    <PostalCode>12209</PostalCode>
    <Country>Germany</Country>
    <Phone>030-0074321</Phone>
  </Customers>
  <Customers>
```

```

<CustomerID>ANATR</CustomerID>
<CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
<PostalCode>05021</PostalCode>
<Country>Mexico</Country>
<Phone>(5) 555-4729</Phone>
</Customers>
</NorthwindCustomers>

```

---

This listing is identical to the `Customers.xml` file shown earlier in [Listing 16.7](#) but with the addition of the following line that references the `CustomersStylesheet.xsl` file:

```
<?xml-stylesheet type="text/xsl" href="CustomersStylesheet.xsl"?>
```

This line causes the XSLT processor to read and apply the rules in the `CustomersStylesheet.xsl` file to the XML data in the `CustomersUsingStylesheet.xml` file. [Figure 16.5](#) shows how `CustomersUsingStylesheet.xml` looks when viewed with Internet Explorer. To view this file, right-click on `CustomersUsingStylesheet.xml` in Windows Explorer and select Open With >> Internet Explorer.



**Figure 16.5:** Viewing `CustomersUsing-Stylesheet.xml` in Internet Explorer

When you open `CustomersUsingStylesheet.xml`, Internet Explorer's XSLT processor opens the `CustomersStylesheet.xsl` file and applies the rules in it to the XML data in `CustomersUsingStylesheet.xml`. The output generated by the XSLT processor is then displayed in Internet Explorer.

I've only touched on using XSLT in this section. XSLT is a very powerful language that contains many functions you can use to format your XML data. For more information, see *Mastering XSLT* by Chuck White (Sybex, 2002). You can also learn more at the World Wide Web Consortium's Web site at [www.w3.org](http://www.w3.org); just look for XSLT in the table of contents.

## Accessing SQL Server Using HTTP

You can access SQL Server using HTTP (Hypertext Transfer Protocol). This allows you to run SQL statements from a browser. For example, you can run a `SELECT` statement that returns XML, and SQL Server will display the results in your browser. You can use XPath statements to locate data in the returned XML, and use XSL stylesheets to format the returned XML. I'll show you how to do all of these things in this section.

**Warning** You can even run `INSERT`, `UPDATE`, and `DELETE` statements—but you'll need to be careful about restricting the ability to run these types of statements because an errant user could easily wreck your database.

Before you can access SQL Server using HTTP, you'll need to configure SQL XML support for IIS (Internet Information Server).

## Configuring SQL XML Support for IIS

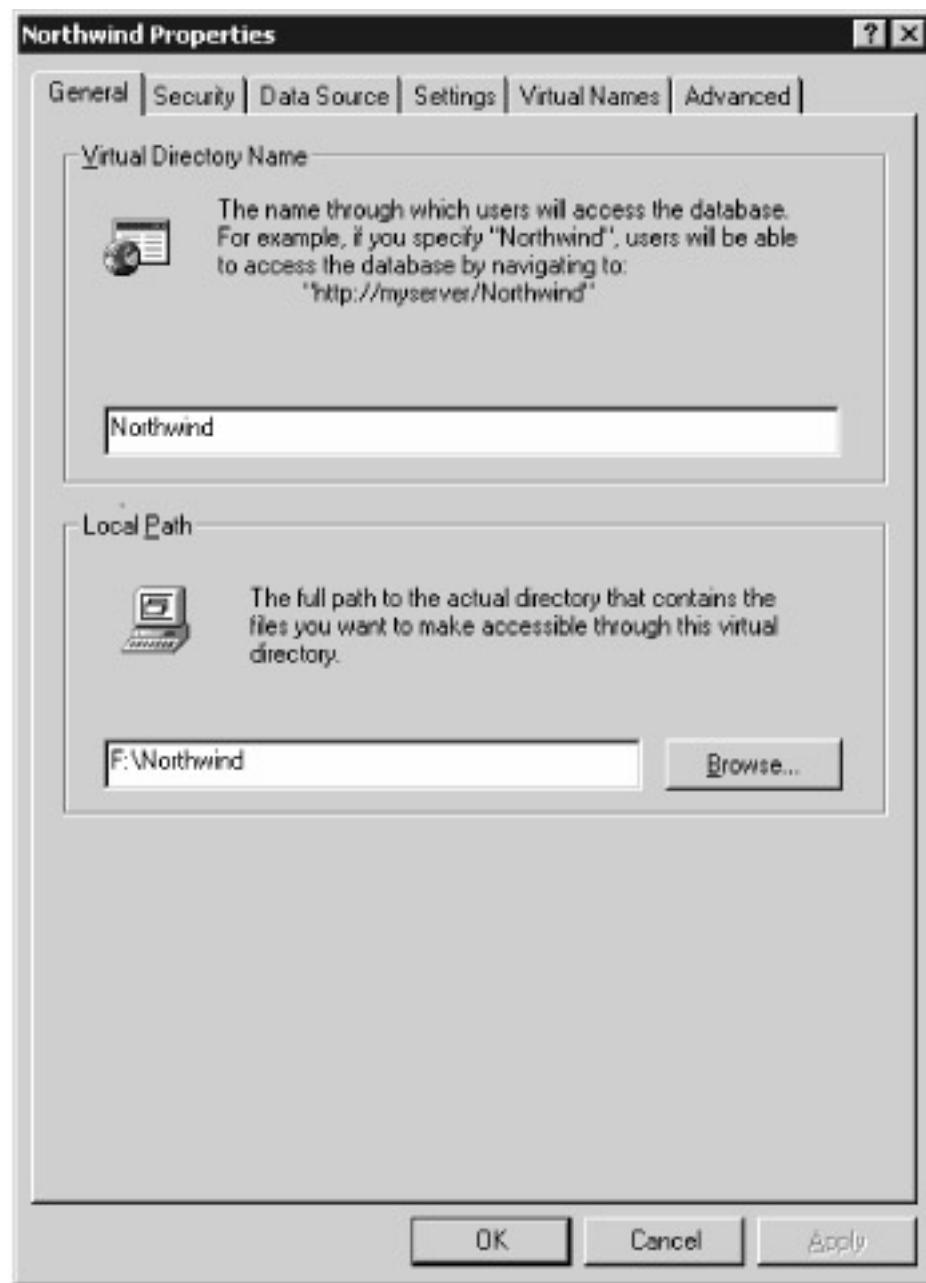
To configure SQL XML support for IIS, select Start > Programs > Microsoft SQL Server > Configure SQL XML Support in IIS. This starts the IIS Virtual Directory Management for SQL Server console, as shown in [Figure 16.6](#). You use this console to define a virtual directory through which you access SQL Server via HTTP.



**Figure 16.6:** The IIS Virtual Directory Management for SQL Server console

To define a virtual directory, expand the node for your computer using the + icon (I've expanded the node for my computer—which is named JMPRICE-DT1—in [Figure 16.6](#)).

Next, right-click on Default Web Site and select New > Virtual Directory from the pop-up menu. You'll need to set the properties for your virtual directory using the New Virtual Directory Properties window. This window contains six tabs, the first of which is named General, which you use to set your Virtual Directory Name (the name through which you access SQL Server) and Local Path (the actual directory in your computer's file system where you store files, such as XML and XSLT files). I've set my Virtual Directory Name to Northwind and my Local Path to F:\Northwind, as shown in [Figure 16.7](#).



**Figure 16.7:** Setting the Virtual Directory Name and Local Path

**Warning** The directory you specify for your Local Path must already exist in your computer's file system. Create it using Windows Explorer, and then browse to that directory using the Browse button.

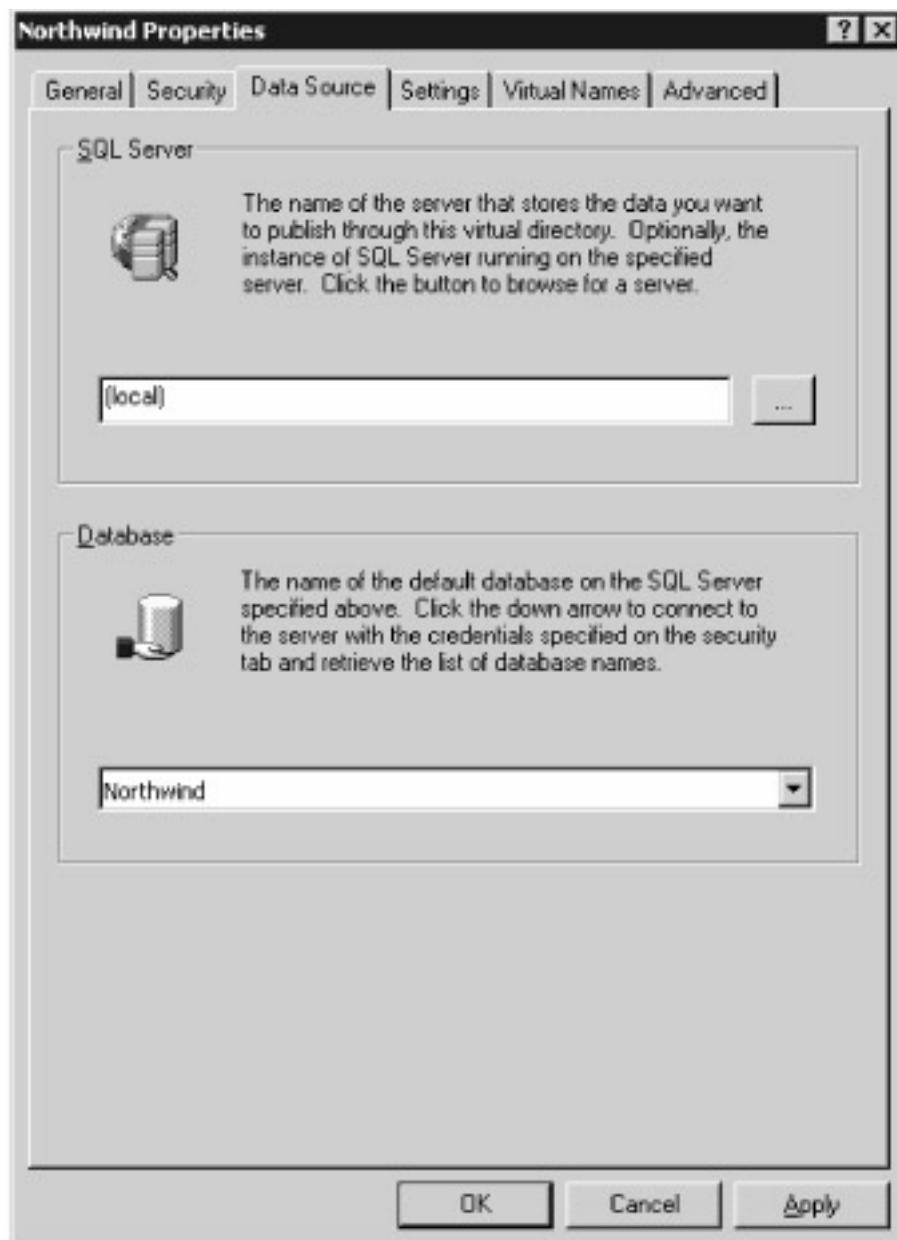
Next, you use the Security tab to set the details of how to authenticate the user when accessing SQL Server. I've used the `sa` SQL Server account, as shown in [Figure 16.8](#).



**Figure 16.8:** Setting the authentication details

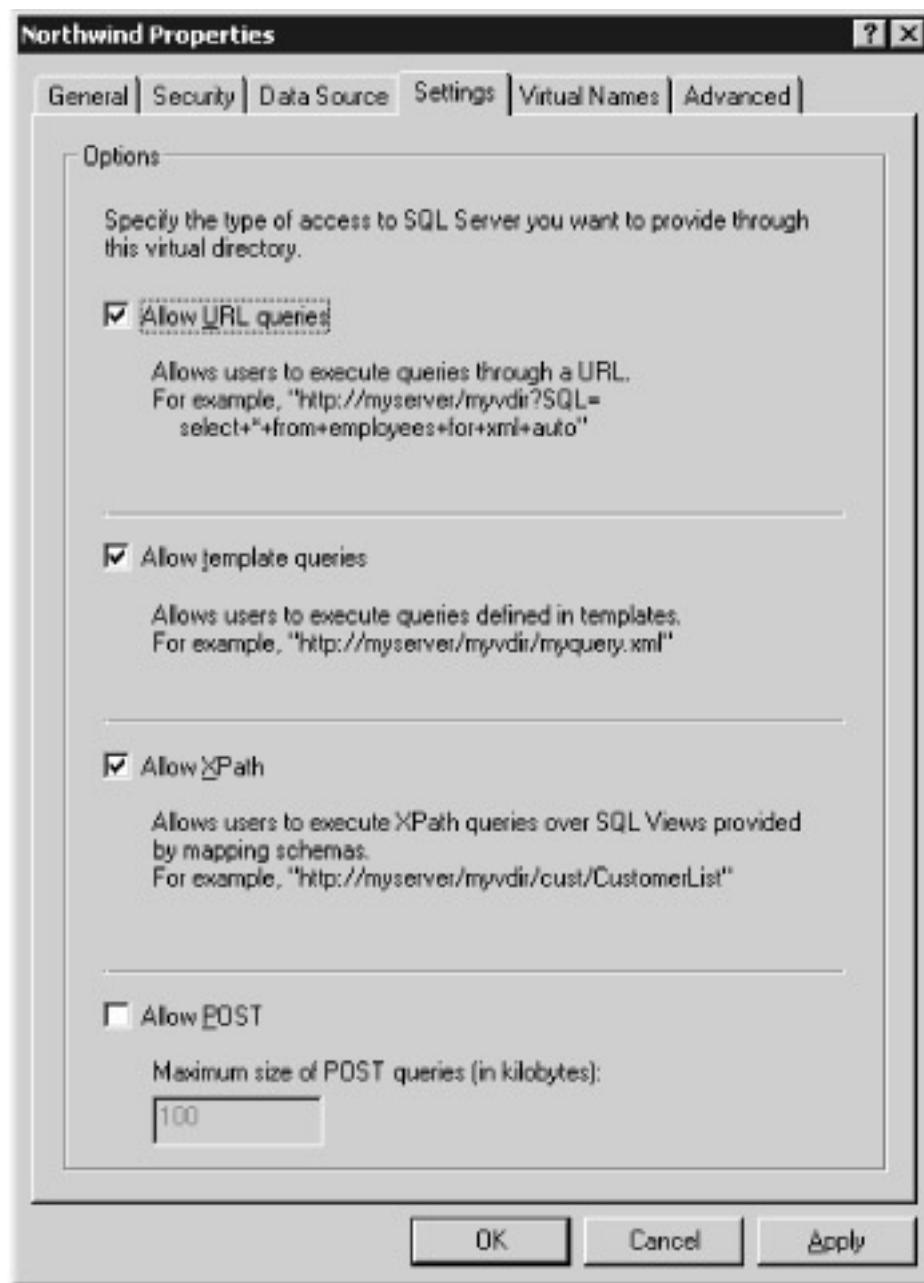
**Warning** In a production system, you'll want to use an account that has limited permissions in the database. For example, you'll probably want to grant read access only to tables.

Next, you use the Data Source tab to set which SQL Server you want to use, along with the database you want to access. I've picked the local SQL Server and the Northwind database, as shown in [Figure 16.9](#).



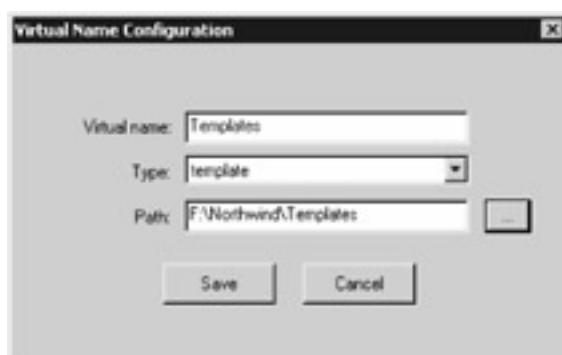
**Figure 16.9:** Setting the data source

Next, you use the Settings tab to specify the type of access to SQL Server you want to provide. Check the following boxes: Allow URL Queries (allows direct execution of SQL statements), Allow Template Queries (allows the use of XML and XSLT files to retrieve and format results from the database), and Allow XPath Queries (allows execution of queries with XPath expressions), as shown in [Figure 16.10](#).

**Figure 16.10:** Setting the type of access

**Warning** In a production system, you'll want to restrict access to Allow Template Queries only. That way, users can execute only queries defined in an XML template file.

Next, you use the Virtual Names tab to map a database schema, a template directory containing XML and XSLT files, or a database object (dbobject) to a path relative to your virtual directory. Click the New button and set your Virtual Name to Templates, the Type to template, and your Path to a subdirectory named `Templates` in your `Northwind` directory , as shown in [Figure 16.11](#). You'll need to create the `Templates` folder first.

**Figure 16.11:** Setting the virtual name configuration

**Warning** The Templates subdirectory you specify in your Path must already exist in your computer's file system. Create it using Windows Explorer, and then browse to that directory using the ellipsis (...) button to the right of the Path field.

Click Save to continue. You won't be changing anything in the Advanced tab, but feel free to examine it if you want to. Click OK to save your settings across all the tabs. Your new virtual directory is then created and will appear in the IIS Virtual Directory Management for SQL Server console.

## Running Direct SQL Statements Using a Browser

In this section, you'll learn how to run direct SQL statements using a browser. I'll be using Internet Explorer in the examples, but you can use whatever browser you wish.

### Running **SELECT** Statements

In this section, you'll see how to run a **SELECT** statement. For example, point your browser to the following URL, which contains an embedded **SELECT** statement:

```
http://localhost/Northwind?sql=SELECT+*+FROM+Customers+WHERE+CustomerID+IN+( 'ALFKI '
, 'ANATR ' )+FOR+XML+AUTO&root=ROOT
```

As you can see, the **SELECT** statement in this URL retrieves two rows from the **Customers** table. The first part of the URL is

`http://localhost/Northwind`

This contains the name of the server (`localhost`) and the virtual directory (`Northwind`). The second part of the URL is

```
?sql=SELECT+*+FROM+Customers+WHERE+CustomerID+IN+( 'ALFKI ' , 'ANATR ' )+FOR+XML+
AUTO&root=ROOT
```

This contains the embedded **SELECT** statement. Because URLs don't allow spaces, you use plus (+) characters instead. The `root` parameter at the end of the URL supplies a name for the root element in the XML returned by the **SELECT** statement; I've supplied a root name of `ROOT` in the previous example, but you can use whatever name you want. [Figure 16.12](#) shows the result of running the **SELECT** statement in Internet Explorer.



**Figure 16.12:** Selecting customers and displaying results

**Warning** If you omit the `root` parameter in your URL, then you'll get the following error: *Only one top level element is allowed in an XML document.*

Spaces aren't the only characters you'll need to replace in your URL. [Table 16.4](#) shows some of the special characters you might use in a SQL statement and the replacement you use in your URL.

**Table 16.4: SPECIAL CHARACTERS IN A SQL STATEMENT AND THEIR REPLACEMENTS IN A URL**

CHARACTER IN SQL STATEMENT	REPLACEMENT IN URL
Space	+
/	%2F
?	%3F
%	%25
#	%23
&	%26

For example, if you wanted to use `LIKE 'C%'` in your `SELECT` statement, then you would use `LIKE+'C%25'`, as shown in the following URL:

```
http://localhost/Northwind?sql=SELECT+*+FROM+Customers+WHERE+CompanyName+LIKE+'C%25'+FOR+XML+AUTO&root=ROOT
```

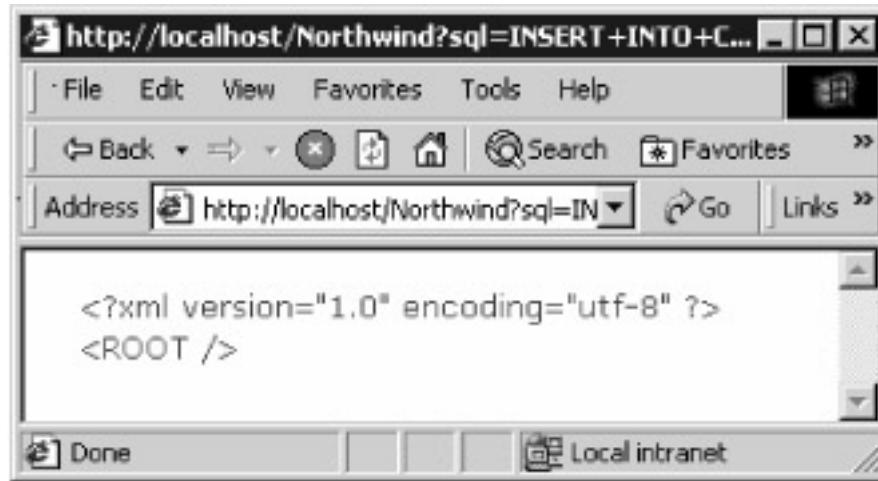
The `SELECT` statement in this URL retrieves the rows from the `Customers` table that has a `CompanyName` starting with C.

## Running *INSERT*, *UPDATE*, and *DELETE* Statements

You can embed SQL `INSERT`, `UPDATE`, and `DELETE` statements in a URL. The following example uses an `INSERT` statement to add a new row to the `Customers` table:

```
http://localhost/Northwind?sql=INSERT+INTO+Customers+(CustomerID,CompanyName)+VALUES+( 'J9COM' , 'J9+Company' )&root=ROOT
```

[Figure 16.13](#) shows the result of running this `INSERT` statement in Internet Explorer.



**Figure 16.13:** Adding a new row to the `Customers` table

The next example uses a `DELETE` statement to remove the new row:

```
http://localhost/Northwind?sql=DELETE+FROM+Customers+WHERE+CustomerID='J9COM'&root=ROOT
```

**Warning** You'll almost certainly want to prevent users from running *INSERT*, *UPDATE*, and *DELETE* statements over HTTP on your production server. You can do this by preventing users from running direct SQL statements, as described in the [previous section](#), or by restricting the permissions assigned to the database user. You could also allow access to the database using only stored procedures; you'll see how to run a stored procedure using a URL in the [next section](#).

## Running Stored Procedures

You can also run stored procedures from a URL. [Listing 16.11](#) contains a script that creates a stored procedure named *CustomersFromCountry()*. This procedure retrieves the rows from the *Customers* table with a *Country* matching the *@MyCountry* parameter that is passed to *CustomersFromCountry()*.

### **Listing 16.11: CUSTOMERSFROMCOUNTRY.SQL**

```
/*
CustomersFromCountry.sql creates a procedure that
retrieves rows from the Customers table whose
Country matches the @MyCountry parameter
*/
CREATE PROCEDURE CustomersFromCountry
    @MyCountry nvarchar(15)
AS
    SELECT *
    FROM Customers
    WHERE Country = @MyCountry
    FOR XML AUTO
```

You run this stored procedure using the following URL:

`http://localhost/Northwind?sql=EXECUTE+CustomersFromCountry+@MyCountry='UK'&root=ROOT`

[Figure 16.14](#) shows the result of running the stored procedure.



**Figure 16.14:** Running a stored procedure

## Running SQL Statements Using an XML Template

You can also execute SQL statements using an XML template, which is just an XML file containing your embedded SQL statement. [Listing 16.12](#) shows an example file named *Customers.xml* that contains an embedded SELECT statement.

### **Listing 16.12: CUSTOMERS.XML**

```
<?xml version="1.0"?>

<Northwind xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:query>
    SELECT TOP 2 CustomerID, CompanyName, City, Country
    FROM Customers
    ORDER BY CustomerID
    FOR XML AUTO, ELEMENTS
  </sql:query>
</Northwind>
```

**Note** You'll find the *Customers.xml* file-and the other XML and XSLT files used in the [next section](#)-in the *xml\Northwind\Templates* directory. You'll need to copy these files into the *Templates* directory you set up earlier for your SQL Server virtual directory.

Notice that the SELECT statement is placed within `sql:query` and `/sql:query` tags. The outer `Northwind` tag is the root node for the XML.

To run the *Customers.xml* file, point your browser to the following URL:

`http://localhost/Northwind/Templates/Customers.xml`

[Figure 16.15](#) shows the result of running the *Customers.xml* file in Internet Explorer.



**Figure 16.15:** Running the *Customers.xml* file

### **Formatting XML Output Using an XSL Stylesheet**

As you'll learn in this section, you can format the XML output generated by SQL Server using an XSL stylesheet. Specifically, you'll see how to format the XML shown earlier in [Figure 16.14](#). [Listing 16.13](#) shows an XSL stylesheet file named *CustomersStylesheet.xsl*.

### **Listing 16.13: CUSTOMERSSTYLESHEET.XSL**

```

<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:template match="/">

<HTML>
<HEAD>
  <TITLE>Customers</TITLE>
</HEAD>
<BODY>
  <xsl:for-each select="Northwind/Customers">
    <p>
      <b>Customer:</b>
      <br><xsl:value-of select="CustomerID"/><br>
      <br><xsl:value-of select="CompanyName"/><br>
      <br><xsl:value-of select="PostalCode"/><br>
      <br><xsl:value-of select="Country"/><br>
      <br><xsl:value-of select="Phone"/><br>
    </p>
  </xsl:for-each>
</BODY>
</HTML>

</xsl:template>
</xsl:stylesheet>

```

---

Notice that the select XPath expression in the `xsl:for-each` tag is set to `Northwind/Customers`. `Northwind` is the root node from the generated XML, and `Customers` are the child nodes from the root. Therefore, this XPath expression selects all the `Customers` nodes from any XML generated by SQL Server.

[Listing 16.14](#) shows an XML file named `CustomersUsingStylesheet.xml`, which uses the `CustomersStylesheet.xsl` file. `CustomersUsingStylesheet.xml` retrieves the top two rows from the `Customers` table.

#### **Listing 16.14: CUSTOMERSUSINGSTYLESHEET.XML**

---

```

<?xml version="1.0"?>

<Northwind xmlns:sql="urn:schemas-microsoft-com:xml-sql"
sql:xsl="CustomersStylesheet.xsl">
  <sql:query>
    SELECT TOP 2 CustomerID, CompanyName, PostalCode, Country, Phone
    FROM Customers
    ORDER BY CustomerID
    FOR XML AUTO, ELEMENTS
  </sql:query>
</Northwind>

```

---

To run the `CustomersUsingStylesheet.xml` file, point your browser to the following URL:

`http://localhost/Northwind/Templates/CustomersUsingStylesheet.xml?contenttype=text/html`

Notice that the `contenttype` parameter at the end of this URL is set to `text/html`, which indicates that the content is to be interpreted as HTML.

**Warning** If you omit the `contenttype` parameter, then you'll get the following error: `End tag 'HEAD' does not match the start tag 'META'.`

Figure 16.16 shows the result of running the `CustomersUsingStylesheet.xml` file in IE. Notice that the output is formatted using the rules defined in the `CustomersStylesheet.xsl` file.



Figure 16.16: Running the `CustomersUsing-Stylesheet .xml` file

## Using the SQL Server *OPENXML()* Function

SQL Server contains a function named `OPENXML( )` that allows you to read XML data as if it were a result set of rows. One use of `OPENXML( )` is to read XML data as rows, and then insert those rows into a table.

In this section, you'll explore the syntax of `OPENXML( )`. You'll also see an example that reads XML data containing details of two customers using `OPENXML( )`, and then you'll insert two new rows into the `Customers` table using the values from that XML data.

### ***OPENXML()* Syntax**

The `OPENXML( )` function uses the following syntax

```
OPENXML(XmlDocumentHandle int [IN], RowPattern nvarchar [IN], [Flags byte[IN]])  
[WITH (SchemaDeclaration | TableName)]
```

where

***XmlDocumentHandle*** specifies an `int` handle to your XML document. You use this handle as a reference to your XML document.

***RowPattern*** specifies an XPath expression to select the data you require from your XML document.

***Flags*** specifies an optional `byte` value that you use to indicate the mapping between your XML data and the database column values. A value of 1 indicates that your XML data being read stores column values in embedded attributes of the nodes ([Listing 16.8](#), shown earlier, illustrates embedded attributes); this is the default. A value of 2 indicates that your XML data stores column values as separate nested elements ([Listing 16.7](#), shown earlier, illustrates nested elements). The values from your XML file are then used as column values in the rows returned by `OPENXML( )`.

***SchemaDeclaration*** specifies the definition of the database schema you want to use to return rows as. An example definition is `CustomerID nvarchar(5), CompanyName nvarchar(40)`. You use either ***SchemaDeclaration*** or ***TableName***.

***TableName*** specifies the name of the database table you want to use. You'll typically use ***TableName*** rather than ***SchemaDeclaration*** when you're working with a table that already exists in the database.

### **Using *OPENXML()***

Before calling `OPENXML( )`, you must first call the `sp_xml_preparedocument( )` procedure. This procedure parses your XML document and prepares a copy of that document in memory. You then use that copy of the XML document with `OPENXML( )`. Once you've completed your call to `OPENXML( )` you call the `sp_xml_removedocument( )` procedure to remove the XML document from memory.

The example in this section uses a stored procedure named `AddCustomersXml( )` to read the XML data containing details of two customers using `OPENXML( )` and to insert two new rows into the `Customers` table using the values from that XML data. [Listing 16.15](#) shows a script named `AddCustomersXml.sql` that creates the `AddCustomersXml( )` stored procedure.

---

#### ***Listing 16.15: ADDCUSTOMERSXML.SQL***

```
/*  
AddCustomersXml.sql creates a procedure that uses OPENXML()  
to read customers from an XML document and then inserts them  
into the Customers table  
*/  
  
CREATE PROCEDURE AddCustomersXml  
    @MyCustomersXmlDoc nvarchar(4000)  
AS
```

```

- declare the XmlDocumentId handle
DECLARE @XmlDocumentId int

- prepare the XML document
EXECUTE sp_xml_preparedocument @XmlDocumentId OUTPUT, @MyCustomersXmlDoc

- read the customers from the XML document using OPENXML()
- and insert them into the Customers table
INSERT INTO Customers
SELECT *
FROM OPENXML(@XmlDocumentId, N'/Northwind/Customers', 2)
WITH Customers

- remove the XML document from memory
EXECUTE sp_xml_removedocument @XmlDocumentId

```

---

OPENXML( ) reads the XML from the document specified by the handle @XmlDocumentId and returns the rows to the INSERT statement. These rows are then added to the Customers table by the INSERT statement.

[Listing 16.16](#) shows a script named RunAddCustomers.sql that runs the AddCustomersXml( ) procedure.

### **Listing 16.16: RUNADDCUSTOMERS.SQL**

---

```

/*
RunAddCustomersXml.sql runs the AddCustomersXml() procedure
*/

- define the XML document
DECLARE @NewCustomers nvarchar(4000)
SET @NewCustomers = N'
<Northwind>
    <Customers>
        <CustomerID>T1COM</CustomerID>
        <CompanyName>Test 1 Company</CompanyName>
    </Customers>
    <Customers>
        <CustomerID>T2COM</CustomerID>
        <CompanyName>Test 2 Company</CompanyName>
    </Customers>
</Northwind>

- run the AddCustomersXml() procedure
EXECUTE AddCustomersXml @MyCustomersXmlDoc=@NewCustomers

- display the new rows
SELECT CustomerID, CompanyName
FROM Customers
WHERE CustomerID IN ('T1COM', 'T2COM')

- delete the new rows
DELETE FROM Customers
WHERE CustomerID IN ('T1COM', 'T2COM')

```

---

[Figure 16.17](#) shows the result of running the RunAddCustomers.sql script in Query Analyzer.

## Using the SQL Server OPENXML() Function

The screenshot shows the SQL Query Analyzer interface. The title bar reads "SQL Query Analyzer" and "File Edit Query Tools Window Help". The query window displays the following script:

```
-- RunAddCustomersXml.sql runs the AddCustomersXml() procedure
-- define the XML document
DECLARE @NewCustomers nvarchar(4000)
SET @NewCustomers = N'
<Northwind>
  <Customers>
    <CustomerID>T1COM</CustomerID>
    <CompanyName>Test 1 Company</CompanyName>
  </Customers>
</Northwind>'
```

The results pane shows the output of the script:

CustomerID	CompanyName
T1COM	Test 1 Company
T2COM	Test 2 Company

Below the results, the status bar indicates "Query batch completed, localhost (8.0) sa (52) Northwind 0:00:00 2 rows In 29, Col 29".

**Figure 16.17:** Running the RunAddCustomers .sql script

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

## Using an *XmlDocument* Object to Store an XML Document

You use an object of the  `XmlDocument` class to represent an XML document in a C# program. An  `XmlDocument` object stores the nodes of the XML document in objects of the  `XmlNode` class. You can, for example, load rows from the database into a  `DataSet` object, and then load an XML representation of those rows into an  `XmlDocument` object. [Table 16.5](#) shows some of the  `XmlDocument` properties; [Table 16.6](#) shows some of the  `XmlDocument` methods; and [Table 16.7](#) shows the  `XmlDocument` events.

**Table 16.5:  `XmlDocument` Properties**

Property	Type	Description
Attributes	<code>XmlAttributeCollection</code>	Gets the <code>XmlAttributeCollection</code> object that contains the attributes of the current node.
BaseURI	<code>string</code>	Gets the base URI of the current node.
ChildNodes	<code>XmlNodeList</code>	Gets all the child nodes of the node.
DocumentElement	<code>XmlElement</code>	Gets the root <code>XmlElement</code> object for the XML document.
DocumentType	<code>XmlDocumentType</code>	Gets the node containing the DOCTYPE declaration.
FirstChild	<code>XmlNode</code>	Gets the first child of the node.
HasChildNodes	<code>bool</code>	Gets a <code>bool</code> that indicates whether this node has any child nodes.
Implementation	<code>XmlImplementation</code>	Gets the <code>XmlImplementation</code> object for the XML document.
InnerText	<code>string</code>	Gets or sets the concatenated values of the node and all of its children.
InnerXml	<code>string</code>	Gets or sets the XML that represents the children of the current node.
IsReadOnly	<code>bool</code>	Gets a <code>bool</code> value that indicates whether the current node is read-only.
LastChild	<code>XmlNode</code>	Gets the last child of the node.
LocalName	<code>string</code>	Gets the local name of the node.
Name	<code>string</code>	Gets the qualified name of the node.
NamespaceURI	<code>string</code>	Gets the namespace URI of the node.
NameTable	<code>XmlNameTable</code>	Gets the <code>XmlNameTable</code> object associated with the XML implementation.
NextSibling	<code>XmlNode</code>	Gets the node immediately following the current node.
NodeType	<code>XmlNodeType</code>	Gets the type of the current node.
OuterXml	<code>string</code>	Gets the XML that represents the current node and all of its children.
OwnerDocument	<code> XmlDocument</code>	Gets the <code> XmlDocument</code> object that the current node belongs to.
ParentNode	<code>XmlNode</code>	Gets the parent of the current node.

Prefix	string	Gets or sets the namespace prefix of the current node.
PreserveWhitespace	bool	Gets or sets a bool value that indicates whether white space is to be preserved when XML is loaded or saved. The default is <code>false</code> .
PreviousSibling	XmlNode	Gets the node immediately preceding the current node.
Value	string	Gets or sets the value of the current node.
XmlResolver	XmlResolver	Sets the <code>XmlResolver</code> object to use for resolving external resources.

**Table 16.6: XmlDocument Methods**

Method	Return Type	Description
AppendChild()	XmlNode	Adds the specified node to the end of child nodes.
CloneNode()	XmlNode	Creates a duplicate of the node.
CreateAttribute()	XmlAttribute	Creates an <code>XmlAttribute</code> object of the specified name.
CreateCDataSection()	XmlCDataSection	Creates an <code>XmlCDataSection</code> object with the specified data.
CreateComment()	XmlComment	Creates an <code>XmlComment</code> object with the specified data.
CreateDocumentFragment()	XmlDocumentFragment	Creates an <code>XmlDocumentFragment</code> object with the specified data.
CreateDocumentType()	XmlDocumentType	Creates a new <code>XmlDocumentType</code> object with the specified data.
CreateElement()	XmlElement	Overloaded. Creates an <code>XmlElement</code> object.
CreateEntityReference()	XmlEntityReference	Creates an <code>XmlEntityReference</code> object with the specified name.
CreateNavigator()	XpathNavigator	Creates an <code>XpathNavigator</code> object that you can use to navigate the XML document.
CreateNode()	XmlNode	Overloaded. Creates an <code>XmlNode</code> object.
CreateTextNode()	XmlText	Creates an <code>XmlText</code> object with the specified text.
CreateWhitespace()	XmlWhitespace	Creates an <code>XmlWhitespace</code> object.
CreateXmlDeclaration()	XmlDeclaration	Creates an <code>XmlDeclaration</code> object.
GetElementById()	XmlElement	Gets the <code>XmlElement</code> object with the specified ID.
GetElementsByTagName()	XmlNodeList	Overloaded. Returns an <code>XmlNodeList</code> object that contains a list of all descendant elements that match the specified name.

GetNamespaceOfPrefix()	string	Looks up the closest <code>xmlns</code> declaration with the specified prefix that is in scope for the current node, and then returns the namespace URI.
GetPrefixOfNamespace()	string	Looks up the closest <code>xmlns</code> declaration with the specified namespace URI that is in scope for the current node, and then returns the prefix.
ImportNode()	XmlNode	Imports a node from another XML document into the current XML document.
InsertAfter()	XmlNode	Inserts the specified node immediately after the specified reference node.
InsertBefore()	XmlNode	Inserts the specified node immediately before the specified reference node.
Load()	void	Overloaded. Loads XML data into your <code> XmlDocument</code> object.
LoadXml()	void	Loads the XML document from the specified string into your <code> XmlDocument</code> object.
PrependChild()	XmlNode	Adds the specified node to the beginning of the child nodes.
ReadNode()	XmlNode	Creates an <code> XmlNode</code> object based on the information in a specified <code> XmlReader</code> object. Your <code> XmlReader</code> must be positioned on a node or attribute.
RemoveAll()	void	Removes all the children and attributes of the current node.
RemoveChild()	XmlNode	Removes the specified child node.
ReplaceChild()	XmlNode	Replaces one child node with another.
Save()	void	Overloaded. Saves the XML document to the specified location.
SelectNodes()	XmlNodeList	Overloaded. Selects a list of nodes matching the specified XPath expression.
SelectSingleNode()	XmlNode	Overloaded. Selects the first <code> XmlNode</code> that matches the specified XPath expression.
WriteContentTo()	void	Saves all the children of the XML document to the specified <code> XmlWriter</code> object.
WriteTo()	void	Saves the XML document to the specified <code> XmlWriter</code> object.

**Table 16.7:  `XmlDocument` Events**

Event	Event Handler	Description
NodeChanging	<code> XmlNodeChangedEventHandler</code>	Fires before a value in a node is changed.
NodeChanged	<code> XmlNodeChangedEventHandler</code>	Fires after a value in a node is changed.
NodeInserting	<code> XmlNodeChangedEventHandler</code>	Fires before a node is inserted.

NodeInserted	XmlNodeChangedEventHandler	Fires after a node is inserted.
NodeRemoving	XmlNodeChangedEventHandler	Fires before a node is removed.
NodeRemoved	XmlNodeChangedEventHandler	Fires after a node is removed.

[Listing 16.17](#) shows a program that illustrates the use of an `XmlDocument` object. This program performs the following steps:

- Creates a `DataSet` object named `myDataSet` and fills it with the top two rows from the `Customers` table.
- Creates an  `XmlDocument` object named `myXmlDocument`, and then loads it with the XML from `myDataSet`. You can use the `GetXml()` method to return the customer rows in `myDataSet` as a string containing a complete XML document. You can then use the output string from `GetXml()` as the input to the `LoadXml()` method of `myXmlDocument`; this loads `myXmlDocument` with the XML document containing the customer details.
- Displays the XML in `myXmlDocument` using the `Save()` method, passing `Console.Out` to the `Save()` method. This results in the XML document being displayed on the screen.
- Retrieves the `XmlNode` objects in `myXmlDocument` using the `SelectNodes()` method, and then displays the text contained in the child nodes of each `XmlNode` using the `InnerText` property. You pass an XPath expression to `SelectNodes()` to retrieve the required nodes.
- Retrieves the `XmlNode` for the ANATR customer using the `SelectSingleNode()` method, and displays the text contained in the child nodes of this `XmlNode`. You pass an XPath expression to `SelectSingleNode()` to retrieve the required node.

### **Listing 16.17: USINGXMLDOCUMENT.CS**

---

```
/*
 Using XmlDocument.cs illustrates the use of an XmlDocument object
 */

using System;
using System.Data;
using System.Data.SqlClient;
using System.Xml;

class Using XmlDocument
{
    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 2 CustomerID, CompanyName, Country "+
            "FROM Customers "+
            "ORDER BY CustomerID";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;

        // step 1: create a DataSet object and fill it with the top 2 rows
        // from the Customers table
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        mySqlDataAdapter.Fill(myDataSet, "Customers");
        mySqlConnection.Close();
        // step 2: create an XmlDocument object and load it with the XML from

```

## Using an XmlDocument Object to Store an XML Document

```
// the DataSet; the GetXml() method returns the rows in
// myDataSet as a string containing a complete XML document; and
// the LoadXml() method loads myXmlDocument with the XML document
// string returned by GetXml()
XmlDocument myXmlDocument = new XmlDocument();
myXmlDocument.LoadXml(myDataSet.GetXml());

// step 3: display the XML in myXmlDocument using the Save() method
Console.WriteLine("Contents of myXmlDocument:");
myXmlDocument.Save(Console.Out);

// step 4: retrieve the XmlNode objects in myXmlDocument using the
// SelectNodes() method; you pass an XPath expression to SelectNodes()
Console.WriteLine("\n\nCustomers:");
foreach (XmlNode myXmlNode in
    myXmlDocument.SelectNodes("/NewDataSet/Customers"))
{
    Console.WriteLine("CustomerID = " +
        myXmlNode.ChildNodes[0].InnerText);
    Console.WriteLine("CompanyName = " +
        myXmlNode.ChildNodes[1].InnerText);
    Console.WriteLine("Country = " +
        myXmlNode.ChildNodes[2].InnerText);
}

// step 5: retrieve the XmlNode for the ANATR customer using
// the SelectSingleNode() method; you pass an XPath
// expression to SelectSingleNode
Console.WriteLine("\nRetrieving node with CustomerID of ANATR");
XmlNode myXmlNode2 =
    myXmlDocument.SelectSingleNode(
        "/NewDataSet/Customers[CustomerID='ANATR']");
Console.WriteLine("CustomerID = " +
    myXmlNode2.ChildNodes[0].InnerText);
Console.WriteLine("CompanyName = " +
    myXmlNode2.ChildNodes[1].InnerText);
Console.WriteLine("Country = " +
    myXmlNode2.ChildNodes[2].InnerText);
}
```

---

Remember, you'll need to change the connection string for your `SqlConnection` object to connect to your database near the start of this program.

The output from this program is as follows:

```
Contents of myXmlDocument:
<?xml version="1.0" encoding="IBM437"?>
<NewDataSet>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkiste</CompanyName>
    <Country>Germany</Country>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
    <Country>Mexico</Country>
```

```
</Customers>
</NewDataSet>

Customers:
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
Country = Germany
CustomerID = ANATR
CompanyName = Ana Trujillo Emparedados y helados
Country = Mexico
```

```
Retrieving node with CustomerID of ANATR
CustomerID = ANATR
CompanyName = Ana Trujillo Emparedados y helados
Country = Mexico
```

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

## Using an `XmlDataDocument` Object to Store an XML Document

In the [previous section](#), you saw how you use an  `XmlDocument` object to store an XML document containing customer details retrieved from a  `DataSet`. That's fine, but wouldn't it be great if you could combine the power of an  `XmlDocument` with a  `DataSet`? Well, you can! That's where the  `XmlDataDocument` class comes in.

You use an object of the  `XmlDataDocument` class to access rows as both  `XmlNode` objects and relational  `DataRow` objects. You associate a  `DataSet` with your  `XmlDataDocument` by passing your  `DataSet` to the  `XmlDataDocument` constructor.

An  `XmlDataDocument` object provides synchronization between the  `DataSet` and the XML document. For example, if you add a new customer as an  `XmlNode` object to your  `XmlDataDocument`, then that customer is also added as a  `DataRow` to your associated  `DataSet`. Similarly, if you add a new customer as a  `DataRow` to your  `DataSet`, then that customer is also added as an  `XmlNode` object in the XML document of the  `XmlDataDocument`. Also, if you update or delete a customer, then that change is made in both the  `DataSet` and the  `XmlDataDocument`. You'll see examples of synchronization shortly.

The  `XmlDataDocument` class is derived from the  `XmlDocument` class; therefore the  `XmlDataDocument` class inherits all the public properties, methods, and events shown in the [previous section](#) for the  `XmlDocument` class. The  `DataSet` property (type  `DataSet`) is the property added to the  `XmlDataDocument` class. It gets the  `DataSet` object, which stored the relational representation of the data. You associate a  `DataSet` with your  `XmlDataDocument` by passing the  `DataSet` to the  `XmlDataDocument` constructor. [Table 16.8](#) shows the additional  `XmlDataDocument` methods.

**Table 16.8:  `XmlDataDocument` Methods**

Method	Return Type	Description
<code>GetElementFromRow( )</code>	<code>XmlElement</code>	Returns the <code>XmlElement</code> object associated with the specified <code>DataRow</code> object.
<code>GetRowFromElement( )</code>	<code>DataRow</code>	Returns the <code>DataRow</code> object associated with the specified <code>XmlElement</code> object.
<code>Load( )</code>	<code>void</code>	Overloaded. Loads information from the specified data source into the <code> XmlDataDocument</code> object and synchronizes the loaded data with the <code> DataSet</code> .

[Listing 16.18](#) shows a program that illustrates the use of an  `XmlDataDocument`. This program performs the following steps:

- Creates a  `DataSet` object named `myDataSet` and fills it with a  `DataTable` named `customersDT` that contains the top two rows from the `Customers` table.
- Display the  `DataRow` objects in `customersDT` using the `DisplayDataRows( )` method, which is defined near the start of the program.
- Creates an  `XmlDataDocument` object named `myXDD`, passing `myDataSet` to the constructor; this associates `myDataSet` with the  `XmlDataDocument`.
- Displays the XML document in `myXDD` by passing `Console.Out` to the `Save( )` method.
- Adds a customer  `DataRow` with a `CustomerID` of `J9COM` to `customersDT`.
- Retrieves the `J9COM` node using the `GetElementFromRow( )` method. This method accepts a  `DataRow` as a parameter and returns the associated  `XmlNode`.
- Sets the `J9COM` node's `Country` to `USA`, first setting the `myDataSet` object's `EnforceConstraints` property to `false`-which you must do before making any changes to nodes.

8. Retrieves the ANATR XmlNode using SelectSingleNode( ).
9. Retrieves the ANATR DataRow using GetRowFromElement( ). This method accepts an XmlElement as a parameter and returns the associated DataRow.
10. Removes the ANATR node using RemoveAll( ).
11. Display the XML document in myXDD using Save( ).
12. Display the DataRow objects in customersDT using DisplayDataRows( ).

### **Listing 16.18: USINGXMLDATADOCUMENT.CS**

---

```
/*
UsingXmlDataDocument.cs illustrates how to use an XmlDocument
object
*/

using System;
using System.Data;
using System.Data.SqlClient;
using System.Xml;

class UsingXmlDataDocument
{
    public static void DisplayDataRows(DataTable myDataTable)
    {
        Console.WriteLine("\n\nCustomer DataRow objects in customersDT:");
        foreach (DataRow myDataRow in myDataTable.Rows)
        {
            foreach (DataColumn my DataColumn in myDataTable.Columns)
            {
                Console.WriteLine(my DataColumn + " = " +
                    myDataRow[my DataColumn]);
            }
        }
    }

    public static void Main()
    {
        SqlConnection mySqlConnection =
            new SqlConnection(
                "server=localhost;database=Northwind;uid=sa;pwd=sa"
            );
        SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
        mySqlCommand.CommandText =
            "SELECT TOP 2 CustomerID, CompanyName, Country " +
            "FROM Customers " +
            "ORDER BY CustomerID";
        SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
        mySqlDataAdapter.SelectCommand = mySqlCommand;
        // step 1: create a DataSet object and fill it with the top 2 rows
        // from the Customers table
        DataSet myDataSet = new DataSet();
        mySqlConnection.Open();
        mySqlDataAdapter.Fill(myDataSet, "Customers");
        mySqlConnection.Close();
        DataTable customersDT = myDataSet.Tables["Customers"];

        // step 2: display the DataRow objects in customersDT using
    }
}
```

```
// DisplayDataRows()
DisplayDataRows(customersDT);

// step 3: create an XmlDataDocument object, passing myDataSet
// to the constructor; this associates myDataSet with the
// XmlDataDocument
XmlDataDocument myXDD = new XmlDataDocument(myDataSet);

// step 4: display the XML document in myXDD
Console.WriteLine("\nXML document in myXDD:");
myXDD.Save(Console.Out);

// step 5: add a customer DataRow to customersDT with a CustomerID
// of J9COM
Console.WriteLine("\n\nAdding new DataRow to customersDT with CustomerID of
J9COM");
DataRow myDataRow = customersDT.NewRow();
myDataRow[ "CustomerID" ] = "J9COM";
myDataRow[ "CompanyName" ] = "J9 Company";
myDataRow[ "Country" ] = "UK";
customersDT.Rows.Add(myDataRow);

// step 6: retrieve the J9COM node using GetElementFromRow()
Console.WriteLine("\nRetrieving J9COM node using GetElementFromRow() ");
XmlNode myXmlNode = myXDD.GetElementFromRow(myDataRow);
Console.WriteLine("CustomerID = "+ myXmlNode.ChildNodes[0].InnerText);
Console.WriteLine("CompanyName = "+ myXmlNode.ChildNodes[1].InnerText);
Console.WriteLine("Country = "+ myXmlNode.ChildNodes[2].InnerText);

// step 7: set J9COM node's Country to USA, first setting
// EnforceConstraints to false
Console.WriteLine("\nSetting J9COM node's Country to USA");
myDataSet.EnforceConstraints = false;
myXmlNode.ChildNodes[2].InnerText = "USA";

// step 8: retrieve the ANATR XmlNode using SelectSingleNode()
Console.WriteLine("\nRetrieving ANATR node using SelectSingleNode() ");
myXmlNode =
    myXDD.SelectSingleNode(
        "/NewDataSet/Customers[CustomerID=\\" ANATR\\"]"
    );

// step 9: retrieve the ANATR DataRow using GetRowFromElement()
Console.WriteLine("\nRetrieving ANATR DataRow using GetRowFromElement() ");
myDataRow =
    myXDD.GetRowFromElement((XmlElement) myXmlNode);
foreach ( DataColumn myDataColumn in customersDT.Columns)
{
    Console.WriteLine(myDataColumn + " = " +
        myDataRow[myDataColumn]);
}

// step 10: remove the ANATR node using RemoveAll()
Console.WriteLine("\nRemoving ANATR node");
myXmlNode.RemoveAll();

// step 11: display the XML document in myXDD using Save()
Console.WriteLine("\nXML document in myXDD:");
myXDD.Save(Console.Out);
```

```
// step 12: display the DataRow objects in customersDT using
// DisplayDataRows()
DisplayDataRows(customersDT);
}
}
```

---

The output from this program is as follows:

```
Customer DataRow objects in customersDT:
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
Country = Germany
CustomerID = ANATR
CompanyName = Ana Trujillo Emparedados y helados
Country = Mexico
```

XML document in myXDD:

```
<?xml version="1.0" encoding="IBM437"?>
<NewDataSet>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkiste</CompanyName>
    <Country>Germany</Country>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
    <Country>Mexico</Country>
  </Customers>
</NewDataSet>
```

Adding new DataRow to customersDT with CustomerID of J9COM

```
Retrieving J9COM node using GetElementFromRow()
CustomerID = J9COM
CompanyName = J9 Company
Country = UK
```

Setting J9COM node's Country to USA

```
Retrieving ANATR node using SelectSingleNode()
```

```
Retrieving ANATR DataRow using GetRowFromElement()
CustomerID = ANATR
CompanyName = Ana Trujillo Emparedados y helados
Country = Mexico
```

Removing ANATR node

```
XML document in myXDD:
<?xml version="1.0" encoding="IBM437"?>
<NewDataSet>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkiste</CompanyName>
    <Country>Germany</Country>
  </Customers>
  <Customers>
  </Customers>
```

```
<Customers>
  <CustomerID>J9COM</CustomerID>
  <CompanyName>J9 Company</CompanyName>
  <Country>USA</Country>
</Customers>
</NewDataSet>
```

Customer DataRow objects in customersDT:

```
CustomerID = ALFKI
CompanyName = Alfreds Futterkiste
Country = Germany
CustomerID =
CompanyName =
Country =
CustomerID = J9COM
CompanyName = J9 Company
Country = USA
```

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Summary

In this chapter, you learned about SQL Server's extensive support for XML. You also saw how to store XML in a C# program using the  `XmlDocument` and  `XmlDataDocument` objects.

SQL Server extends the `SELECT` statement to allow you to query the database and get results back as XML. Specifically, you can add a `FOR XML` clause to the end of a `SELECT` statement, which specifies that SQL Server is to return results as XML.

You examined the Extensible Markup Language Path (XPath) and the Extensible Stylesheet Language Transformation (XSLT). XPath is a language that allows you to search and navigate an XML document using expressions. XML is a great way to represent data in a portable format, but XML doesn't contain information on how to format that data for display. XSLT allows you to control the formatting of XML data, and may be used to transform XML data to a format suitable for display.

You can access SQL Server using HTTP (Hypertext Transfer Protocol). This allows you to run SQL statements from a browser; for example, you can run a `SELECT` statement that returns XML, and SQL Server will display the results in your browser. You can use XPath statements to locate data in the returned XML, and use XSLT stylesheets to format the returned XML.

SQL Server contains a function named `OPENXML( )` that allows you to read XML data as if it were a result set of rows. One use of `OPENXML( )` is to read XML data as rows and then insert those rows into a table.

You use an object of the  `XmlDocument` class to represent an XML document in a C# program. An  `XmlDocument` object stores the nodes of the XML document in objects of the  `XmlNode` class. You can, for example, load rows from the database into a  `DataSet`, and then load an XML representation of those rows into an  `XmlDocument` object.

You use an object of the  `XmlDataDocument` class to access rows as both  `XmlNode` objects and relational  `DataRow` objects. You associate a  `DataSet` with your  `XmlDataDocument` by passing your  `DataSet` to the  `XmlDataDocument` constructor. An  `XmlDataDocument` object provides synchronization between the  `DataSet` and the XML document. For example, if you add a new customer as an  `XmlNode` object to your  `XmlDataDocument`, then that customer is also added as a  `DataRow` to your associated  `DataSet`.

In the [next chapter](#), you'll learn about Web services.

## Chapter 17: Web Services

### Overview

A web service is a software component that you can call over the Web, and one of the key features of .NET is the ability to easily create Web services.

Companies can create Web services to allow customer interaction. For example, a shipping company might create a Web service that allows other companies to pass an XML document across the Web containing a list of items that need to be shipped. The shipping company could accept that file and schedule a pickup for those items, returning an XML document from the Web service containing a list of tracking numbers for each item to be shipped.

Because Web services return and accept data in the form of XML documents, Web services are truly platform independent. For example, you could have a Web service written in C# communicate with another Web service written in Java, passing data in the form of XML documents.

In this chapter, you'll see how to create a Web service using VS .NET and use it in a Windows application. You'll also see how to register a Web service so that other organizations can use your service. For comprehensive coverage of Web services, see *.NET Web Services Solutions* by Kris Jamsa (Sybex, 2003).

Featured in this chapter:

- Creating a Web service
- Viewing a WSDL file and testing a Web service
- Using a Web service
- Registering a Web service

## Team LiB

# Creating a Web Service

In this section, you'll create a Web service that contains a method that returns a `DataSet` containing rows from the `Customers` table.

Start VS .NET and select File > New > Project. In the New Project dialog box, select Visual C# Projects in the Project Types pane on the left, and select ASP.NET Web Service in the Templates pane on the right. Enter `http://localhost/NorthwindWebService` in the Location field (see [Figure 17.1](#)). Click OK to continue.

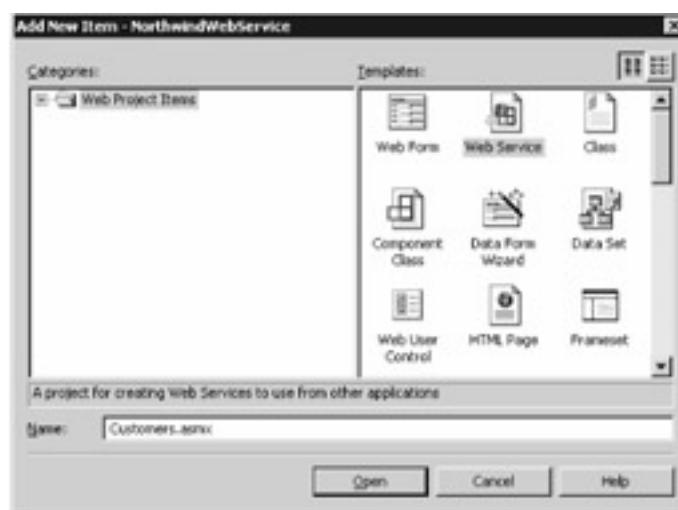


**Figure 17.1:** Creating a Web service in VS .NET

**Note** If you have installed IIS on a computer other than your local machine, then replace `localhost` with the name of your remote computer in the Location field.

After VS .NET has created the new project, open Solution Explorer and delete the `Service1.asmx` file from your project; you'll be adding your own `.asmx` file next, and it's easier to simply delete the initial `Service1.asmx` file.

Select Project > Add Web Service, and enter `Customers.asmx` in the Name field of the Add New Item dialog box (see [Figure 17.2](#)). Click Open to continue. VS .NET adds a file named `Customers.asmx` to your project.



**Figure 17.2:** Adding a new Web service

Select View > Code to view the C# code in the `Customers.asmx.cs` file. [Listing 17.1](#) shows my example `Customers.asmx.cs` file.

### **Listing 17.1: CUSTOMERS.ASMX.CS**

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace NorthwindWebService
{
    /// <summary>
    /// Summary description for Customers.
    /// </summary>
    ///
    [WebService(Namespace="http://DbProgramming/NorthwindWebService")]
    public class Customers : System.Web.Services.WebService
    {
        public Customers()
        {
            //CODEGEN: This call is required by the ASP.NET Web Services Designer
            InitializeComponent();
        }

        #region Component Designer generated code

        //Required by the Web Services Designer
        private IContainer components = null;

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose(bool disposing)
        {
            if(disposing && components != null)
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #endregion

        // WEB SERVICE EXAMPLE
        // The HelloWorld() example service returns the string Hello World
        // To build, uncomment the following lines then save and build the project
        // To test this web service, press F5

        // [WebMethod]
        // public string HelloWorld()
        // {
        //     return "Hello World";
        // }
```

---

 }

Notice that the `Customers` class is derived from the `System.Web.Services.WebService` class, which indicates that the `Customers` class forms part of a Web service.

Near the end of [Listing 1.1](#), you'll notice a method named `HelloWorld()` that is commented out. This commented code shows you how to write a method that is exposed by your Web service. You'll notice that a line containing `[WebMethod]` is placed before the method, which indicates that the method would be exposed by the Web service. Of course, because the `HelloWorld()` method is commented out, the method isn't compiled and therefore isn't actually exposed by the Web service.

Replace the example `HelloWorld()` method in your code with the `RetrieveCustomers()` method shown in [Listing 17.2](#). `RetrieveCustomers()` connects to the Northwind database and returns a `DataSet` containing rows from the `Customers` table. You pass a `WHERE` clause to the `RetrieveCustomers()` method in the `whereClause` parameter; this `WHERE` clause is then used in the `SELECT` statement to limit the rows retrieved from the `Customers` table.

## **Listing 17.2: CUSTOMERSWEBSERVICE.CS**

---

```
[WebMethod]
public DataSet RetrieveCustomers(string whereClause)
{
    SqlConnection mySqlConnection =
        new SqlConnection("server=localhost;database=Northwind;uid=sa;pwd=sa");
    string selectString =
        "SELECT CustomerID, CompanyName, Country " +
        "FROM Customers " +
        "WHERE " + whereClause;
    SqlCommand mySqlCommand = mySqlConnection.CreateCommand();
    mySqlCommand.CommandText = selectString;
    SqlDataAdapter mySqlDataAdapter = new SqlDataAdapter();
    mySqlDataAdapter.SelectCommand = mySqlCommand;
    DataSet myDataSet = new DataSet();
    mySqlConnection.Open();
    mySqlDataAdapter.Fill(myDataSet, "Customers");
    mySqlConnection.Close();
    return myDataSet;
}
```

---

**Note** You'll need to change the string used to create the `mySqlConnection` object in your code to connect to your Northwind database.

Because the code uses classes in the `System.Data.SqlClient` namespace, you'll also need to add the following line near the top of your `Customers.asmx.cs` file:

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

By default, a Web service uses a namespace of `http://tempuri.org`, and you should change that to the URL used by your organization. The following example sets the namespace for the Web service to `http://DbProgramming/NorthwindWebService`:

```
[WebService(Namespace="http://DbProgramming/NorthwindWebService")]
public class Customers : System.Web.Services.WebService
```

Notice that you set the `Namespace` in a line placed before the `Customers` class. Go ahead and add a line similar to the previous one to your own code.

Build your Web service by selecting Build > Build Solution.

That's it! You've built your Web service.

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Viewing a WSDL File and Testing a Web Service

WSDL stands for Web Services Description Language, and a WSDL file contains a complete description of your Web service, including the information required to call your service's methods. A WSDL file is written in XML and specifies the following information:

- Web service methods
- Data types used by the methods
- Request and response message formats for communication with the methods

**Note** For comprehensive information on WSDL, visit [www.w3.org/TR/wsdl](http://www.w3.org/TR/wsdl).

You access your Web service by pointing your browser to the following URL:

`http://localhost/NorthwindWebService/Customers.asmx`

As you can see from [Figure 17.3](#), the resulting page displayed in your browser contains two links named Service Description and Retrieve Customers.



**Figure 17.3:** Accessing a Web service

**Note** You can also access your Web service by right-clicking on the `Customers.asmx` file in the Solution Explorer window in VS .NET and selecting Set As Start Page from the pop-up menu. You then select Debug > Start Without Debugging to test your service. VS .NET will start Internet Explorer and display the same test page as was shown in [Figure 17.3](#).

## Viewing the WSDL File for the Web Service

If you click the Service Description link, you'll see the description of your Web service in the form of a WSDL file, which is shown in [Listing 17.3](#). Notice that this WSDL file is written in XML and contains the details on how to call the methods exposed by the example Web service. The WSDL file also contains the data types of the parameters used and the calls you can make to your methods.

### Listing 17.3: WEB SERVICE WSDL FILE

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
  xmlns: http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns: soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns: s="http://www.w3.org/2001/XMLSchema"
  xmlns: s0="http://DbProgramming/NorthwindWebService"
  xmlns: soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns: tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns: mime="http://schemas.xmlsoap.org/wsdl/mime/"
  targetNamespace="http://DbProgramming/NorthwindWebService"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
```

```
<s:schema elementFormDefault="qualified"
  targetNamespace="http://DbProgramming/NorthwindWebService">
  <s:import namespace="http://www.w3.org/2001/XMLSchema" />
  <s:element name="RetrieveCustomers">
    <s:complexType>
      <s:sequence>
        <s:element minOccurs="0" maxOccurs="1"
          name="whereClause" type="s:string" />
      </s:sequence>
    </s:complexType>
  </s:element>
  <s:element name="RetrieveCustomersResponse">
    <s:complexType>
      <s:sequence>
        <s:element minOccurs="0" maxOccurs="1"
          name="RetrieveCustomersResult">
          <s:complexType>
            <s:sequence>
              <s:element ref="s:schema" />
              <s:any />
            </s:sequence>
          </s:complexType>
        </s:element>
      </s:sequence>
    </s:complexType>
  </s:element>
</s:types>
<message name="RetrieveCustomersSoapIn">
  <part name="parameters" element="s0:RetrieveCustomers" />
</message>
<message name="RetrieveCustomersSoapOut">
  <part name="parameters" element="s0:RetrieveCustomersResponse" />
</message>
<message name="RetrieveCustomersHttpGetIn">
  <part name="whereClause" type="s:string" />
</message>
<message name="RetrieveCustomersHttpGetOut">
  <part name="Body" element="s0:DataSet" />
</message>
<message name="RetrieveCustomersHttpPostIn">
  <part name="whereClause" type="s:string" />
</message>
<message name="RetrieveCustomersHttpPostOut">
  <part name="Body" element="s0:DataSet" />
</message>
<portType name="CustomersSoap">
  <operation name="RetrieveCustomers">
    <input message="s0:RetrieveCustomersSoapIn" />
    <output message="s0:RetrieveCustomersSoapOut" />
  </operation>
</portType>
```

```
<portType name="CustomersHttpGet">
  <operation name="RetrieveCustomers">
    <input message="s0:RetrieveCustomersHttpGetIn" />
    <output message="s0:RetrieveCustomersHttpGetOut" />
  </operation>
</portType>
<portType name="CustomersHttpPost">
  <operation name="RetrieveCustomers">
    <input message="s0:RetrieveCustomersHttpPostIn" />
    <output message="s0:RetrieveCustomersHttpPostOut" />
  </operation>
</portType>
<binding name="CustomersSoap" type="s0:CustomersSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"
/>
  <operation name="RetrieveCustomers">

    <soap:operation
      soapAction="http://DbProgramming/NorthwindWebService/RetrieveCustomers"
      style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<binding name="CustomersHttpGet" type="s0:CustomersHttpGet">
  <http:binding verb="GET" />
  <operation name="RetrieveCustomers">
    <http:operation location="/RetrieveCustomers" />
    <input>
      <http:urlEncoded />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<binding name="CustomersHttpPost" type="s0:CustomersHttpPost">
  <http:binding verb="POST" />
  <operation name="RetrieveCustomers">
    <http:operation location="/RetrieveCustomers" />
    <input>
      <mime:content type="application/x-www-form-urlencoded" />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<service name="Customers">
  <port name="CustomersSoap" binding="s0:CustomersSoap">
    <soap:address

      location="http://localhost/NorthwindWebService/Customers.asmx" />
  </port>
  <port name="CustomersHttpGet" binding="s0:CustomersHttpGet">
    <http:address
```

```

    location="http://localhost/NorthwindWebService/Customers.asmx" />
</port>
<port name="CustomersHttpPost" binding="s0:CustomersHttpPost">

    <http:address
        location="http://localhost/NorthwindWebService/Customers.asmx" />
    </port>
</service>
</definitions>
```

Next, you'll see how to test your Web service.

## Testing a Web Service

To test your Web service, point your browser to the following URL:

`http://localhost/NorthwindWebService/Customers.asmx`

Click the Retrieve Customers link. Your browser displays a page (see [Figure 17.4](#)) that you can use to test the `RetrieveCustomers()` method exposed by your Web service.



**Figure 17.4:** The Web service test page

The test page contains a text box with a label of `whereClause` where you can enter values for the `whereClause` parameter of your `RetrieveCustomers()` method. The text you enter for your `whereClause` is passed to the `RetrieveCustomers()` method when you click the `Invoke` button on the page. Enter the following text as your `whereClause`:

`CustomerID='ALFKI'`

Click the `Invoke` button to run the `RetrieveCustomers()` method. With this `whereClause`, the `RetrieveCustomers()` method returns a `DataSet` with a `DataTable` containing the one row from the `Customers` table with a `CustomerID` of `ALFKI`, as shown in [Figure 17.5](#). Notice that the equals (=) and single quote (') characters in the `whereClause` parameter value of the URL have been converted to the codes `%3D` and `%27` respectively.



**Figure 17.5:** Running the `RetrieveCustomers()` method with a `whereClause` of `CustomerID= 'ALFKI'`

As you can see from [Figure 17.5](#), the `DataSet` is returned as an XML document. You can use this XML in your client programs that use the Web service. You'll see how to write a client program in the [next section](#).

Let's take a look at another example; enter the following string as your `whereClause` and click the `Invoke` button:

`CustomerID IS NOT NULL`

This causes the `RetrieveCustomers()` method to return a `DataSet` with a `DataTable` containing all the rows from the `Customers` table (see [Figure 17.6](#)). Notice that the space characters in the `whereClause` parameter value have been converted to plus (+) characters. You'll need to scroll down the page to see the other customers.

```

<?xml version="1.0" encoding="utf-8" ?>
- <DataSet xmlns="http://DbProgramming/NorthwindWebService">
- <xss:schema id="NewDataSet" xmlns="" xmlns:xss="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
- <xss:element name="NewDataSet" msdata:IsDataSet="true">
- <xss:complexType>
- <xss:choice maxOccurs="unbounded">
- <xss:element name="Customers">
- <xss:complexType>
- <xss:sequence>
- <xss:element name="CustomerID" type="xs:string" minOccurs="0" />
- <xss:element name="CompanyName" type="xs:string"
  minOccurs="0" />
- <xss:element name="Country" type="xs:string" minOccurs="0" />
</xss:sequence>
</xss:complexType>
</xss:element>
</xss:schema>
<diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
- <NewDataSet xmlns="">
- <Customers diffgr:id="Customers1" msdata:rowOrder="0">
  <CustomerID>ALFKI</CustomerID>
  <CompanyName>Alfreds Futterkiste</CompanyName>
  <Country>Germany</Country>
</Customers>
- <Customers diffgr:id="Customers2" msdata:rowOrder="1">
  <CustomerID>ANATR</CustomerID>
  <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
  <Country>Mexico</Country>
</Customers>
- <Customers diffgr:id="Customers3" msdata:rowOrder="2">

```

**Figure 17.6:** Running the `RetrieveCustomers()` method with a `whereClause` of `CustomerID IS NOT NULL`

Next, you'll see how to use your Web service in a Windows application.

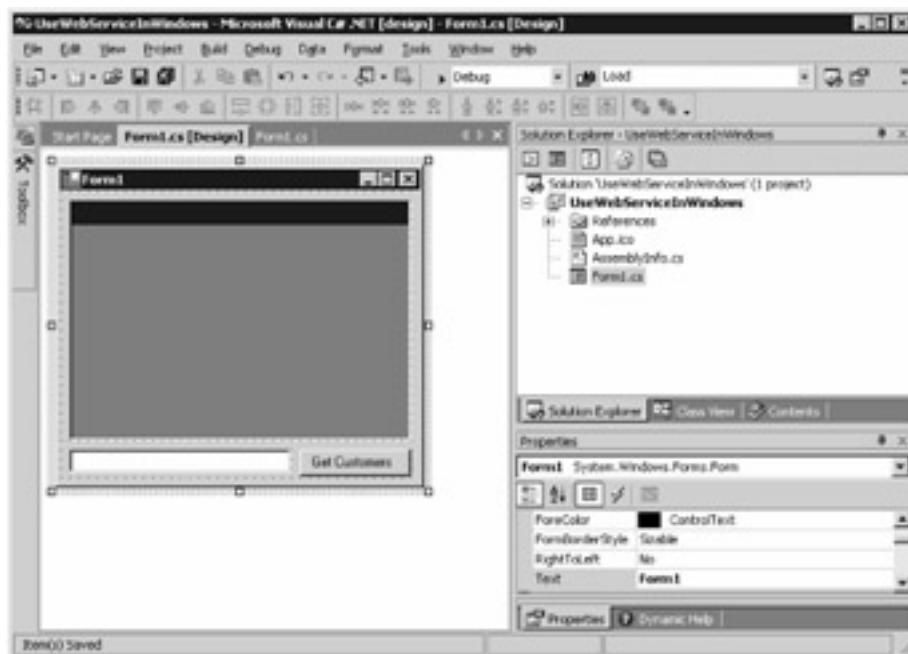
**Team LiB**

◀ PREVIOUS    NEXT ▶

## Using a Web Service

In this section, you'll see how to use a Web service in a Windows application.

Start VS .NET and select File > New > Project. Create a new Windows application named `UseWebServiceInWindows`. Drag a `DataGridView`, `TextBox`, and `Button` control to your form. Set the `Name` property of your `DataGridView` to `customersDataGridView`. Set the `Name` property of your `TextBox` to `whereClauseTextBox`, and remove the text `textBox1` from the `Text` property. Set the `Name` property of your `Button` to `getCustomersButton`, and set the `Text` property to `Get Customers`. These controls are shown in [Figure 17.7](#).



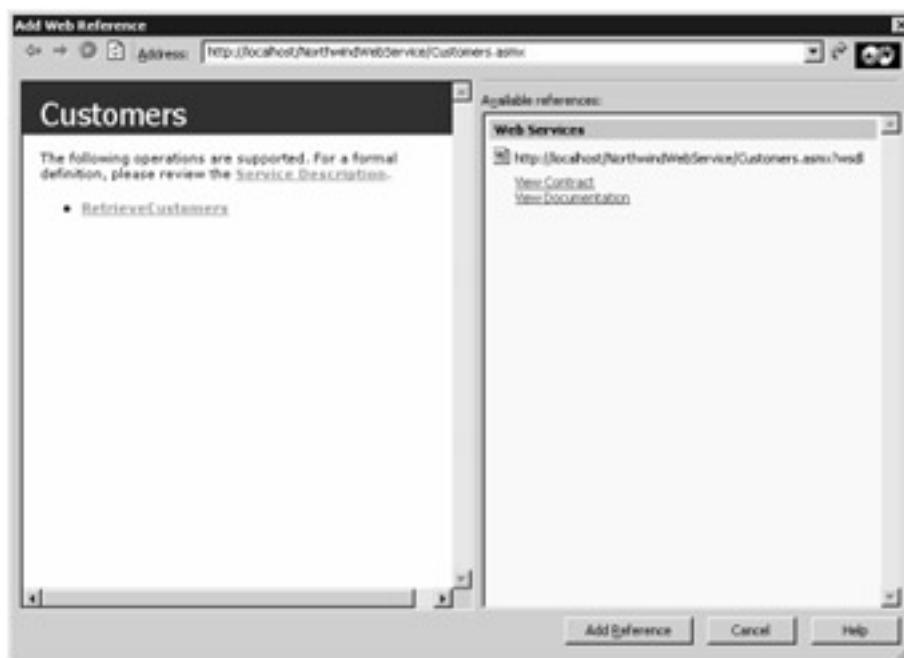
**Figure 17.7:** Form with controls

Open the Solution Explorer window and right-click the References node. Select Add Web References from the pop-up menu. This displays the Add Web Reference dialog box, which allows you to search for Web services. Enter the following URL in the Address box, and press the Enter key on your keyboard:

```
http://localhost/NorthwindWebService/Customers.asmx
```

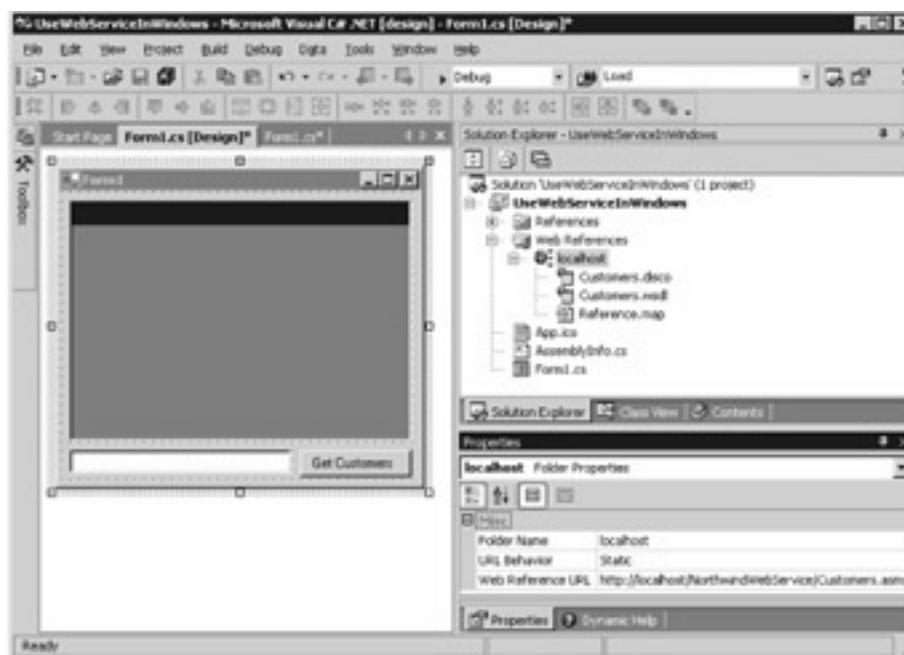
**Note** If your Web service is not deployed on the local computer, then replace `localhost` with the name of your remote computer.

Your Web service will be located and a test page displayed (see [Figure 17.8](#)).

**Figure 17.8:** Northwind Web Service

You can view the WSDL file for your Web service by clicking the Service Description link, and you can test your Web service by clicking the Retrieve Customers link.

Click the Add Reference button to add the reference to your Web service to your project and continue. You can see the new reference in the Solution Explorer window (see [Figure 17.9](#)).

**Figure 17.9:** The new Web reference in Solution Explorer

Double-click the Button on your form to open the code editor, and add the following code to your button's click method:

```
localhost.Customers myCustomersService = new localhost.Customers();
customersDataGridView.DataSource =
    myCustomersService.RetrieveCustomers(whereClauseTextBox.Text);
customersDataGridView.DataMember = "Customers";
```

**Note** Once again, if your Web service is not deployed on the local computer, then replace *localhost* in this code with the name of your remote computer.

This code creates an object named `myCustomersService` to call your Web service, and displays the returned results from the `RetrieveCustomers()` method in `customersDataGrid`.

Compile and run your Windows application by selecting Debug > Start Without Debugging. Enter `CustomerID='ALFKI'` in the text box, and click the `Get Customers` button; the retrieved results are shown in [Figure 17.10](#).



**Figure 17.10:** The running form

Next, you'll see how to register your Web service.

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

## Registering a Web Service

In this section, you'll see how to register a Web service using Microsoft's Universal Description, Discovery, and Integration (UDDI) service. You can think of UDDI as a distributed directory of Web services that you can use to register and locate Web services published by organizations. UDDI is an industry standard developed by Microsoft, IBM, Sun Microsystems, and other software and hardware companies.

**Note** For comprehensive information on UDDI, visit [www.uddi.org](http://www.uddi.org) and [uddi.microsoft.com](http://uddi.microsoft.com).

Once you've registered your Web service, anyone can use your service as a software component in their own system; similarly, you could use other people's Web services in your system. You can even register Web services for your own organization's intranet and build an internal system made up of Web services written internally.

In this section, you'll register the `NorthwindWebService` you created earlier in this chapter. To do that, follow these steps:

From VS .NET, click the Start Page tab, click the XML Web Services link, and then click the Register Your XML Web Service Today link (see [Figure 17.11](#)). You can search for Web services using the Find A Service page.



**Figure 17.11:** The XML Web Services page

From the UDDI Web Service Registration page, you can register your Web service with either the test or production environment. Since your Web service is an example, click the UDDI Test Environment radio button and click the Submit button (see [Figure 17.12](#)). If you create a really useful Web service that you believe other organizations will want to use, you can register your Web service with the production environment.



**Figure 17.12:** The UDDI Web Service Registration page

Read the text in the UDDI Business Registry Node page (see [Figure 17.13](#)). This page explains the next steps you follow. Click the Sign In button when you've finished reading the text.



**Figure 17.13:** The UDDI Business Registry Node page

You'll need a Microsoft Passport account to continue. If you have such an account, enter your details (see [Figure 17.14](#)). Click the Continue button to proceed.



**Figure 17.14:** Logging in using a Microsoft Passport account

**Note** If you don't have a Passport account, click the Get One Now link and sign up for a Passport account.

Enter your email address, name, and phone number in the UDDI Business Registry Node page (see [Figure 17.15](#)). The name and phone number are near the bottom of the page and you'll need to scroll down to see them. Click the Save button to continue.



**Figure 17.15:** Entering your email address, name, and phone number

Read the Terms Of Use page and click Accept if you want to continue.

**Figure 17.16:** The terms of use page

Enter your business name and an optional description (see [Figure 17.17](#)). Click Save to continue.

**Figure 17.17:** Setting the business name and description

You'll be asked to select the UDDI environment again, so make sure the UDDI Test Environment radio button is selected, and click Submit to continue.

Make sure your organization is selected, and click Submit to continue.

Next, enter the details for your Web service. Enter a name for your Web service, along with a description. The `.asmx` URL for your `NorthwindWebService` will be similar to the following URL:

`http://localhost/NorthwindWebService/Customers.asmx`

Your `.wsdl` URL will be similar to the following URL:

`http://localhost/NorthwindWebService/Customers.asmx?WSDL`

Select Miscellaneous for your Service Category. These settings are shown in [Figure 17.18](#). Click Submit to register your Web service.



**Figure 17.18:** Setting the Web service details

That's it. You've successfully registered your Web service. Feel free to search for and examine the Web services currently registered using the Find A Service page.

**Team LiB**

◀ PREVIOUS ▶ NEXT ▶

**Team LiB**[◀ PREVIOUS](#)[NEXT ▶](#)

## Summary

A Web service is a software component that you can call over the Web, and one of the key features of .NET is the ability to easily create Web services. Companies can create Web services to allow their customers to interact with them.

Because Web services return and accept data in the form of XML documents, Web services are truly platform independent. For example, you could have a Web service written in C# communicate with another Web service written in Java, passing data in the form of XML documents.

In this chapter, you saw how to create a Web service using VS .NET and use it in a Windows application. You also saw how to register a Web service so that other organizations can use your service.

I hope you found this book informative and useful, and I hope I've held your interest! Database programming with C# is a very large subject, but armed with this book, I have every confidence you will master it.

**Team LiB**[◀ PREVIOUS](#)[NEXT ▶](#)

## Index

**Note to the Reader:** Throughout this index **boldfaced** page numbers indicate primary discussions of a topic. *Italicized* page numbers indicate illustrations.

### A

ABS function, [111-112](#)

AcceptChanges method

    in DataRow, [348](#)

    in DataSet, [301](#)

    in DataTable, [346](#), [468](#)

AcceptChangesDuringFill property, [294](#)

AcceptRejectRule property, [420](#)

Access databases, connecting to, [12-14](#)

access modifiers, [161](#)

ACOS function, [111-112](#)

Action property, [406](#)

Add Connection option, [332](#)

Add method

    in ColumnMappings, [330](#)

    in Constraints, [352-353](#)

    in DataTableMapping, [329](#)

    for ListView controls, [289-290](#)

    in myDataSet, [426](#)

    in myDataTable, [380](#)

    in mySqlCommand, [242](#)

    in Rows, [377](#)

Add New Item dialog box

    for DataSets, [332](#), [333](#)

    for forms, [173](#), [173](#)

    for Web services, [614](#), [614](#)

Add option for row display, [178](#)

Add Table dialog, [256](#), [257](#)

Add This Dataset To The Designer option, [533](#)

Add value in MissingSchemaAction, [319](#)

Add Web Reference dialog box, [624](#)

Add Web Service option, [614](#)

AddCustomersRow method, [337](#), [413](#)

AddCustomersXml.sql script, [597-598](#)

AddDataRow method, [377](#), [379-380](#), [394](#)

Added value

    in DataRowState, [379](#)

    in DataViewRowState, [368](#), [436](#), [456](#)

AddModifyAndRemoveDataRows.cs program, [381](#)

AddModifyAndRemoveDataRowViews.cs program, [468-472](#)

AddNew method, [455](#), [467](#)

AddProduct.sql stored procedure, [125-127](#), [246-247](#)

AddProduct2.sql stored procedure, [250-251](#)

AddProduct3.sql stored procedure, [252-253](#)

AddProduct4.sql stored procedure, [387-388](#), [394](#)

AddRestrictions.cs program, [357-361](#)

AddToCart method, [561](#), [564](#)

AddWithKey value, [319](#)

ADO.NET classes, [137](#)

generic data, [141-143](#), [142](#)

managed provider, [138-141](#), [139](#)

AdRotator controls, [526](#)

Advanced tab, [202](#)

AFTER INSERT clause, [129](#)

AFTER UPDATE clause, [133](#)

aggregate functions, [77](#), [107-110](#), [109-110](#)

aliases, [78-79](#), [78](#)

ALL function, [108](#)

All tab, [202](#), [203](#)

Allow Paging property, [538](#)

Allow Saving Password option, [202](#)

Allow Template Queries only option, [589](#)

Allow URL Queries option, [589](#)

Allow XPath Queries option, [589](#)

AllowDBNull property

    in DataColumn, [349-350](#), [353-354](#)

    for schema columns, [219](#)

AllowDelete property, [454](#)

AllowEdit property, [454](#)

AllowNew property, [454](#)

ALTER PROCEDURE statement, [126](#)

ALTER TABLE statement, [90](#)

ampersands (&) in URLs, [591](#)

AND operator, [71](#), [71](#)

AppendChild method, [601](#)

AppendCookie method, [552](#)

Application class,

    as static class [162-163](#)

    for Web state information, [558-559](#)

ApplicationObjectTest.aspx file, [558-559](#)

applications

ASP.NET

    creating, [517-524](#), [518-520](#)

    VS.NET for, [17](#)

    Windows. See [Windows applications](#)

ApplyDefaultSort property, [454](#)

AS clause

    for columns, [328-329](#), [329](#)

    for tables, [78-79](#), [78](#)

ASC keyword, [73](#), [74](#), [457](#)

ASCII function, [113-114](#)

ASIN function, [111-112](#)

ASP.NET applications

    creating, [517-524](#), [518-520](#)

    VS.NET for, [17](#)

ASP.NET Web Services, [17](#)

assembly files, [163](#)

asterisks (\*)

    for comments, [97](#)

    in query builder, [49](#)

    in SELECT, [63](#)

    in XPath expressions, [581](#)

at characters (@)

    for variables, [96](#)

    for XML attributes, [582](#)

ATAN function, [111-112](#)

ATN2 function, [111](#)

Attributes property, [600](#)

AUTO mode for XML, [227](#), [570](#)

Auto value in XmlReadMode, [326](#)

AutoEventWireUp attribute, [521](#)

AutoIncrement property, [349-350](#)

AutoIncrementSeed property, [349-350](#), [356](#)

AutoIncrementStep property, [349-350](#), [356](#)

automatic SQL statement generation, [398-399](#)

AVG function, [108](#)

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

## Index

### B

Back Color property in DataGridView, [538](#)

BackColor property for forms, [156](#)

backslashes (\) in connection strings, [12](#)

BaseCatalogName property, [219](#)

BaseColumnName property, [219](#)

BaseSchemaName property, [219](#)

BaseTableName property, [219](#)

BaseURI property, [600](#)

BEGIN keyword, [97](#)

BEGIN TRANSACTION statement, [87-88](#)

BeginEdit method

    in DataRow, [348](#), [382](#)

    in DataRowView, [462](#), [467](#)

BeginInit method

    in DataSet, [301](#)

    in DataView, [455](#)

BeginTransaction method, [188](#), [237](#), [496](#)

Behavior property, [536](#)

BETWEEN operator, [69](#)

bigint type in SQL Server, [43](#), [269](#)

BigInt value in SqlDbType, [241](#)

BINARY BASE64 option, [227](#), [576-578](#), [577](#)

binary type in SQL Server, [43](#), [269](#)

Binary value in SqlDbType, [241](#)

binding controls, [179-180](#)

bit type in SQL Server, [43](#), [269](#)

Bit value in SqlDbType, [241](#)

Block.cs program, [506-509](#)

blocking, transaction, [504-505](#), [504-505](#)

blocks

    dividing retrieved rows into, [76-77](#), [77](#)

    in T-SQL, [97](#)

Bold property, [538](#)

bool type, [268-269](#)

Boolean type, [268](#)

Border Color property, [540](#)

Border Width property, [540](#)

Borders link for DataGrid controls, [539-540](#), [539](#)

BoundColumn Properties option, [537](#)

brackets (())

    in pattern matching, [65](#)

    in table names, [64](#)

    in XPath expressions, [581](#)

BREAK statement, [100-101](#)

Broken value, [195](#)

browsers

    INSERT, UPDATE, and DELETE statements with, [592](#)

    SELECT statements with, [590-591](#), [591](#)

    stored procedures with, [592-593](#), [593](#)

BU (Bulk update) locking mode, [502](#)

Build menu, [20](#)

Build Solution option, [23](#)

Building Console Applications link, [24](#)

building projects, [23](#)

Bulk update (BU) locking mode, [502](#)

Button controls

    in Web forms, [524](#)

    in Windows applications, [156-158](#), [157-158](#), [165](#)

Buy button, [560-562](#), [561-562](#)

byte type, [268-270](#)

**Team LiB**

◀ PREVIOUS    **NEXT ▶**

## Index

### C

Cache objects, [559](#)

calculated fields, [79](#)

Calendar controls, [526](#)

calling functions, [120](#)

Cancel All option, [178](#)

Cancel method in SqlCommand, [209](#)

Cancel option for row display, [178](#)

CancelEdit method

    in DataRow, [348](#), [404](#)

    in DataRowView, [462](#), [467](#)

Caption property, [349](#)

carets (^) in pattern matching, [65](#)

Cascade Delete Related Fields option, [54](#), [438](#)

Cascade Update Related Fields option, [54](#), [438](#), [440-442](#)

Cascade value, [439](#)

CASE statement, [98-99](#)

CaseSensitive property

    in DataSet, [299](#)

    in DataTable, [346](#)

casting objects, [267](#)

catch blocks, [7](#), [9-10](#)

cdata directive, [572](#)

CEILING function, [111-112](#)

Cell Padding property, [540](#)

Cell Spacing property, [540](#)

ChangeDatabase method, [188](#)

Chaos value, [493](#)

CHAR function, [113](#), [115](#)

char type in SQL Server, [43](#), [268-269](#)

Char value in SqlDbType, [241](#)

CHARINDEX function, [113](#), [115](#)

Check existing data on creation option, [54](#), [56](#)

CheckBox controls, [165](#), [525](#)

CheckBoxList controls, [525](#)

CheckedListBox controls, [165](#)

child DataView objects, [472-475](#)

child tables, [40](#), [86](#), [417](#)

child XML nodes, [579](#)  
ChildColumns property, [424](#)  
ChildKeyConstraint property, [424](#)  
ChildNodes property, [600](#)  
ChildRelations property, [346](#)  
ChildTable property, [424](#)  
Choose a data connection screen, [174](#), [174](#)  
Choose a Query Type screen, [339](#), [339](#)  
Choose tables and columns to display on the form screen, [177](#), [177](#)  
Choose tables or views screen, [175](#), [175](#)  
Choose the dataset you want to use screen, [173](#), [174](#)  
Choose the display style screen, [178](#), [178](#)  
Choose Your Data Connection screen, [338](#), [338](#)  
Class View, [164](#), [164](#)  
Clear method  
    in DataSet, [301](#)  
    in DataTable, [346](#)  
ClearErrors method, [348](#)  
clients for Web state information, [552-556](#), [554](#)  
Clone method  
    in DataSet, [301](#)  
    in DataTable, [346](#)  
CloneNode method, [601](#)  
Close method  
    in mySqlConnection, [378](#)  
    in SqlCommand, [9](#)  
    in SqlConnection, [147](#), [188](#)  
    in SqlDataReader, [9](#)  
CLOSE statement, [105](#)  
CloseConnection value, [215](#)  
Closed value, [195](#)  
closing  
    cursors, [105](#)  
    database connections, [190-192](#)  
    DataReader objects, [264](#)  
    SqlCommand objects, [9](#)  
    SqlDataReader objects, [9](#)  
COALESCE function, [118-119](#)  
code behind forms, [158-163](#), [162](#)  
code files in Solution Explorer, [163](#)  
Codebehind attribute, [521](#)  
Collation field, [53](#)

colon (:) for labels, [101](#)

Column List option, [536](#)

Column property, [405](#)

column types, [42-44](#)

ColumnChanged event, [347](#), [404-405](#)

ColumnChanging event, [347](#), [404-405](#)

ColumnMapping property, [349](#)

ColumnMappings property, [330](#)

ColumnName property

  in DataColumn, [349](#)

  in schema columns, [219](#)

ColumnOrdinal property, [219](#)

columns, [29](#)

  computations in, [79](#)

  for DataGrid controls, [536-537](#), [536](#)

  display names for, [78-79](#), [78](#)

  identity

    generating, [351](#)

    retrieving values of, [384-386](#)

  mapping, [328-331](#), [329](#)

  in Persons table, [52-53](#)

  in XML, [568](#)

Columns property

  in DataColumn, [142](#)

  in DataTable, [346](#)

  in ForeignKeyConstraint, [421](#)

  in UniqueConstraint, [418](#)

Columns tab, [52-53](#)

ColumnSize property, [219](#)

combining retrieved rows, [76](#), [76](#)

ComboBox controls, [165](#)

Command objects, [139](#)

  parameters for

    adding, [241-242](#)

    placeholders for, [240](#)

  VS.NET for, [255-257](#), [256-257](#)

Command Prompt tool, [11-12](#)

Command Prompt window, [23](#), [23](#)

Command property, [402](#)

CommandBehavior enumeration, [215](#)

CommandBuilder

  classes for, [140](#), [398](#)

  performance with, [398](#)

commands, executing, [207](#)

Command objects for, [255-257](#), [256-257](#)

ExecuteNonQuery for, [230-236](#)

ExecuteReader for, [212-225](#)

ExecuteScalar for, [225-226](#)

ExecuteXmlReader for, [227-229](#)

parameters for, [239-245](#)

SqlCommand objects for, [207-211](#)

stored procedures, [245-255](#)

transactions, [236-239](#)

CommandText property

  in Command, [210](#), [230](#), [234](#)

  in mySqlCommand, [242](#)

  in SelectCommand, [168](#), [314](#)

  in SqlCommand, [8](#), [146](#), [208](#), [256-257](#)

CommandTimeout property, [208](#)

CommandType enumeration, [210](#)

CommandType property, [208](#)

comments in Transact-SQL, [97](#)

Commit method

  in SqlConnection, [237](#)

  in SqlTransaction, [486](#)

COMMIT TRANSACTION statement, [87-89](#), [503](#), [505](#)

CompareValidator controls, [526](#)

compiling programs

  process, [10-12](#)

  using VS.NET, [23](#), [23](#)

Components category in Toolbox, [156](#)

computations in fields, [79](#)

Compute method, [346](#)

concurrency in changes, [375-376](#), [441-442](#)

conditional logic, [97-98](#)

conditions, multiple, [71-72](#), [71-72](#)

configuration functions, [107](#)

Configure Data Adapter link, [341](#)

Configure SQL XML Support in IIS option, [586](#)

Connect To Database option, [91](#)

Connect to SQL Server dialog box, [60](#), [60](#)

connecting to databases, [187](#)

  Access, [12-14](#)

  Oracle, [14-17](#)

  SqlConnection objects for, [187-189](#)

    connection pooling with, [192-195](#)

    Connection state with, [195-196](#)

    events for, [196-200](#)

opening and closing, [190-192](#)  
VS.NET for, [200-204](#), [200-205](#)

Connecting value, [195](#)

Connection objects, [139](#), [141](#)  
events for, [196-200](#)  
state of, [195-196](#)  
VS.NET for, [200-204](#), [200-205](#)

connection pooling, [192-195](#)

Connection property  
in SelectCommand, [341](#)  
in SqlCommand, [208](#)  
in SqlTransaction, [486](#)

connection strings, [7](#), [12](#)  
for Oracle database connections, [15](#)  
for SqlConnection objects, [167](#), [167](#)

Connection tab, [201](#), [201](#)

ConnectionPooling.cs program, [193-195](#)

ConnectionState enumeration, [195](#)

ConnectionString property  
for passwords, [181](#), [182](#)  
in SqlConnection, [188](#), [200](#), [202](#)

ConnectionTimeout property, [188](#)

console applications, [17](#)

Constraint class, [143](#)

CONSTRAINT clause, [89](#)

ConstraintCollection class, [143](#), [352](#)

ConstraintName property  
in ForeignKeyConstraint, [421](#)  
in UniqueConstraint, [418](#)

constraints. See [restrictions](#)

Constraints property, [143](#), [346](#), [352](#)

Contents option, [25](#)

Contents tab, [26](#)

context in XML, [580](#)

Continue property, [401](#)

CONTINUE statement, [100](#)

ContinueUpdateOnError property, [294](#), [409](#), [411](#)

controls

in Web forms, [524-526](#)  
in Windows applications, [164-165](#)  
adding, [180](#), [181](#)  
data binding in, [179-180](#)  
DataGrid, [165-172](#), [166-171](#)

labels and button, [156-158](#), [157-158](#)

CONVERT function, [118](#)

cookies, [552-554](#), [554](#)

Cookies collection, [553](#)

CookieTest.aspx file, [553-554](#), [554](#)

Copy method, [301](#)

CopyTo method, [455](#)

COS function, [111](#)

COT function, [111](#)

COUNT function, [77](#), [107-108](#), [110](#)

COUNT\_BIG function, [108](#)

Count property, [454](#)

Create a relationship between tables screen, [176](#), [176](#)

Create as CLUSTERED option, [56](#)

Create Columns Automatically At Run Time option, [536](#)

CREATE FUNCTION statement, [120](#)

CREATE INDEX statement, [90](#)

CREATE TABLE statement, [89](#), [233-234](#)

CREATE TRIGGER statement, [129](#), [133](#)

Create UNIQUE option, [55](#)

CreateAttribute method, [601](#)

CreateCDataSection method, [601](#)

CreateChildDataGridView.cs program, [473-475](#)

CreateChildView method, [462](#), [472-473](#)

CreateCommand method, [146](#), [188](#), [211](#)

CreateComment method, [601](#)

CreateDataView method, [475-476](#)

CreateDocumentFragment method, [601](#)

CreateDocumentType method, [601](#)

CreateElement method, [601](#)

CreateEntityReference method, [602](#)

CreateNavigator method, [602](#)

CreateNode method, [602](#)

CreateParameter method, [209](#)

CreateTextNode method, [602](#)

CreateWhitespace method, [602](#)

CreateXmlDeclaration method, [602](#)

Current value, [376](#)

CurrentPageIndex property, [541](#)

CurrentRows value, [368](#), [456](#)

cursor type, [43](#)

ursors, [103-104](#)

closing, [105](#)

declaring, [104](#)

example, [106-107](#)

functions for, [107](#)

opening, [104](#)

retrieving rows from, [104-105](#)

Customers.asmx.cs program, [615-616](#)

Customers table

contents of, [39-40](#), [44-45](#), [44](#)

definition of, [45](#)

indexes for, [42](#)

relationships for, [54](#), [54](#)

Customers.xml file, [578-579](#), [579-580](#), [593-594](#), [594](#)

customersDA DataAdapter object, [430-431](#)

customersDT, DataRow objects for

adding, [434](#)

deleting, [435](#)

pushing changes in, [435-437](#)

updating, [434](#)

CustomersFromCountry.sql script, [592-593](#)

CustomersStylesheet.xsl file, [582-584](#), [586](#), [594-595](#)

CustomersUsingStylesheet.xml file, [584-585](#), [585](#), [595-596](#), [596](#)

CustomersWebService.cs program, [616-617](#)

CustomersWithAttributes.xml script, [581-582](#)

CustomValidator controls, [526](#)

CustOrderHist stored procedure, [309](#)

**Team LiB**

◀ PREVIOUS ▶ NEXT ▶

## Index

### D

Data Adapter Configuration Wizard, [337-341](#), [338-340](#)

Data Adapter Preview dialog box, [168](#), [169](#)

data binding, [179-180](#)

Data category in Toolbox, [155](#)

Data Connections node, [323](#), [332](#)

Data Definition Language (DDL) statements, [59](#)

    with ExecuteNonQuery, [233-236](#)

    working with, [89](#)

Data Field setting, [537](#)

Data Form Wizard, [172-179](#), [172-179](#)

Data Formatting Expression field, [537](#)

Data Key Field property, [536](#)

Data Link Properties dialog box, [91](#), [91](#), [200-202](#), [201-203](#)

Data Manipulation Language (DML) statements, [59](#), [62](#)

Data namespace, [143](#)

data set classes, [137](#)

Data Source tab, [588](#), [588](#)

Data Transformation Services folder, [32](#)

data types

    fixed length, [51](#)

    in SQL Server, [42-44](#)

    in SqlDbType, [241-242](#)

    for T-SQL variables, [95](#)

DataAdapter directory, [337](#)

DataAdapter objects, [140-141](#), [293](#)

    events for, [399-404](#)

    FillSchema method for, [351](#), [362-367](#)

    for populating objects, [316-318](#)

    for pushing changes, [373-377](#)

    setting up, [429-433](#)

    for stored procedures, [390-392](#)

    Transaction property in, [413](#)

    in VS.NET, [337-341](#), [338-341](#)

database administrators (DBAs), [8](#)

database connections, [187](#)

    Access, [12-14](#)

    Oracle, [14-17](#)

    SqlConnection objects for, [187-189](#)

        connection pooling with, [192-195](#)

        connection state with, [195-196](#)

events for, [196-200](#)  
opening and closing, [190-192](#)  
VS.NET for, [200-204](#), [200-205](#)

database element in connection strings, [7](#)

Database (DB) lock type, [501](#)

database management systems, [30](#)

Database property, [188](#)

databases, [29-30](#)  
connecting to. See [database connections](#)  
integrity of, [85-86](#)  
Northwind. See [Northwind database](#)  
transactions in. See [transactions](#)

Databases folder, [32](#)

.DataBind method, [534](#), [541](#), [549](#)

DataBindings group, [179](#)

DataBindings property, [548](#)

DataTable objects, [142-143](#)  
properties for, [349-350](#)  
restrictions on, [356-362](#)

DataRowChangeEventArgs class, [404-405](#)

DataTableCollection class, [142](#)

DataTableMapping object, [330](#)

DataTableMappingCollection class, [330](#)

DataGridView controls, [525](#)  
Borders properties for, [539-540](#), [539](#)  
Columns properties for, [536-537](#), [536](#)  
for database access, [165-172](#), [166-171](#), [532-535](#), [532](#), [534](#)  
Format properties for, [538](#), [539](#)  
General properties for, [535-536](#), [535](#)  
PageIndexChanged event handler, [540-543](#), [540](#), [542](#)  
Paging properties for, [537-538](#), [537](#)  
for shopping cart, [563](#), [563](#)

DataGridView1\_PageIndexChanged method, [541](#)

DATALENGTH function, [118-119](#)

DownList controls  
for database access, [543-551](#), [544-548](#), [550](#)  
in Web forms, [525](#)

DataMember property  
in DataGridView, [533](#), [535](#)  
in DropDownList, [545](#)

DataReader objects, [140-141](#), [259](#)  
multiple SQL statements for, [282-288](#)  
for null values, [281-282](#)

SqlDataReader objects, [259-262](#)

creating, [262-263](#)  
reading rows from, [263-266](#)  
for strongly typed column values, [266-281](#)  
in VS.NET, [288-291](#), [289-291](#)

DataRelation objects, [143](#)  
constraints in, [427-428](#)  
creating and using, [424-429](#)  
properties for, [423-424](#)

DataRelationCollection, [143](#)

DataRow objects, [142-143](#), [148](#)  
with CustomersDT and OrdersDT  
    adding, [434](#)  
    deleting, [435](#)  
    updating, [434](#)  
with DataTable objects  
    adding, [377-380](#), [393-394](#), [434](#)  
    deleting, [382-384](#), [396-397](#)  
    filtering and sorting, [368-373](#)  
    finding, [367-368](#)  
    modifying, [380-382](#), [395-396](#)  
    merging, [318-321](#)  
    navigating, [428-429](#)  
    properties and methods for, [347-349](#)

DataRowChangeEventArgs class, [405-406](#)

DataRowCollection class, [142](#), [148](#)

DataRowState enumeration, [379](#)

DataRowVersion enumeration, [376](#)

DataRowView objects, [461-462](#)  
    adding, [467](#)  
    finding, [464-466](#)  
    indexes for, [463](#)  
    modifying, [467-472](#)  
    removing, [468](#)

DataSet objects, [138](#), [293-294](#), [298](#), [299](#)  
    automatic SQL statements for, [398-399](#)  
    creating, [302](#), [433-434](#)  
    DataAdapter objects for. See [DataAdapter objects](#)  
    DataRow objects for, [347-349](#)  
    DataTable objects for. See [DataTable objects](#)  
    for forms, [170](#)  
    for mapping tables and columns, [328-331](#), [329](#)  
    merging objects into, [318-321](#)  
    populating, [433-434](#)  
        DataAdapter objects for, [316-318](#)  
        DataTable objects for, [311-316](#)  
        ranges of rows for, [306-308](#)  
        SELECT statements for, [302-306](#)  
        stored procedures for, [309-311](#)

properties and methods for, [299-302](#)  
purpose of, [141](#)  
SqlDataAdapter objects for  
    creating, [297-298](#)  
    properties and methods for, [294-297](#)  
strongly typed, [332](#)  
    creating, [332-336](#), [333-335](#)  
    modifying data with, [413-415](#)  
    working with, [336-337](#), [337](#)  
transactions with, [412-413](#)  
update failures in, [408-412](#)  
in VS.NET, [342-343](#), [342-344](#)  
working with, [143-144](#), [146](#)  
for XML  
    schema relationships, [450-452](#), [451-452](#)  
    writing and reading objects, [321-328](#)

DataSet property  
    in DataRelation, [424](#)  
    in DataTable, [346](#)  
    in DataViewManager, [475](#)

DataSet tab, [332](#)

DataSetName property, [299](#)

DataSource property  
    in DataGrid, [170](#), [533](#), [535](#)  
    in DataList, [545](#)  
    in SqlConnection, [188](#)

DataTable objects, [141-143](#), [147-148](#)  
with DataRow objects  
    adding, [377-380](#), [393-394](#), [434](#)  
    deleting, [382-384](#), [396-397](#)  
    filtering and sorting, [368-373](#)  
    finding, [367-368](#)  
    modifying, [380-382](#), [395-396](#)  
    navigating, [428-429](#)  
with DataSet objects  
    merging into, [318-321](#)  
    populating, [311-316](#)  
events for, [347](#), [404-407](#)  
identity columns for, [384-386](#)  
modifying rows in, [373-384](#)  
properties and methods in, [345-347](#)  
restrictions on, [350-356](#)

DataTable property, [401](#)

DataTableCollection object, [141](#)

DataType property  
    in DataColumn, [349](#)  
    in schema columns, [219](#)

DataView objects, [143](#)  
child, [472-475](#)

creating, [456-458](#)

DataRowView. See [DataRowView objects](#)

filtering, [460-461](#)

properties and methods in, [453-456](#)

sorting with, [460](#)

VS.NET for, [478-480](#), [479-480](#)

working with, [458-460](#)

DataView property, [461](#)

DataViewManager objects, [475](#)

creating, [475-476](#)

working with, [476-478](#)

DataViewManager property, [454](#)

DataViewRowState enumeration, [368](#), [436](#), [455-456](#)

DataViewSettings property, [475](#)

DATEADD function, [116-117](#)

DATEDIFF function, [116-117](#)

DATENAME function, [116-117](#)

DATEPART function, [116-117](#)

dates

constraints for, [56](#)

T-SQL functions for, [116-117](#)

datetime type in SQL Server, [43](#), [51](#), [268-269](#)

DateTime value in SqlDbType, [241](#)

DAY function, [116-117](#)

DB (Database) lock type, [501](#)

db\_owner role, [36](#)

DBAs (database administrators), [8](#)

dbo user, [36](#), [36](#)

DDL (Data Definition Language) statements, [59](#)

with ExecuteNonQuery, [233-236](#)

working with, [89](#)

Deadlock.cs program, [511-514](#)

deadlocks in transactions, [509-514](#)

DEALLOCATE statement, [105](#)

Debug menu, [20](#)

decimal type in SQL Server, [43](#), [268-269](#)

Decimal value in SqlDbType, [241](#)

DECLARE statement, [96](#), [104](#)

declaring cursors, [104](#)

Default value

in CommandBehavior, [215](#)

in DataRowsVersion, [376](#)

in Persons table, [52](#)  
default values in SQL Server, [38](#)  
DefaultValue property, [350-351](#)  
DefaultViewManager property, [299](#)  
DEGREES function, [111](#)  
DELAY keyword, [103](#)  
Delete method  
    in DataRow, [348](#)  
    in DataRowView, [462](#)  
    in DataView, [455, 468](#)  
Delete option for row display, [178](#)  
DELETE permission, [38](#)  
DELETE statement, [62, 84-85, 85](#)  
    with browsers, [592, 592](#)  
    with ExecuteNonQuery, [230-233](#)  
    multiple, [286-288](#)  
DeleteCommand property  
    in DataAdapter, [376-377, 392, 398](#)  
    setting, [382](#)  
    in SqlDataAdapter, [294, 339](#)  
Deleted value  
    in DataRowState, [379](#)  
    in DataViewRowState, [368, 456](#)  
DeleteProduct stored procedure, [388-389](#)  
DeleteProduct2.sql stored procedure, [389-390](#)  
DeleteProductTrigger.sql script, [133-134](#)  
DeleteRule property, [421, 428, 439-440](#)  
deletes  
    DeleteRule property for, [439-440](#)  
    SQL Server for, [438, 439](#)  
deleting  
    DataRow objects, [382-384, 396-397, 435](#)  
    DataRowView objects, [468](#)  
    functions, [120](#)  
    rows, [84-85, 85](#)  
    stored procedures, [126](#)  
Depth property, [260](#)  
DESC keyword, [73, 74, 457](#)  
descriptions for columns, [52](#)  
Design Table, [41, 438](#)  
DesignTypeVisible property, [208](#)  
Detached value, [379](#)  
Diagram Pane, [49-50, 49](#)

diagrams

- in query builder, [49-50](#), [49](#)
- in SQL Server, [33](#), [33](#)

Dialog Editor category, [155](#)

DIFFERENCE function, [113](#), [115](#)

DiffGram value

- in XmlReadMode, [326](#)
- in XmlWriteMode, [323](#)

directives

- preprocessor, [161](#), [524](#)
- for XML, [571-572](#)

Dirty Reads, [492](#)

DiscountPrice function, [120-121](#)

DiscountPrice.sql program, [120-121](#)

display names, [78-79](#), [78](#)

DisplayDataRow method, [379](#), [396](#)

DisplayDataRows method, [607](#)

DisplayRow method, [231-232](#)

DisplayRows method, [497](#), [506](#)

Dispose method, [159](#)

DISTINCT keyword, [74-75](#), [75](#), [108](#)

dividing retrieved rows into blocks, [76-77](#), [77](#)

DML (Data Manipulation Language) statements, [59](#), [62](#)

Do not automatically recompute statistics option, [56](#)

document structure in XML, [578-579](#), [579-580](#)

documentation

- in SQL Server, [26](#), [27](#)
- in VS.NET, [23-26](#), [24-26](#)

DocumentElement property, [600](#)

DocumentType property, [600](#)

double type, [268](#)

Downloads link, [18](#)

DRI permission, [38](#)

DROP FUNCTION statement, [120](#)

DROP INDEX statement, [91](#)

DROP PROCEDURE statement, [126](#)

DROP TABLE statement, [90](#)

DropDownList controls, [525](#), [529](#)

duplicates in retrieved rows, [74-75](#), [75](#)

## Index

### E

Edit menu, [20](#)

Edit Relation dialog box, [451-452](#), [451](#)

editing DataList templates, [545](#)

element directive, [571](#)

elements

  in XML, [579](#)

  in XPath expressions, [581](#)

ELEMENTS option, [227](#), [576](#)

ELSE keyword, [97-98](#)

END keyword, [97](#)

EndEdit method

  in DataRow, [348](#)

  in DataRowView, [467](#)

EndInit method

  in DataSet, [301](#)

  in DataView, [455](#)

#endregion preprocessor directive, [161](#), [524](#)

Enforce constraint for INSERTs and UPDATEs option, [57](#)

Enforce constraint for replication option, [57](#)

Enforce relationship for INSERTs and UPDATEs option, [54](#)

Enforce relationship for replication option, [54](#)

EnforceConstraints property, [299](#), [351](#)

entering statements, [60-62](#), [61](#)

Enterprise Manager, [31-38](#), [31](#), [33-37](#)

  for functions, [120](#), [121](#)

  for indexes, [42](#)

  for queries, [48-50](#), [49-50](#)

  for relationships, [41](#)

  for stored procedures, [126](#)

  for tables, [51-52](#), [51-52](#)

  for triggers, [130](#)

environment variables, [11](#)

EOF property, [228](#)

equal operator (=) in WHERE clauses, [64-65](#)

@@ERROR function, [88-89](#), [118](#)

error messages, raising, [103](#)

Error value, [319](#)

errors in updates

  checking for, [411](#)

  fixing, [411-412](#)

Errors property

- in FillErrorEventArgs, [401](#)
- in SqlRowUpdatingEventArgs, [402](#)

event handlers, [196](#)

EventArgs class, [400](#)

events

- for Connection, [196-200](#)
- for DataAdapter, [399-404](#)
- for DataTable, [347, 404-407](#)
- in VS.NET, [203-204, 203-205](#)

exception handling, [9-10](#)

Exclusive (X) locking mode, [501](#)

EXEC permission, [38](#)

EXECUTE statement, [127](#)

ExecuteAddProduct.cs program, [249-250](#)

ExecuteAddProduct3.cs program, [254-255](#)

ExecuteDDL.cs program, [234-236](#)

ExecuteInsertUpdateDelete.cs program, [231-233](#)

ExecuteMultipleSelects.cs program, [284-285](#)

ExecuteMultipleSQL.cs program, [286-288](#)

ExecuteNonQuery method

- DDL statements with, [233-236](#)
- modify commands with, [230-233](#)
- in SqlCommand, [209](#)

ExecuteReader method, [8, 211](#)

- controlling command behavior with, [214-223](#)
- with SELECT statement, [212-214](#)
- in SqlCommand, [209](#)
- in SqlDataReader, [262-263](#)
- for TableDirect, [223-225](#)

ExecuteScalar.cs program, [226](#)

ExecuteScalar method, [211](#)

- with SELECT statement, [225-226](#)
- in SqlCommand, [209](#)

ExecuteSelect.cs program, [213-214](#)

ExecuteTableDirect.cs program, [224-225](#)

ExecuteTransaction.cs program, [237-239](#)

ExecuteXmlReader.cs program, [228-229](#)

ExecuteXmlReader method, [209, 211, 227-229](#)

executing commands, [207](#)

- Command objects for, [255-257, 256-257](#)
- ExecuteNonQuery for, [230-236](#)
- ExecuteReader for, [212-225](#)

ExecuteScalar for, [225-226](#)  
ExecuteXmlReader for, [227-229](#)  
parameters for, [239-245](#)  
SqlCommand objects for, [207-211](#)  
stored procedures, [245-255](#)  
transactions, [236-239](#)

executing multiple SQL statements, [282-288](#)

Executing value, [195](#)

EXP function, [111](#)

EXPLICIT mode for XML, [227](#), [570-574](#)

expressions in XPath, [580-582](#)

ExtendedProperties property

    in DataRelation, [424](#)

    in DataSet, [300](#)

    in ForeignKeyConstraint, [421](#)

    in UniqueConstraint, [418](#)

Extensible Markup Language Path (XPath)

    expressions in, [580-582](#)

    and XML document structure, [578-579](#), [579-580](#)

Extensible Stylesheet Language Transformation (XSLT), [582-586](#), [585](#)

Extent lock type, [501](#)

## Index

### F

failures in updates, [408-412](#)

FETCH statement, [104](#)

Fetching value, [195](#)

FieldCount property, [260](#)

File menu, [20](#)

Fill factor field, [55-56](#)

Fill method

  in DataAdapter, [353](#)

  in DataSet, [303-307](#)

  in SqlDataAdapter, [147, 171, 296](#)

FillError event

  in DataAdapter, [400-401](#)

  in SqlDataAdapter, [297](#)

FillEventArgs class, [400-401](#)

FillEventHandler method, [400](#)

FillSchema.cs program, [363-367](#)

FillSchema method

  in DataAdapter, [351, 362-367](#)

  in SqlDataAdapter, [296](#)

filtering

  DataRow objects, [368-373](#)

  DataView objects, [460-461](#)

Find A Service page, [626](#)

Find method, [455](#)

FindByCustomer method, [337](#)

FindByCustomerID method, [414](#)

FindFilterAndSortDataRows.cs program, [370-373](#)

finding

  DataRow objects, [367-368](#)

  DataRowView object indexes, [463](#)

  DataRowView objects, [464-466](#)

  documentation, [25](#)

FindingDataRowViews.cs program, [464-466](#)

FindRows method

  in DataRowView, [464-466](#)

  in DataView, [455](#)

firehose cursors, [259](#)

firing triggers, [127-128](#)

FirstChild property, [600](#)

FirstExample.cs program, [4-7](#)

compiling and running, [10-12](#)

explanation of, [7-10](#)

fixed length data types, [51](#)

float type in SQL Server, [43, 268-270](#)

Float value in SqlDbType, [241](#)

FLOOR function, [111-112](#)

Font Name property, [538](#)

Font Size property, [538](#)

Footer Text field, [537](#)

FOR XML clause, [227, 567-568](#)

for AUTO mode, [228, 570](#)

for BINARY BASE64 option, [576-578, 577](#)

for ELEMENTS option, [575](#)

for EXPLICIT mode, [570-574](#)

for RAW mode, [568-569, 569](#)

for XMLDATA option, [574-575](#)

foreach loops, [148, 218](#)

Forecolor property, [538](#)

foreign keys

integrity of, [85](#)

in Northwind database, [40-41, 41](#)

Order Details table, [47, 356](#)

Orders table, [46](#)

Products table, [48](#)

ForeignKeyConstraint objects, [143](#)

creating, [422-423](#)

properties of, [420-421](#)

for updates and deletes, [439-440](#)

Form1.cs program, [158-161, 337](#)

Form1\_Load method, [289-290, 336, 343, 414](#)

Format link for DataGrid controls, [538, 539](#)

formatting, XML stylesheets for, [594-596, 596](#)

forms

for Web

building, [527-532, 527-528](#)

controls for, [524-526](#)

for Windows applications

creating, [172-179, 172-179](#)

running, [182-183, 182](#)

Forms namespace, [161](#)

Formula field, [53](#)

ForXmlAuto.sql script, [570](#)

ForXmlAutoBinaryBase64.sql script, [578](#)

ForXmlAutoElements.sql script, [576](#)

ForXmlAutoXmlData.sql script, [575](#)

ForXmlExplicit.sql script, [573-574](#)

ForXmlRaw.sql script, [568-569](#), [569](#)

Fragment value, [326](#)

FROM clause, [63-64](#), [63](#)

FULL OUTER JOINs, [80](#)

full-text catalogs, [38](#)

functions in T-SQL, [107](#)

aggregate, [107-110](#), [109-110](#)

date and time, [116-117](#)

mathematical, [111-113](#)

string, [113-116](#)

system, [117-119](#)

user-defined

  inline table-valued, [121-122](#)

  multistatement table-valued, [122-124](#), [123-124](#)

  scalar, [119-121](#), [121](#)

  in SQL Server, [38](#)

  in T-SQL, [119](#)

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Index

### G

General properties and tab

for DataGrid controls, [535-536](#), [535](#)

for virtual directories, [587](#), [587](#)

Generate Dataset dialog box, [170](#), [170](#), [342](#), [342](#), [450](#), [450](#), [533](#), [544](#)

Generate Dataset link, [341](#)

Generate the SQL statements screen, [340](#), [340](#)

generic data classes, [141-143](#), [142](#)

Get\* methods for strongly typed column values, [268-275](#), [272](#)

Get Started link, [18](#)

GetBoolean method, [260](#)

GetByte method, [260](#)

GetBytes method, [260](#)

GetChanges method

  in DataSet, [301](#)

  in DataTable, [347](#)

GetChar method, [260](#)

GetChars method, [260](#)

GetChildRows method, [348](#), [424](#), [428-429](#)

GetColumnError method, [348](#)

GetColumnsInError method, [348](#)

GetDataTypeName method

  in DataReader, [272](#)

  in SqlDataReader, [260](#)

GETDATE function, [116-117](#)

GetDateTime method, [260](#)

GetDecimal method, [260](#)

GetDeleteCommand, [399](#)

GetDouble method, [260](#)

GetElementFromRow method, [607](#)

GetElementsById method, [602](#)

GetElementsByTagName method, [602](#)

GetEnumerator method, [455](#)

GetErrors method, [347](#)

GetFieldType method

  in DataReader, [272](#)

  in SqlDataReader, [260](#)

GetFillParameters method, [296](#)

GetFloat method, [260](#)

GetGuid method, [260](#)  
GetInsertCommand method, [399](#)  
GetInt16 method, [261](#)  
GetInt32 method, [261](#), [268](#), [271](#)  
GetInt64 method, [261](#)  
GetName method, [261](#)  
GetNamespaceOfPrefix method, [602](#)  
GetOrdinal method  
    in DataReader, [264](#)  
    in SqlDataReader, [261](#)  
GetParentRow method, [348](#), [424](#), **[429](#)**  
GetParentRows method, [348](#)  
GetPrefixOfNamespace method, [602](#)  
GetRowFromElement method, [607](#)  
GetSchemaTable method, [217](#), [261](#)  
GetSql\* methods, **[266-281](#)**  
GetSqlBinary method, [261](#)  
GetSqlBoolean method, [261](#)  
GetSqlByte method, [261](#)  
GetSqlDateTime method, [261](#)  
GetSqlDecimal method, [261](#)  
GetSqlDouble method, [261](#)  
GetSqlGuid method, [261](#)  
GetSqlInt16 method, [261](#)  
GetSqlInt32 method, [261](#)  
GetSqlInt64 method, [261](#)  
GetSqlMoney method, [261](#)  
GetSqlSingle method, [261](#)  
GetSqlString method, [261](#)  
GetSqlValue method, [261](#)  
GetSqlValues method, [261](#)  
GetString method, [262](#), [271](#)  
GetUpdateCommand, [399](#)  
GETUTCDATE function, [116-117](#)  
GetValue method, [262](#)  
GetValues method, [262](#)  
GetXml method  
    in DataSet, [301](#), [321](#), [443](#)  
    in XmlDocument, [603-604](#)

GetXmlSchema method, [301](#)

GOTO statement, [101-102](#)

granularity of locks, [500](#)

graphical user interfaces (GUIs), [153](#)

greater than operator (>) in WHERE clauses, [65](#)

Grid Lines property, [540](#)

Grid Pane, [49-50](#)

GridLayout, [522](#)

GROUP BY clause

  for blocks, [76-77](#), [77](#)

  with functions, [109-110](#)

GroupBox controls, [165](#)

grouping statements, [87-89](#), [88](#)

groups, restrictions on, [77](#), [78](#)

guest user, [36](#)

Guid type, [268](#), [270](#)

GUIDs for Persons table, [53](#)

GUIs (graphical user interfaces), [153](#)

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Index

### H

HandlingUpdateFailures.cs program, [408](#)

HasChanges method, [301](#)

HasChildNodes property, [600](#)

HasErrors property

  in DataRow, [348](#)

  in DataSet, [300](#)

  in DataTable, [346](#)

HAVING clause

  with functions, [109](#)

  for groups, [77, 78](#)

Header and Footer property, [536](#)

header and footer template, [545](#)

Header Image field, [537](#)

Header Text field, [537](#)

Headlines link, [18](#)

Help menu, [21, 26](#)

hidden code, viewing, [162](#)

hide directive, [571](#)

Horizontal Alignment property, [538](#)

HTML category in Toolbox, [156](#)

HTTP (Hypertext Transport Protocol)

  state in, [552](#)

  XML for, [586-596, 586-594, 596](#)

HttpCookie object, [552](#)

HyperLink controls, [525](#)

hyphens (-)

  for comments, [97](#)

  in pattern matching, [65](#)

## Index

### I

icon files, [163](#)

ID directive, [572](#)

ID property

  for forms, [522](#)

  for Label controls, [528](#)

IDENTITY clause, [128](#)

identity columns

  generating, [351](#)

  retrieving values of, [384-386](#)

Identity field, [52](#)

@@IDENTITY function, [118, 125](#)

Identity Increment field, [52](#)

Identity Seed field, [52](#)

IDREF directive, [572](#)

IDREFS directive, [572](#)

IF keyword, [97-98](#)

Ignore duplicate key option, [55](#)

Ignore value, [319](#)

IgnoreSchema value

  in XmlReadMode, [326](#)

  in XmlWriteMode, [323](#)

IIS (Internet Information Server)

  installing, [518](#)

  XML support for, [586-590, 586-590](#)

IIS Virtual Directory Management for SQL Server console, [586-587, 586, 590](#)

Image controls, [525](#)

image type in SQL Server, [43, 269](#)

Image value in SqlDbType, [241](#)

ImageButton controls, [525](#)

images, functions for, [107](#)

Implementation property, [600](#)

ImportNode method, [602](#)

IN operator, [68-69, 69](#)

Index column for transaction locks, [503](#)

Index Filegroup field, [55](#)

Index Results window, [24, 24](#)

Index tab, [24](#)

Index window, [24](#)

indexes  
creating, [55-56](#), [55](#), [90-91](#)  
for DataRowView objects, [463](#)  
for documentation, [24](#), [24](#)  
dropping, [91](#)  
in Northwind database, [41-42](#), [42](#)

InferSchema value, [326](#)

infinity sign for relationships, [40](#)

InfoMessage.cs program, [198-200](#)

InfoMessage event, [189](#), **[198-200](#)**

InitializeComponent method, [161](#), [524](#)

inline table-valued functions, [119](#), **[121-122](#)**

InnerText property, [600](#)

InnerXml property, [600](#)

INSERT permission, [38](#)

INSERT statement, [62](#), **[82](#)**, [83](#)  
with browsers, [592](#), [592](#)  
with ExecuteNonQuery, [230-233](#)  
integrity with, [85-86](#)  
multiple, [286-288](#)

Insert Table dialog box, [546](#), [547](#)

InsertAfter method, [602](#)

InsertBefore method, [602](#)

InsertCommand property  
in DataAdapter, **[373-374](#)**, [391](#), [398](#)  
setting, [391](#)  
in SqlDataAdapter, [294](#), [339](#)

InsertProductTrigger.sql script, **[129-131](#)**

instances, Query Analyzer for, [60](#), [60](#)

int type in SQL Server, [43](#), [51](#), [268-269](#)

Int value in SqlDbType, [241](#)

Int16 type, [269](#)

Int32 type, [268](#)

Int64 type, [268](#)

integrity of databases, **[85-86](#)**

Intent exclusive (IX) locking mode, [501](#)

Intent shared (IS) locking mode, [501](#)

internal access modifier, [161](#)

Internet Information Server (IIS)  
installing, [518](#)  
XML support for, **[586-590](#)**, [586-590](#)

IS (Intent shared) locking mode, [501](#)

IS NULL operator  
with joins, [81](#)  
working with, [70](#), [70](#)

IsAliased property, [219](#)

IsAutoIncrement property, [219](#)

IsClosed property, [260](#)

IsContactNameNull method, [337](#), [414](#)

ISDATE function, [118-119](#)

IsDBNull method  
in DataReader, [281](#)  
in SqlDataReader, [262](#)

IsEdit property, [461](#)

IsExpression property, [219](#)

IsHidden property, [220](#)

IsIdentity property, [219](#)

IsKey property, [219](#)

IsLong property, [220](#)

IsNew property, [461](#)

ISNULL function, [118-119](#)

IsNull method, [349](#)

IsNullable property, [243](#)

ISNUMERIC function, [118-119](#)

isolation levels for transactions, [492-500](#), [495](#)

IsolationLevel enumeration, [493](#)

IsolationLevel property, [486](#)

IsPostBack property, [534](#)

IsPrimaryKey property, [418](#)

IsReadOnly property, [600](#)

IsReadOnly value, [220](#)

IsRowGuid field, [53](#)

IsRowVersion property, [219](#)

IsUnique property, [219](#)

Italic property, [538](#)

Item Templates editor, [546](#), [546](#)

ItemArray property, [348](#)

Items property, [528](#)

IX (Intent exclusive) locking mode, [501](#)

**Team LiB****◀ PREVIOUS** **NEXT ▶**

## Index

### J

joins

for populating objects, [303-304](#)

retrieving rows from, [79-81](#), [80](#)

working with, [283-285](#)

**Team LiB****◀ PREVIOUS** **NEXT ▶**

**Team LiB****◀ PREVIOUS** **NEXT ▶**

## Index

### K

Key lock type, [501](#)

KeyInfo value, [215](#)

keys. See [foreign keys](#); [primary keys](#)

**Team LiB****◀ PREVIOUS** **NEXT ▶**

## Index

### L

labels and Label controls, [165](#)  
in T-SQL, [101-102](#)  
for Web forms, [524](#), [527-528](#)  
for Windows applications, [156-158](#), [157-158](#), [180](#)

language attribute, [521](#)

last function, [581](#)

last one wins concurrency, [375](#), [441-442](#)

LastChild property, [600](#)

LEFT function, [113](#), [115](#)

LEFT OUTER JOINs, [80-81](#)

LEN function, [113](#), [115](#)

less than operator (<) in WHERE clauses, [65](#)

LIKE operator

    in filters, [461](#)

    in pattern matching, [65-68](#), [66-68](#)

LinearLayout, [522](#)

LinkButton controls, [525](#)

LinkLabel controls, [165](#)

ListBox controls, [165](#), [525](#)

ListChanged event

    in DataView, [455](#)

    in DataViewManager, [475](#)

ListChangedEventArgs method

    in DataView, [455](#)

    in DataViewManager, [475](#)

ListItem Collection Editor, [528](#), [528](#)

lists in retrieving rows, [68-69](#), [69](#)

ListView controls, [288-290](#), [289](#)

Literal controls, [526](#)

Load method

    in XmlDocument, [607](#)

    in XmlNode, [602](#)

LoadDataRow method, [347](#)

loading statements, [62](#)

LoadXml method, [602](#), [604](#)

Locale property, [300](#)

LocalName property, [600](#)

Location field, [154](#)

Lock Type column, [503](#)

locks for transactions, [500](#)  
blocking, [504-505](#), [504-505](#)  
deadlocks in, [509-514](#)  
modes for, [501-502](#)  
serializable/repeatable read transactions, [506-509](#)  
timeouts for, [505](#)  
types of, [500-501](#)  
viewing information for, [502-503](#), [503](#)

Locks/Object node, [502](#)

Locks/Process ID node, [502](#)

LOG function, [111](#)

LOG10 function, [111](#)

logical operators, [71-72](#), [71-72](#)

long type, [268-269](#)

loops in T-SQL, [99-101](#)

LOW value for deadlock priority, [510](#), [514](#)

LOWER function, [113](#), [115](#)

LTRIM function, [113](#)

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Index

### M

main method, [181](#)

MainMenu controls, [165](#)

Manage Constraints option, [56](#)

Manage Indexes option, [42](#)

Manage Indexes/Keys option, [55](#)

Manage Relationships option, [438](#)

Manage Triggers option, [130](#)

managed provider classes, [138-141](#), [139](#)

Management folder, [32](#)

Mapped value, [362](#)

mapping tables and columns, [328-331](#), [329](#)

Mappings.cs program, [330-331](#)

marking modifications, [382](#)

mathematical functions, [111-113](#)

MAX function, [108](#), [110](#)

max pool size setting, [192](#)

Maximum Characters Per Column field, [569](#)

MaxLength property, [350-351](#), [357](#)

Merge.cs program, [319-321](#)

Merge method, [301](#), [318](#)

MergeFailed method, [302](#)

merging objects, [318-321](#)

Message property, [10](#)

Meta Data Services folder, [32](#)

metadata functions, [107](#)

Microsoft Passport accounts, [628](#), [628](#)

MIN function, [108](#), [110](#)

minus signs (-)

for comments, [97](#)

in pattern matching, [65](#)

MissingMappingAction property, [295](#)

MissingSchemaAction enumeration, [318-319](#)

MissingSchemaAction property, [295](#)

Mode column for transaction locks, [503](#)

Mode property, [538](#)

modes, XML, [227](#), [568-569](#), [569](#)

Modified value, [379](#)

ModifiedCurrent value, [368](#), [456](#)

ModifiedOriginal value, [368](#), [456](#)

ModifyDataRow method, [380-382](#), [395-396](#), [409-410](#)

modifying data, [354](#)

    automatic generation of SQL statements, [398-399](#)

    DataAdapter events for, [399-404](#)

    DataColumn objects for, [349-350](#), [356-362](#)

    DataRow objects for, [347-349](#)

    DataRowView objects for, [467-472](#)

    DataTable objects for. See [DataTable objects](#)

    FillSchema method for, [362-367](#)

    rows, [83-84](#), [84](#)

    stored procedures for, [386-397](#)

    with strongly typed DataSets, [413-415](#)

    tables, [90](#)

    transactions with, [412-413](#)

    update failures in, [408-412](#)

    VS.NET code, [21-22](#)

ModifyRowsUsingUPDATE method, [409-410](#), [412](#)

money type in SQL Server, [43](#), [269](#)

money value in SqlDbType, [241](#)

MONTH function, [116-117](#)

MS\_POSITIONING attribute, [522](#)

multi-line comments, [97](#)

multiple conditions, [71-72](#), [71-72](#)

multiple DataAdapter objects, [316-318](#)

multiple DataTable objects, [311-316](#)

multiple SQL statements, [282-288](#)

multiple tables

    for populating objects, [303-304](#)

    retrieving rows from, [79-81](#), [80](#)

MultipleDataTables.cs program, [312-314](#)

MultipleDataTables2.cs program, [314-316](#)

MultipleDataTables3.cs program, [316-318](#)

multistatement table-valued functions, [119](#), [122-124](#), [123-124](#)

My Profile link, [19](#)

myButton\_Click method, [157](#), [163](#)

MyDataSet.cs program, [335-336](#)

MyDataSet.xsd file, [332-335](#), [333](#)

MySqlConnection.cs program, [191-192](#)

myXmlFile.xml file, [322-323](#)

myXmlFile2.xml file, [323-324](#)

myXmlSchemaFile.xml file, [324-325](#)

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Index

### N

Name property, [600](#)

NameRequiredFieldValidator controls, [530](#), [530](#)

names

for DataSets, [173-174](#)

for projects, [154](#)

in retrieving rows, [78-79](#), [78](#)

of server computers, [31](#)

spaces in, [64](#)

for tables, [52](#), [52](#), [64](#)

Namespace property, [300](#)

namespaces, [141](#)

NamespaceURI property, [600](#)

NameTable property, [600](#)

navigating

DataRow objects, [428-429](#)

row display, [178](#)

Navigation Controls, [178](#)

NCHAR function, [113](#)

nchar type in SQL Server, [43](#), [51](#), [269](#)

NChar value in SqlDbType, [241](#)

negative signs (-)

for comments, [97](#)

in pattern matching, [65](#)

nested foreach loops, [218](#)

Nested property, [424](#)

nested XML, [443-447](#)

NestedXml.cs program, [443-445](#)

nestedXmlFile.xml file, [446-447](#)

New Project dialog box, [19](#), [19](#)

for ASP.NET applications, [518](#), [518](#), [527](#)

for Web services, [613-614](#), [614](#)

for Windows applications, [154](#), [154](#), [448](#), [448](#)

New Stored Procedure option, [126](#)

New Table option, [51](#)

New User Defined Function option, [120](#)

New Virtual Directory Properties window, [587](#)

NewCustomersRow method, [337](#), [413](#)

NEWID function, [53](#), [118](#)

NewPageIndex property, [541](#)

NewRow method  
in DataTable, [347](#), [377](#)  
in myDataTable, [380](#)

Next Page Button Text property, [538](#)

NextResult method, [262](#), [283](#)

NextSibling property, [600](#)

node sets, [579](#)

node subtrees, [579](#)

NodeChanged event, [603](#)

NodeChanging event, [603](#)

NodeInserted event, [603](#)

NodeInserting event, [603](#)

NodeRemoved event, [603](#)

NodeRemoving event, [603](#)

nodes in XML, [578-580](#)

NodeType property, [601](#)

None value  
in DataGridViewRowState, [369](#), [456](#)  
in Rule, [439](#)

nonNestedXmlFile.xml file, [445-446](#)

nonrepeatable reads, [492](#)

NORMAL value for deadlock priority, [510](#)

Northwind database, [4](#), [\*\*39-40\*\*](#), [39](#)  
column types in, [\*\*42-44\*\*](#)  
Columns tab for, [\*\*52-53\*\*](#)  
constraints in, [\*\*56-57\*\*](#), [57](#)  
Customers table in, [\*\*44-45\*\*](#), [44](#)  
foreign keys in, [\*\*40-41\*\*](#), [41](#)  
indexes in, [\*\*41-42\*\*](#), [42](#), [\*\*55-56\*\*](#), [55](#)  
null values in, [41](#)  
Order Details table in, [47](#), [47](#)  
Orders table in, [\*\*45-47\*\*](#), [46](#)  
permissions in, [53](#), [53](#)  
primary keys in, [40](#), [53](#)  
Products table in, [29-30](#), [48](#), [48](#)  
queries in, [\*\*48-50\*\*](#), [\*\*49-50\*\*](#)  
relationships in, [\*\*40-41\*\*](#), [41](#), [54](#), [54](#)  
tables for, [\*\*51-52\*\*](#), [\*\*51-52\*\*](#)

not-equal operator (<>) in WHERE clauses, [65](#)

NOT NULL constraint, [90](#)

NOT operator, [\*\*70-72\*\*](#)

ntext type in SQL Server, [43](#), [269](#)

Ntext value in SqlDbType, [241](#)

null values

  in Northwind database, [41](#)

  reading, [281-282](#)

  retrieving rows with, [70](#), [70](#)

  for variables, [96](#)

NULLIF function, [118](#)

Number property, [10](#)

number signs (#)

  in pattern matching, [65](#)

  in URLs, [591](#)

Numeric Buttons property, [538](#)

numeric type, [43](#), [269](#)

NumericPrecision property, [219](#)

NumericScale property, [219](#)

nvarchar type in SQL Server, [43](#), [51](#), [269](#)

NVarChar value in SqlDbType, [241](#)

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Index

### O

Object Browser, [60](#), [60](#)

Object column for transaction locks, [503](#)

ODBC managed provider classes, [138](#)

Odbc namespace, [141](#)

OdbcCommand class, [139](#)

OdbcCommandBuilder class, [140](#), [398](#)

OdbcConnection class, [139](#)

OdbcDataAdapter class, [140](#)

OdbcDataReader class, [140](#)

OdbcParameter class, [140](#)

ODBCParameterCollection class, [140](#)

OdbcTransaction class, [140](#), [236](#), [486](#)

OLE DB managed provider classes, [138](#)

OleDb namespace, [12-13](#), [141](#)

OleDbCommand class, [13](#), [139](#)

OleDbCommandBuilder class, [140](#), [398](#)

OleDbConnection class, [12](#), [14-15](#), [139](#), [200](#)

OleDbConnectionAccess.cs program, [13-14](#)

OleDbConnectionOracle.cs program, [15-17](#)

OleDbDataAdapter class, [140](#)

OleDbDataReader class, [13](#), [140](#)

OleDbParameter class, [140](#)

OleDbParameterCollection class, [140](#)

OleDbTransaction class, [140](#), [236](#), [486](#)

one-to-many relationships, [40](#)

OnInit method, [524](#)

Online Community link, [18](#)

Open method

  in Connection, [190](#)

  in mySqlConnection, [378](#)

  in SqlConnection, [8](#), [147](#), [188](#), [212](#)

Open Query File dialog box, [62](#)

OPEN statement, [104](#)

Open value, [195](#)

opening

  cursors, [104](#)

  database connections, [190-192](#)

files, [62](#)

SqlConnection objects, [8](#)

OPENXML function, [596-599](#), [599](#)

optimistic concurrency, [375](#)

OR operator, [71-72](#), [72](#)

Oracle databases, connecting to, [14-17](#)

ORDER BY clause, [72-73](#), [73-74](#)

Order Details table

contents of, [47](#), [47](#)

foreign keys for, [356](#)

primary keys for, [355](#)

Orders table

contents of, [45-47](#), [46](#)

primary keys for, [354](#)

ordersDA objects, up, [431-433](#)

ordersDT objects with DataRow objects

adding, [434](#)

deleting, [435](#)

pushing changes in, [435-437](#)

updating, [434](#)

Ordinal property, [350](#)

Original value, [376](#)

OriginalRows value, [369](#), [456](#)

Outer joins, [80](#)

OuterXml property, [601](#)

Overline property, [538](#)

Owner column for transaction locks, [503](#)

OwnerDocument property, [601](#)

**Team LiB**

[◀ PREVIOUS](#)

[NEXT ▶](#)

## Index

### P

PacketSize property, [188](#)

Pad Index option, [56](#)

Page lock type, [501](#)

Page Size property, [538](#)

PageIndexChanged handler, [540-543](#), [540](#), [542](#)

Paging properties for DataGrid controls, [537-538](#), [537](#)

Panel controls, [165](#), [525](#)

Parameter classes, [140](#)

ParameterCollection classes, [140](#)

parameters, [239](#)

    adding, [241-242](#)

    executing commands with, [244-245](#)

    placeholders for, [240](#)

    setting values of, [243](#)

Parameters property

    in mySqlCommand, [242](#)

    in SqlCommand, [208](#)

parent-child relationships, [40](#), [417](#)

parent rows, primary keys in, [437-443](#), [439](#)

parent tables, [40](#)

ParentColumns property, [424](#)

ParentKeyConstraint property, [424](#)

ParentNode property, [601](#)

ParentTable property, [424](#)

Parse method, [553](#)

passwords

    ConnectionString property for, [181](#), [182](#)

    in integrated security, [190](#)

    for Oracle database connections, [15](#)

    saving, [202](#)

    for SqlConnection objects, [167](#)

    in Windows applications, [181](#), [182](#)

Path environment variable, [11](#)

PATINDEX function, [113](#)

pattern matching, [65-68](#), [66-68](#)

pauses

    SQL Server, [31](#)

    in T-SQL, [102](#)

percent signs (%)

in filters, [461](#)

in pattern matching, [65](#)

in URLs, [591](#)

PerformReadCommittedTransaction method, [497](#)

PerformSerializableTransaction method, [497](#)

periods (.) in XPath expressions, [581](#)

permissions

  in Persons table, [53](#), [53](#)

  for SQL Server roles, [37-38](#), [37](#)

persist security info setting, [202](#)

Persons table

  columns in, [52-53](#)

  constraints for, [56-57](#), [57](#)

  creating, [51-52](#)

  indexes in, [55-56](#), [55](#)

  permissions in, [53](#), [53](#)

  primary keys for, [53](#)

  relationships for, [54](#), [54](#)

phantom rows, [492](#)

PI function, [111-112](#)

PictureBox controls, [165](#)

PlaceHolder controls, [525](#)

placeholders for parameters, [240](#)

pooling connections, [192-195](#)

PopulateDataSetUsingProcedure.cs program, [310-311](#)

PopulateDataSetUsingRange.cs program, [307-308](#)

PopulateDataSetUsingSelect.cs program, [305-306](#)

populating DataSet objects, [433-434](#)

  DataAdapter objects for, [316-318](#)

  DataTable objects for, [311-316](#)

  ranges of rows for, [306-308](#)

  SELECT statements for, [302-306](#)

  stored procedures for, [309-311](#)

Position property, [538](#)

POWER function, [111-112](#)

Precision field, [52](#)

Prefix property

  in DataSet, [300](#)

  in XmlDocument, [601](#)

Prepare method, [209](#)

PrependChild method, [602](#)

preprocessor directives, [161](#), [524](#)

PreserveWhitespace property, [601](#)

Preview Data link, [341](#)

Previous Page Button Text property, [538](#)

PreviousSibling property, [601](#)

primary keys, [42](#), [89-90](#)

integrity of, [85-86](#)

in Northwind database, [40](#)

in Order Details table, [47](#), [355](#)

in Orders table, [46](#), [354](#)

in Persons table, [53](#)

in Products table, [48](#), [353](#)

updating, [437-443](#), [439](#)

PrimaryKey property

in DataTable, [346](#), [380](#)

in Order Details table, [355](#)

in Orders table, [354](#)

in Products table, [353](#)

PRINT statement, [98](#)

private access modifier, [161](#)

ProductAudit.sql script, [128](#)

ProductCursor.sql script, [106-107](#)

Products table

contents of, [29-30](#), [48](#), [48](#)

primary key for, [353](#)

ProductsToBeReordered function, [121-122](#)

ProductsToBeReordered.sql script, [121-122](#)

ProductsToBeReordered2 function, [122-124](#)

ProductsToBeReordered2.sql script, [122-124](#)

programs, compiling and running, [10-12](#), [23](#), [23](#)

Project menu, [20](#)

projects

creating, [19](#), [19](#)

names for, [154](#)

properties, viewing, [92](#), [94](#)

Properties dialog box

for DataGrids, [535](#), [535](#)

for Windows applications, [156](#)

Property Builder link, [535](#)

Proposed value, [376](#)

ProposedValue property, [405](#)

protected access modifier, [161](#)

protected internal access modifier, [161](#)

Provider tab, [201](#), [201](#)

ProviderType property, [219](#)

public access modifier, [161](#)

public role, [36](#), [37](#)

PushChangesToDatabase method, [437](#)

PushChangesUsingProcedures.cs program, [397](#)

pwd element, [7](#)

**Team LiB**

**◀ PREVIOUS** **NEXT ▶**

**Team LiB****◀ PREVIOUS** **NEXT ▶**

## Index

### Q

Query Analyzer

for inserting rows, [82](#), [83](#)

for instances, [60](#), [60](#)

for statements

    entering and running, [60-62](#), [61](#)

    saving and loading, [62](#)

for T-SQL, [96](#), [96](#)

for XML, [569](#), [569](#)

query builder, [49](#), [49](#)

    with Command objects, [257](#), [257](#)

    for Windows applications, [168](#), [169](#)

Query window, [60-62](#), [60](#)

question marks (?) in URLs, [591](#)

QUOTENAME function, [114](#)

**Team LiB****◀ PREVIOUS** **NEXT ▶**

## Index

### R

RAD (rapid application development) tools, [3](#)

RADIANS function, [111](#)

RadioButton controls, [165](#), [525](#)

RadioButtonList controls, [525](#), [528-529](#)

RAISERROR statement, [103](#)

RAND function, [111](#)

ranges

  in populating objects, [306-308](#)

  in retrieving rows, [69](#)

RangeValidator controls, [526](#)

rapid application development (RAD) tools, [3](#)

RAW mode for XML, [227](#), [568-569](#), [569](#)

RDBMSs (relational database management systems), [30](#)

READ COMMITTED isolation level, [492-493](#)

Read method

  in mySqlDataReader, [212](#), [283](#)

  in SqlDataReader, [262-263](#)

  in XmlReader, [228](#)

READ UNCOMMITTED isolation level, [492](#), [494](#)

ReadCommitted value, [493](#)

reading data, [259](#)

  multiple statements for, [282-288](#)

  null values, [281-282](#)

  SqlDataReader objects for, [259-262](#)

    creating, [262-263](#)

    reading rows from, [263-266](#)

  strongly typed column values, [266-281](#), [332-337](#), [333-335](#), [337](#)

  in VS.NET, [288-291](#), [289-291](#)

  in XML, [325-328](#)

ReadNode method, [602](#)

ReadOnly property, [350-351](#)

ReadOuterXml method, [228](#)

ReadSchema value, [326](#)

ReadUncommitted value, [493](#)

ReadXml method, [301](#), [321](#), [325-326](#)

ReadXmlSchema method, [301](#), [321](#)

real type in SQL Server, [43](#), [270](#)

Real value in SqlDbType, [241](#)

RecordsAffected property

in SqlDataReader, [260](#)  
in SqlRowUpdatedEventArgs, [403](#)

references in Solution Explorer, [163](#)

#region preprocessor directive, [161](#), [524](#)

Register Your XML Web Service Today link, [626](#)

registering Web services, [626-630](#), [627-631](#)

RegularExpressionValidator controls, [526](#)

RejectChanges method  
    in DataRow, [349](#)  
    in DataSet, [301](#)  
    in DataTable, [347](#), [468](#)

RelatedColumns property, [421](#)

RelatedTable property, [421](#)

relational database management systems (RDBMSs), [30](#)

relational databases, [29](#), [40](#)

RelationName property, [424](#)

Relations property, [143](#), [300](#)

relationships, [39-40](#), [39](#), [417-418](#)  
    creating, [54](#), [54](#)  
    DataRelation objects for, [143](#)  
        constraints in, [427-428](#)  
        creating and using, [424-429](#)  
        properties for, [423-424](#)  
    ForeignKeyConstraint objects for, [420-423](#)  
    modifying rows, [429-437](#)  
    nested XML for, [443-447](#)  
    in Northwind database, [40-41](#), [41](#)  
    primary key updates for, [437-443](#), [439](#)  
    UniqueConstraint objects for, [418-420](#)  
    VS.NET for, [448-452](#), [448-452](#)

Relationships tab, [438](#), [439](#)

RemoveAll method, [603](#)

RemoveChild method, [603](#)

RemoveCustomersRow method, [337](#), [413](#)

RemoveDataRow method, [383](#), [396-397](#)

removing  
    DataRow objects, [382-384](#), [396-397](#), [435](#)  
    DataRowView objects, [468](#)  
    functions, [120](#)  
    rows, [84-85](#), [85](#)  
    stored procedures, [126](#)

REPEATABLE READ isolation level, [492](#)

repeatable read transactions, [506-509](#)

RepeatableRead value, [493](#)

Repeater controls, [525](#)  
ReplaceChild method, [603](#)  
REPLICATE function, [114](#)  
Replication folder, [32](#)  
Request object, [553](#)  
RequiredFieldValidator controls, [526](#)  
Reset method, [301](#)  
ResetCommandTimeout method, [209](#)  
Resource column for transaction locks, [503](#)  
Response object, [552](#)  
restrictions  
    on DataColumn objects, [356-362](#)  
    on DataRelation objects, [427-428](#)  
    on DataTable objects, [352-353](#)  
    FillSchema for, [362-367](#)  
    in Persons table, [56-57, 57](#)  
    in retrieving rows, [64-65, 64, 77, 78](#)  
results sets  
    stored procedures not returning, [245-252](#)  
    stored procedures returning, [252-255](#)  
Retrieve Customers link, [624](#)  
Retrieve Data From Table option, [92](#)  
RetrieveCustomers method, [616, 621-622, 623](#)  
retrieving data, [63-64, 63](#)  
    calculated fields in, [79](#)  
    combining in, [76, 76](#)  
    from cursors, [104-105](#)  
    display names and aliases in, [78-79, 78](#)  
    dividing into blocks, [76-77, 77](#)  
    duplicates in, [74-75, 75](#)  
    group restrictions in, [77, 78](#)  
    identity column values, [384-386](#)  
    lists in, [68-69, 69](#)  
    multiple conditions for, [71-72, 71-72](#)  
    from multiple tables, [79-81, 80](#)  
    NOT operator in, [70](#)  
    with null values, [70, 70](#)  
    pattern matching in, [65-68, 66-68](#)  
    ranges in, [69](#)  
    restrictions in, [64-65, 64](#)  
    sorting in, [72-73, 73-74](#)  
    TOP N, [74, 75](#)  
    from views, [81-82](#)  
    as XML, [227-229](#)  
RETURN statement, [102](#)

REVERSE function, [114](#)

RID (Row) lock type, [501](#)

RIGHT function, [114-115](#)

RIGHT OUTER JOINS, [80](#)

roles, [36-38](#), [37](#)

Rollback method, [486](#), [488](#)

ROLLBACK TRANSACTION statement, [87-89](#), [503](#)

root nodes in XML, [578-580](#)

ROUND function, [111-112](#)

Row (RID) lock type, [501](#)

Row property

    in DataColumnChangeEventArgs, [405](#)

    in DataRowChangeEventArgs, [406](#)

    in DataRowView, [461](#)

    in SqlRowUpdatingEventArgs, [402](#)

RowChanged event, [347](#), [404-407](#)

RowChangedEventHandler method, [406](#)

RowChanging event, [347](#), [404-407](#)

RowChangingEventHandler method, [406](#)

@@ROWCOUNT function, [118](#)

RowDeleted event, [347](#), [404](#), [407](#)

RowDeletedEventHandler method, [407](#)

RowDeleting event, [347](#), [404](#), [407](#)

RowDeletingEventHandler method, [407](#)

RowFilter property, [454](#), [460](#), [476](#)

rows, [29](#)

    adding, [82](#), [83](#)

    modifying, [83-84](#), [84](#), [373-384](#)

    ranges of, [69](#), [306-308](#)

    removing, [84-85](#), [85](#)

    retrieving. See [retrieving data](#)

    sorting, [72-73](#), [73-74](#)

    SqlDataReader objects for, [263-266](#)

    storing locally, [144-151](#)

    in XML, [568](#)

Rows property

    in DataRow, [142](#)

    in DataTable, [346](#)

    in myDataTable, [304](#)

RowState property

    in DataRow, [348](#)

    in myNewDataRow, [379](#)

RowStateFilter property, [454](#)

RowUpdated event  
  in DataAdapter, [403-404](#)  
  in SqlDataAdapter, [297](#)

RowUpdatedEventHandler method, [403-404](#)

RowUpdating event  
  in DataAdapter, [401-403](#)  
  in SqlDataAdapter, [297](#)

RowUpdatingEventHandler method, [403](#)

RowVersion property, [462](#)

RTRIM function, [114-115](#)

Rule enumeration, [439](#)

rules in SQL Server, [38](#)

Run method, [162-163](#)

RunAddCustomers.sql script, [598-599](#), [599](#)

running

  forms, [157](#), [158](#), **182-183**, [182](#)

  programs, [10-12](#), [23](#), [23](#)

  statements, [8](#), [60-62](#), [61](#)

  stored procedures, [127](#)

**Team LiB**

◀ PREVIOUS    NEXT ▶

## Index

### S

S (Shared) locking mode, [501](#)

Save method

    in SqlTransaction, [486](#), [488](#)

    in XmlDocument, [603-604](#)

Save Query dialog box, [62](#)

SAVE TRANSACTION statement, [486-487](#)

Savepoint.cs program, [489-492](#)

Savepoint.sql script, [487-488](#), [489](#)

savepoints

    SqlTransaction objects for, [488-492](#)

    T-SQL for, [486-488](#), [489](#)

saving

    passwords, [202](#)

    statements, [62](#)

    tables, [52](#), [52](#)

sbyte type, [269](#)

scalar user-defined functions, [119-121](#), [121](#)

Scale field, [52](#)

Schema Editor, [450](#), [451](#)

Schema modification (Sch-M) locking mode, [502](#)

schema relationships, [450-452](#), [451-452](#)

Schema stability (Sch-S) locking mode, [502](#)

SchemaOnly value, [215](#), [217-223](#)

SchemaOnlyCommandBehavior.cs program, [220-223](#)

schemas, [30](#)

SchemaType enumeration, [362](#)

SCOPE\_IDENTITY function, [385](#)

SDK (Software Development Kit)

    for documentation, [23-25](#), [24-25](#)

    installing, [4](#)

Search Online link, [18](#)

Search Results window, [25](#), [25](#)

Search tab, [25](#)

Search window, [25](#)

searching

    for DataRow objects, [367-368](#)

    for DataRowView object indexes, [463](#)

    for DataRowView objects, [464-466](#)

    in documentation, [25](#)

Security folder, [32](#)

security functions, [107](#)

Security tab, [587](#), [588](#)

Select method, [347](#)

SELECT permission, [38](#)

SELECT statement, [61-62](#)

    AS clause, [78-79](#), [78](#)

    BETWEEN operator, [69](#)

    with browsers, [590-591](#), [591](#)

    calculated fields in, [79](#)

    DISTINCT keyword, [74-75](#), [75](#)

    ExecuteReader with, [212-223](#)

    ExecuteScalar with, [225-226](#)

    ExecuteXMLReader with, [227-229](#)

    FROM clause, [63-64](#), [63](#)

    GROUP BY clause, [76-77](#), [77](#)

    HAVING clause, [77](#), [78](#)

    IN operator, [68-69](#), [69](#)

    IS NULL operator, [70](#), [70](#)

    JOIN operator, [79-81](#), [80](#)

    LIKE operator, [65-68](#), [66-68](#)

    logical operators in, [71-72](#), [71-72](#)

    multiple, [282-288](#)

    NOT operator, [70](#)

    ORDER BY clause, [72-73](#), [73-74](#)

    for populating objects, [302-306](#)

    running, [8](#)

    storing rows locally with, [144-151](#)

    TOP keyword, [74](#), [75](#)

    UNION operator, [76](#), [76](#)

    views in, [81-82](#)

    WHERE clause, [64-65](#), [64](#)

    for XML, [567-568](#)

SelectCommand property

    in DataAdapter, [398](#)

    in SqlDataAdapter, [146](#), [168](#), [168](#), [296](#), [339](#)

Selected property

    for DropDownList controls, [529](#)

    for RadioButtonList controls, [529](#)

SelectIntoDataSet.cs program, [148-149](#)

SelectNodes method, [603-604](#)

SelectSingleNode method, [603-604](#), [607](#)

semicolons (;) for multiple statements, [312](#)

SequentialAccess value, [215](#)

SERIALIZABLE isolation level, [492-494](#)

serializable transactions, blocking, [506-509](#)

Serializable value, 493

server element in connection strings, 7

Server Explorer, 91-92

servers, Web state information on, 556-559

ServerVersion property, 188

Service Manager tool, 30-31, 31

services, Web, 613

creating, 613-617, 614

registering, 626-630, 627-631

testing, 621-622, 622-623

using, 624-626, 624-626

VS.NET for, 17

WSDL files for, 617-621, 618

Session objects, 556-558

SessionObjectTest.aspx file, 557-558

SET DEADLOCK\_PRIORITY statement, 510, 514

SET LOCK\_TIMEOUT statement, 505

SET NOCOUNT ON statement, 129, 389-390

SET TRANSACTION ISOLATION LEVEL statement, 493-494

SetContactNameNull method, 414

SetDefault value, 439

SetNull method in DataRow, 349

SetNull value in Rule, 439

SetParentRow method, 349

Settings tab, 589, 589

setup.exe file, 4

SetupCustomersDA method, 431

Shared (S) locking mode, 501

Shared with intent exclusive (SIX) locking mode, 502

shopping cart application, 560, 560

    Buy button in, 560-562, 561-562

    code for, 563-565

    shopping cart in, 563, 563

short type, 269-270

Show Diagram option, 92

Show Navigation Buttons property, 538

Show SQL Pane option, 92

SIGN function, 111

SIN function, 112

single-line comments, 97

Single type, [268](#)

SingleResult value, [215](#)

SingleRow command behavior, [215-217](#)

SingleRowCommandBehavior.cs program, [215-217](#)

SIX (Shared with intent exclusive) locking mode, [502](#)

slashes ()

for comments, [97](#)

in URLs, [591](#)

in XPath expressions, [580-581](#)

smalldatetime type in SQL Server, [43, 270](#)

SmallDateTime value in SqlDbType, [241](#)

smallint type in SQL Server, [43, 270](#)

SmallInt value in SqlDbType, [241](#)

smallmoney type, [43, 270](#)

SmallMoney value in SqlDbType, [242](#)

software, obtaining, [4](#)

Software Development Kit (SDK)

for documentation, [23-25, 24-25](#)

installing, [4](#)

Solution Explorer

for Web services, [624, 625](#)

for Windows applications, [163-164, 163](#)

Sort Expression field, [537](#)

Sort property, [454, 460, 463-464, 476](#)

sorting

DataRow objects, [368-373](#)

DataView objects, [460](#)

in retrieving rows, [72-73, 73-74](#)

SOUNDEX function, [114](#)

Source value in SchemaType, [362](#)

sp\_xml\_preparedocument procedure, [597](#)

sp\_xml\_removedocument procedure, [597](#)

SPACE function, [114](#)

spaces in names, [64](#)

special characters in URLs, [591](#)

SQL (Structured Query Language), [3, 59-60](#)

automatic generation of statements in, [398-399](#)

data manipulation language statements in, [62](#)

multiple statement execution in, [282-288](#)

Query Analyzer for. See [Query Analyzer](#)

Transact-SQL. See [Transact-SQL \(T-SQL\)](#)

SQL Server, [29](#)

documentation in, [26, 27](#)

Enterprise Manager for. See [Enterprise Manager](#)  
managed provider classes in, [138](#)  
SqlConnection connections for, [189-190](#)  
    connection pooling with, [192-195](#)  
    connection state with, [195-196](#)  
    events for, [196-200](#)  
    opening and closing, [190-192](#)  
    VS.NET for, [200-204](#), [200-205](#)  
starting and stopping, [30-31](#), [31](#)  
stored procedures in, [35](#), [35](#), [245-255](#)  
for transaction locks. See [locks for transactions types in](#), [42-44](#)  
for updates and deletes, [438](#), [439](#)  
for XML. See [XML](#)

SQL Server Login dialog box, [175](#), [175](#)

sql\_variant type, [44](#), [270](#)

SqlBinary type, [276](#)

SqlBoolean type, [276](#)

SqlByte type, [276](#)

SqlClient namespace, [141](#)

SqlCommand objects, [139](#)  
    closing, [9](#)  
    CreateCommand for, [211](#)  
    creating, [8](#), [146](#), [209-211](#), [255-256](#), [257](#)  
    properties and methods in, [207-209](#)

SqlCommandBuilder class, [140](#), [398](#)

SqlConnection objects, [139](#), [187-189](#)  
    creating, [7-8](#), [145](#)  
    for forms, [166-167](#), [167](#)  
    opening, [8](#)  
    SQL Server database connections with, [189-190](#)  
        connection pooling with, [192-195](#)  
        connection state with, [195-196](#)  
        events for, [196-200](#)  
        opening and closing, [190-192](#)  
        VS.NET for, [200-204](#), [200-205](#)

SqlDataAdapter objects, [140](#), [146](#)  
    creating, [297-298](#)  
    for forms, [166-167](#), [167](#)  
    properties and methods for, [294-297](#)

SqlDataReader objects, [140](#)  
    closing, [9](#)  
    creating, [262-263](#)  
    displaying column values from, [9](#)  
    properties and methods for, [259-262](#)  
    reading rows from, [263-266](#)

SqlDateTime type, [276](#)

SqlDbType enumeration, [241](#)

SqlDecimal type, [276](#)

SqlDouble type, [276](#)

SqlException class, [7](#), [9-10](#)

SqlGuid type, [276](#)

SqlInt16 type, [276](#)

SqlInt32 type, [276](#)

SqlInt64 type, [276](#)

SqlMoney type, [276](#)

SqlParameter class, [140](#)

SqlParameterCollection class, [140](#)

SqlRowUpdatedEventArgs class, [403](#)

SqlRowUpdatingEventArgs class, [401-402](#)

SqlSingle type, [276](#)

SqlString type, [276](#)

SqlTransaction objects, [140](#), [236](#)

    creating, [237](#)

    properties and methods for, [486](#)

    for savepoints, [488-492](#)

    for transaction isolation levels, [496-500](#)

SqlTypes namespace, [262](#)

SQRT function, [112-113](#)

square brackets ([])

    in pattern matching, [65](#)

    in table names, [64](#)

    in XPath expressions, [581](#)

SQUARE function, [112-113](#)

StackTrace property, [10](#)

Start page in VS.NET, [17](#), [18](#)

Start Page tab, [626](#)

Start without Debugging option, [23](#), [157](#), [158](#), [171](#), [182](#)

state information for Web applications

    on client, [552-556](#), [554](#)

    on server, [556-559](#)

state of Connection object, [195-196](#)

State property

    in Connection, [195](#)

    in SqlConnection, [188](#)

StateBag object, [554](#)

StateChange.cs program, [197-198](#)

StateChange event

    coding, [203-204](#), [203-205](#)

in SqlConnection, [189](#)

working with, [196-198](#)

StateChangeHandler method, [196](#)

statements

entering and running, [60-62](#), [61](#)

grouping, [87-89](#), [88](#)

saving and loading, [62](#)

StatementType property, [402](#)

Status column for transaction locks, [503](#)

Status property, [402](#)

STDEV function, [108](#)

STDEVP function, [108](#)

stored procedures, [124-125](#), [386](#)

with browsers, [592-593](#), [592-593](#)

creating, [125-126](#), [126](#), [387-390](#)

DataAdapter objects for, [390-392](#)

for DataRow objects, [393-397](#)

for populating objects, [309-311](#)

result sets with

not returning, [245-252](#)

returning, [252-255](#)

with RETURN, [102](#)

running, [127](#)

in SQL Server, [35](#), [35](#), [245-255](#)

triggers in, [127-128](#)

DeleteProductTrigger, [133-134](#)

InsertProductTrigger, [129-131](#), [130](#)

UpdateUnitPriceProductTrigger, [131-133](#)

Stored Procedures node, [126](#)

StoredProcedure value, [210](#)

storing data

DataSet objects for. See [DataSet objects](#)

locally, [144-151](#)

SqlDataAdapter objects for, [294-298](#)

STR function, [114-115](#)

Strikeout property, [538](#)

string functions, [113-116](#)

string type, [269-270](#)

strongly typed column values, [266-268](#)

DataSet objects for

modifying data, [413-415](#)

reading data, [332-337](#), [333-335](#), [337](#)

Get\* methods for, [268-275](#), [272](#)

GetSql\* methods for, [276-281](#)

StronglyTypedColumnValues.cs program, [273-275](#)

StronglyTypedColumnValueSql.cs program, [279-281](#)

StronglyTypedDataSet directory, [332](#)

StronglyTypedDataSet2 directory, [414](#)

Structured Query Language (SQL), [3](#), [59-60](#)

    automatic generation of statements in, [398-399](#)

    data manipulation language statements in, [62](#)

    multiple statement execution in, [282-288](#)

    Query Analyzer for. See [Query Analyzer](#)

    Transact-SQL. See [Transact-SQL \(T-SQL\)](#)

STUFF function, [114](#), [116](#)

stylesheets, [594-596](#), [596](#)

subqueries, [69](#)

SUBSTRING function, [114](#), [116](#)

SUM function, [108](#), [110](#)

Support Services folder, [32](#)

System namespace, [7](#)

System.Data namespace, [143](#)

System.Data.DataSet class, [138](#)

System.Data.Odbc namespace, [141](#)

System.Data.OleDb namespace, [12-13](#), [141](#)

System.Data.SqlClient namespace, [141](#)

System.Data.SqlTypes namespace, [262](#)

System.EventArgs class, [400](#)

system functions, [117-119](#)

system statistical functions, [107](#)

System.Windows.Forms namespace, [161](#)

**Team LiB**

**◀ PREVIOUS**

**NEXT ▶**

## Index

### T

T-SQL. See [Transact-SQL \(T-SQL\)](#)

Table controls, [526](#)

Table lock type, [501](#)

Table property

  in DataColumn, [350](#)

  in DataRow, [348](#)

  in DataView, [454](#)

  in ForeignKeyConstraint, [421](#)

  in UniqueConstraint, [418](#)

table type, [44](#)

TableDirect commands, [207](#), [210-211](#), **[223-225](#)**

TableDirect value, [210](#)

TableMapping object, [329](#)

TableMapping property, [402](#)

TableMappingCollection class, [329](#)

TableMappings property, [296](#), [329](#)

TableName property, [312](#), [330](#), [346](#)

tables, [3](#), [29](#)

  aliases for, **[78-79](#)**, [78](#)

  altering, [90](#)

  creating, **[89-90](#)**

  dropping, [90](#)

  Enterprise Manager for, **[51-52](#)**, [51-52](#)

  integrity of, **[85-86](#)**

  joined

    for populating objects, **[303-304](#)**

    retrieving rows from, **[79-81](#)**, [80](#)

    working with, **[283-285](#)**

  mapping, **[328-331](#)**, [329](#)

  names for, [52](#), [52](#), [64](#)

  relationships in. See [relationships](#)

  rows in. See [rows](#)

  in SQL Server, [33](#), [34](#)

Tables property, [300](#)

templates

  in DataList, [545](#)

  XML, **[593-594](#)**, [594](#)

Terms Of Use page, [629](#), [629](#)

Test Connection option, [92](#)

testing Web services, **[621-622](#)**, [622-623](#)

Text command type, [210](#)

text functions, [107](#)

Text property

- in DataBindings, [179](#)
- for DropDownList controls, [529](#)
- for Label controls, [528](#)
- for RadioButtonList controls, [529](#)

text type, [43](#), [270](#)

Text value

- in CommandType, [210](#)
- in SqlDbType, [242](#)

TextBox controls, [165](#)

- in Web forms, [524](#)
- in Windows forms, [180](#)

time functions, [\*\*116-117\*\*](#)

TIME keyword, [103](#)

timeouts

- for database connections, [190](#)
- for transaction locks, [505](#)

timestamp type in SQL Server, [44](#), [270](#)

Timestamp value in SqlDbType, [242](#)

tinyint type in SQL Server, [43](#), [270](#)

TinyInt value in SqlDbType, [242](#)

Toolbox, [\*\*155-156\*\*](#)

Tools menu, [21](#)

TOP keyword, [74](#), [75](#)

ToString method

- in DataView, [455](#)
- for ListView controls, [289](#)

@@TRANCOUNT function, [118](#)

Transact-SQL (T-SQL)

- CASE statements in, [\*\*98-99\*\*](#)
- comments in, [97](#)
- conditional logic in, [\*\*97-98\*\*](#)
- cursors in, [\*\*103-107\*\*](#)
- functions in, [107](#)
  - aggregate, [\*\*107-110\*\*](#), [109-110](#)
  - date and time, [\*\*116-117\*\*](#)
  - mathematical, [\*\*111-113\*\*](#)
  - string, [\*\*113-116\*\*](#)
  - system, [\*\*117-119\*\*](#)
    - user-defined, [\*\*119-124\*\*](#), [121](#), [123-124](#)
- GOTO statements in, [\*\*101-102\*\*](#)
- RAISERROR statements in, [103](#)
- RETURN statements in, [102](#)
- for savepoints, [\*\*486-488\*\*](#), [489](#)
- stored procedures in, [\*\*124-127\*\*](#), [126](#)

for transaction isolation levels, [493-495](#), [495](#)

triggers in, [127-128](#)

  DeleteProductTrigger, [133-134](#)

  InsertProductTrigger, [129-131](#), [130](#)

  UpdateUnitPriceProductTrigger, [131-133](#)

variables in, [95-96](#), [96](#)

WAITFOR statements in, [102-103](#)

WHILE statements in, [99-101](#)

Transaction class, [140](#), [236](#)

Transaction property

  in DataAdapter, [413](#)

  in SqlCommand, [208](#)

TransactionIsolation.cs program, [497-500](#)

TransactionIsolation.sql script, [494-495](#), [495](#)

transactions, [87-89](#), [236-239](#), [485](#)

  with DataSet objects, [412-413](#)

  isolation levels for, [492-500](#), [495](#)

  locks for. See [locks for transactions](#)

  savepoints for

    SqlTransaction objects for, [488-492](#)

    T-SQL for, [486-488](#), [489](#)

  SqlTransaction objects for, [140](#), [236](#)

    creating, [237](#)

    for isolation levels, [496-500](#)

    properties and methods in, [486](#)

    for savepoints, [488-492](#)

triggers, [127-128](#)

  DeleteProductTrigger, [133-134](#)

  InsertProductTrigger, [129-131](#), [130](#)

  UpdateUnitPriceProductTrigger, [131-133](#)

try/catch blocks, [7](#), [9-10](#)

types

  fixed length, [51](#)

  in SQL Server, [42-44](#)

  in SqlDbType, [241-242](#)

  for T-SQL variables, [95](#)

## Index

### U

U (Update) locking mode, [501](#)

UDDI Business Registry Node page, [628-629](#), [628-630](#)

UDDI (Universal Description, Discovery, and Integration) service, [626](#)

UDDI Test Environment option, [627](#), [630](#)

UDDI Web Service Registration page, [627](#), [627](#)

uid element in connection strings, [7](#)

uint type, [269](#)

UInt16 type, [269](#)

UInt32 type, [269](#)

UInt64 type, [269](#)

ulong type, [269](#)

Unchanged value

  in DataRowState, [379](#)

  in DataViewRowState, [369](#), [456](#)

Underline property, [538](#)

UNICODE function, [114](#), [116](#)

UNION cause, [76](#), [76](#)

UNION ALL clause, [573](#)

Unique property, [350-351](#), [353-354](#)

UniqueConstraint objects

  creating, [418-420](#)

  properties for, [418](#)

  purpose of, [143](#)

uniqueidentifier column, [53](#)

uniqueidentifier type, [44](#), [270](#)

Uniquedentifier value, [242](#)

Universal Description, Discovery, and Integration (UDDI) service, [626](#)

Unspecified value, [493](#)

Update (U) locking mode, [501](#)

Update method

  in DataAdapter, [377-379](#)

  in SqlDataAdapter, [296](#)

UPDATE permission, [38](#)

UPDATE statement, [62](#), [83-84](#), [84](#)

  with browsers, [592](#), [592](#)

  with ExecuteNonQuery, [230-233](#)

  integrity with, [86](#)

  multiple, [286-288](#)

UpdateCommand property  
in DataAdapter, [374-376](#), [391-392](#), [398](#)

setting, [381-382](#)

in SqlDataAdapter, [296](#), [339](#)

UpdateCustomerT1 method, [511-512](#)

UpdateCustomerT2 method, [511](#), [513](#)

UpdateProduct.sql stored procedure, [388](#)

UpdateProductT1 method, [511-513](#)

UpdateProductT2 method, [511-512](#)

UpdateRowSource property, [208](#)

UpdateRule property, [421](#), [428](#), [439-440](#)

updates

DataRow objects, [434](#)

failures in, [408-412](#)

of primary keys, [437-443](#), [439](#)

SQL Server for, [438](#), [439](#)

UpdateRule property for, [439-440](#)

UpdateUnitPriceProductTrigger.sql script, [131-133](#)

UPPER function, [114-115](#)

URLs

INSERT, UPDATE, and DELETE statements in, [592](#)

SELECT statements in, [590-591](#)

special characters in, [591](#)

stored procedures in, [592-593](#)

USE statement, [61](#)

user-defined data types, [38](#)

user-defined functions

inline table-valued, [121-122](#)

multistatement table-valued, [122-124](#), [123-124](#)

scalar, [119-121](#), [121](#)

in SQL Server, [38](#)

in T-SQL, [119](#)

usernames, [15](#), [190](#)

users in SQL Server, [36](#), [36](#)

ushort type, [269](#)

using statement, [7](#)

UsingColumnOrdinals.cs program, [264-266](#)

UsingDataView.cs program, [458-460](#)

UsingDataView2.cs program, [461](#)

Using DataViewManager.cs program, [476-478](#)

UsingDefaultSort.cs program, [460](#)

UsingParameters.cs program, [244-245](#)

UsingXmlDataDocument.cs program, [608-612](#)

Using XmlDocument.cs program, [604-606](#)

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

# Index

## V

ValidationSummary controls, [526](#)

Value property, [243](#)

for DropDownList controls, [529](#)

for RadioButtonList controls, [529](#)

in XmlDocument, [601](#)

values of parameters, [243](#)

Values property, [401](#)

VAR function, [108](#)

varbinary type in SQL Server, [43](#), [270](#)

VarBinary value in SqlDbType, [242](#)

varchar type in SQL Server, [43](#), [270](#)

VarChar value in SqlDbType, [242](#)

variables in T-SQL, [95-96](#), [96](#)

Variant value, [242](#)

VARP function, [108](#)

View menu, [20](#)

View Schema link, [450](#)

View Wizard Results screen, [340](#), [340](#)

viewing

hidden code, [162](#)

properties, [92](#), [94](#)

transaction lock information, [502-503](#), [503](#)

views

retrieving rows from, [81-82](#)

in SQL Server, [34-35](#), [35](#)

ViewState property

in Page, [552](#)

for Web state information, [554-556](#)

ViewStateTest.aspx file, [555-556](#)

virtual directories, [587](#)

Virtual Name Configuration dialog box, [589](#), [590](#)

Virtual Names tab, [589](#)

Visual Studio .NET (VS.NET), [3](#), [17](#)

for ASP.NET applications, [517-524](#), [518-520](#)

for Command objects, [255-257](#), [256-257](#)

compiling and running programs using, [23](#), [23](#)

for connection objects, [200-204](#), [200-205](#)

DataAdapter objects in, [337-341](#), [338-341](#)

for database access, [91-92](#), [91-94](#)

DataReader objects in, [288-291](#), [289-291](#)

DataSet objects in, [342-343](#), [342-344](#)  
DataView objects in, [478-480](#), [479-480](#)  
documentation in, [23-26](#), [24-26](#)  
events in, [203-204](#), [203-205](#)  
modifying code generated by, [21-22](#)  
projects in, [19](#), [19](#)  
for relationships, [448-452](#), [448-452](#)  
starting, [17-19](#), [18](#)  
working in, [20-21](#), [20](#)

Team LiB

◀ PREVIOUS ▶ NEXT ▶

## Index

### W

WAITFOR statement, [102-103](#)

Web applications

    ASP.NET, [517-524](#), [518-520](#)

    creating, [532-535](#), [532](#), [534](#)

    DataGrid controls for. See [DataGrid controls](#)

    DataList controls for, [543-551](#), [544-548](#), [550](#)

    forms for

        building, [527-532](#), [527-528](#)

        controls for, [524-526](#)

    shopping cart, [560](#), [560](#)

        Buy button in, [560-562](#), [561-562](#)

        code for, [563-565](#)

        shopping cart in, [563](#), [563](#)

    state information in

        on client, [552-556](#), [554](#)

        on server, [556-559](#)

Web Forms category in Toolbox, [156](#)

Web Hosting link, [19](#)

Web services, [613](#)

    creating, [613-617](#), [614](#)

    registering, [626-630](#), [627-631](#)

    testing, [621-622](#), [622-623](#)

    using, [624-626](#), [624-626](#)

    VS.NET for, [17](#)

    WSDL files for, [617-621](#), [618](#)

Web Services Description Language (WSDL) files, [617-621](#), [618](#)

WebForm1.aspx file, [520-522](#), [530-531](#), [541-543](#), [550-551](#)

WebForm1.aspx.cs file, [522-524](#), [532](#), [563-565](#)

WebService class, [616](#)

What's New link, [18](#)

WHERE clause, [64-65](#), [64](#)

    BETWEEN operator, [69](#)

    with DELETE, [84](#)

    with functions, [109](#)

    IN operator, [68-69](#), [69](#)

    with INSERT, [83](#)

    IS NULL operator, [70](#), [70](#)

    JOIN operator, [79-81](#), [80](#)

    LIKE operator, [65-68](#), [66-68](#)

    logical operators in, [71-72](#), [71-72](#)

    NOT operator, [70](#)

    UNION operator, [76](#), [76](#)

WHILE statement, [99-101](#)

wildcard characters in pattern matching, [65-66](#)

Window menu, [21](#)

Windows applications, [153-154](#)

  Class View in, [164](#), [164](#)

  code behind forms in, [158-163](#), [162](#)

  controls in, [164-165](#)

    adding, [180](#), [181](#)

    data binding in, [179-180](#)

    DataGrid, [165-172](#), [166-171](#)

    labels and button, [156-158](#), [157-158](#)

  creating, [154-155](#), [154-155](#), [448-450](#), [448-450](#)

  forms for

    creating, [172-179](#), [172-179](#)

    running, [182-183](#), [182](#)

  main method in, [181](#)

  passwords in, [181](#), [182](#)

  Properties window for, [156](#)

  Solution Explorer in, [163-164](#), [163](#)

  Toolbox for, [155-156](#)

  VS.NET for, [17](#)

Windows Forms category in Toolbox, [156](#)

WindowsApplication4.csproj file, [342](#), [414](#)

WorkstationId property, [188](#)

WriteAndReadXml.cs program, [326-328](#)

WriteContentTo method, [603](#)

WriteLine method, [7](#)

WriteSchema value, [323](#)

WriteTo method, [603](#)

WriteXml method, [301](#), [322-324](#), [443](#)

WriteXmlSchema, [301](#), [321](#), [324-325](#)

writing XML, [322-325](#)

WSDL (Web Services Description Language) files, [617-621](#), [618](#)

## Index

### X

X (Exclusive) locking mode, [501](#)

XML

for browsers, [590-593](#), [591-593](#)

DataSet objects for, [321-328](#)

document structure in, [578-579](#), [579-580](#)

FOR XML clause for. See [FOR XML clause](#)

for HTTP, [586-596](#), [586-594](#), [596](#)

for IIS, [586-590](#), [586-590](#)

nested, [443-447](#)

OPENXML function, [596-599](#), [599](#)

retrieving data as, [227-229](#)

schema relationships in, [450-452](#), [451-452](#)

stylesheets in, [594-596](#), [596](#)

templates in, [593-594](#), [594](#)

Web services, [17](#)

XmlDataDocument objects, [606-612](#)

XmlDocument objects, [600-606](#)

and XPath, [578-582](#), [579-580](#)

and XSLT, [582-586](#), [585](#)

XML controls, [526](#)

xml directive, [571](#)

XML Schema category in Toolbox, [155](#)

XML tab, [332](#)

XML Web Services link, [18](#), [626](#)

XMLDATA option, [227](#), [574-575](#)

XmlDataDocument objects, [606-612](#)

XmlDocument objects, [600-660](#)

XmlReader class, [228](#)

XmlReadMode enumeration, [326](#)

XmlResolver property, [601](#)

xmltext directive, [572](#)

XmlWriteMode enumeration, [323](#)

XPath (Extensible Markup Language Path)

expressions in, [580-582](#)

and XML document structure, [578-579](#), [579-580](#)

xsl tags, [583](#)

XSLT (Extensible Stylesheet Language Transformation), [582-586](#), [585](#)

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

## Index

### Y

YEAR function, [116-117](#)

**Team LiB**

[◀ PREVIOUS](#) [NEXT ▶](#)

## List of Figures

### **Chapter 1: Introduction to Database Programming with ADO.NET**

[Figure 1.1:](#) The Start page

[Figure 1.2:](#) The New Project dialog box with the appropriate settings for a C# console application

[Figure 1.3:](#) The VS .NET environment

[Figure 1.4:](#) The running program

[Figure 1.5:](#) The documentation home page

[Figure 1.6:](#) Searching the index for the word console

[Figure 1.7:](#) Searching all of the documentation for the word WriteLine

[Figure 1.8:](#) The documentation contents viewed in VS .NET

[Figure 1.9:](#) SQL Server documentation home page

[Figure 1.10:](#) SELECT examples documentation

### **Chapter 2: Introduction to Databases**

[Figure 2.1:](#) The Service Manager

[Figure 2.2:](#) The Enterprise Manager

[Figure 2.3:](#) The Customers, Orders, Order Details, and Products tables

[Figure 2.4:](#) The tables of the Northwind database

[Figure 2.5:](#) The Customers table properties

[Figure 2.6:](#) The alphabetical list of products view properties

[Figure 2.7:](#) The CustOrdHist stored procedure properties

[Figure 2.8:](#) The dbo user properties

[Figure 2.9:](#) The public role properties

[Figure 2.10:](#) The public role permissions

[Figure 2.11:](#) Relationships between the Customers, Orders, Order Details, and Products tables

[Figure 2.12:](#) Relationship between the Customers and Orders table

[Figure 2.13:](#) Indexes for the Customers table

[Figure 2.14:](#) Rows from the Customers table

[Figure 2.15:](#) Rows from the Orders table

[Figure 2.16:](#) Restricted rows from the Order Details table

[Figure 2.17:](#) Restricted rows from the Products table

[Figure 2.18:](#) The query builder

[Figure 2.19:](#) Building and running a query

[Figure 2.20:](#) Adding a new table

[Figure 2.21:](#) Entering the name of the table

[Figure 2.22:](#) Setting the permissions

[Figure 2.23:](#) Creating the relationship

[Figure 2.24:](#) Creating an index

[Figure 2.25:](#) Creating a constraint

## **Chapter 3: Introduction to Structured Query Language (SQL)**

[Figure 3.1:](#) Connecting to a SQL Server database

[Figure 3.2:](#) Viewing database items using the Object Browser and executing a SELECT statement using the Query window

[Figure 3.3:](#) Using a SELECT statement to retrieve rows from the Customers table

[Figure 3.4:](#) Using a WHERE clause to restrict rows from the Customers table to those where Country is equal to 'UK'

[Figure 3.5:](#) Products where ProductName is like 'Cha\_'

[Figure 3.6:](#) Products where ProductName is like 'Cha%'

[Figure 3.7:](#) Products where ProductName is like '[ ABC ] %'

[Figure 3.8:](#) Products where ProductName is like '[ ^ABC ] %'

[Figure 3.9:](#) Products where ProductName is like '[ A-E ] %'

[Figure 3.10:](#) Using the IS NULL operator to retrieve customers where Fax contains a null value

[Figure 3.11:](#) Using the AND operator to retrieve products where UnitsInStock is less than 10 and ReorderLevel is less than or equal to 20

[Figure 3.12:](#) Using the OR operator to retrieve products where either UnitsInStock is less than 10 or ReorderLevel is less than or equal to 20

[Figure 3.13:](#) Using the ORDER BY clause to order products by ascending ProductName

[Figure 3.14:](#) Using the DESC and ASC keywords to order products by descending UnitsInStock and ascending ReorderLevel

[Figure 3.15:](#) Using the TOP keyword to retrieve the top 10 products by ProductID

[Figure 3.16:](#) Using the DISTINCT keyword to retrieve distinct Country column values

[Figure 3.17:](#) Using the UNION operator to combine retrieved rows from two SELECT statements

[Figure 3.18:](#) Using the GROUP BY clause to divide rows into blocks

[Figure 3.19:](#) Using the HAVING clause to restrict retrieved groups of rows

[Figure 3.20:](#) Using the AS clause to specify the display name for columns

[Figure 3.21:](#) Using a multitable SELECT statement to retrieve orders placed by a specific customer

[Figure 3.22:](#) Using an INSERT statement to add a new row to the Customers table

[Figure 3.23:](#) Using an UPDATE statement to modify the Address column of a row in the Customers table

[Figure 3.24:](#) Using an UPDATE statement to remove a row from the Customers table

[Figure 3.25:](#) Using a transaction

[Figure 3.26:](#) Entering database details using the Data Link Properties dialog box

[Figure 3.27:](#) Viewing the rows in the Customers table using the Server Explorer

[Figure 3.28:](#) Entering a SQL statement

[Figure 3.29:](#) Building a SQL statement visually

[Figure 3.30:](#) Properties of the CustomerID columns

## **Chapter 4: Introduction to Transact-SQL Programming**

[Figure 4.1:](#) Executing T-SQL using Query Analyzer

[Figure 4.2:](#) Using the AVG( ) function to compute the average value of the UnitPrice column

[Figure 4.3:](#) Using the SUM( ) function to compute the total of the UnitPrice column

[Figure 4.4:](#) Using Enterprise Manager to define a function

[Figure 4.5:](#) Using an inline table-valued function

[Figure 4.6:](#) Using a multistatement table-valued function

[Figure 4.7:](#) Using Enterprise Manager to define a procedure

[Figure 4.8:](#) Using Enterprise Manager to view a trigger

## **Chapter 5: Overview of the ADO.NET Classes**

[Figure 5.1:](#) Some of the managed provider objects

[Figure 5.2:](#) Some of the generic data set objects

## **Chapter 6: Introducing Windows Applications and ADO.NET**

[Figure 6.1:](#) Creating a C# Windows application in Visual Studio .NET

[Figure 6.2: A blank form](#)

[Figure 6.3: The form with a label and button control](#)

[Figure 6.4: The running form](#)

[Figure 6.5: Hiding code in VS .NET using the #region directive](#)

[Figure 6.6: Viewing hidden code in VS .NET](#)

[Figure 6.7: The Solution Explorer](#)

[Figure 6.8: The Class View](#)

[Figure 6.9: Form with a DataGridView](#)

[Figure 6.10: Form with SqlConnection and SqlDataAdapter objects](#)

[Figure 6.11: Setting the ConnectionString property for the sqlConnection1 object](#)

[Figure 6.12: SelectCommand property for the SqlDataAdapter1 object](#)

[Figure 6.13: The Query Builder](#)

[Figure 6.14: Previewing the rows retrieved by the SELECT statement](#)

[Figure 6.15: Entering the DataSet details in the Generate Dataset dialog box](#)

[Figure 6.16: The running form](#)

[Figure 6.17: The running form](#)

[Figure 6.18: Adding a data form using the Data Form Wizard](#)

[Figure 6.19: Entering the name of the new DataSet](#)

[Figure 6.20: Choosing the data connection](#)

[Figure 6.21: Logging in to the SQL Server Northwind database](#)

[Figure 6.22: Selecting the Customers and Orders tables for use in the form](#)

[Figure 6.23: Creating a relationship between two tables](#)

[Figure 6.24: Selecting the columns to display from each table](#)

[Figure 6.25: Choosing the display style](#)

[Figure 6.26: The completed form](#)

[Figure 6.27: The editCustomerID text box is bound to the CustomerID column](#)

[Figure 6.28: Binding the City column to the editCity text box](#)

[Figure 6.29: Setting the ConnectionString property](#)

[Figure 6.30:](#) The running form

## **Chapter 7: Connecting to a Database**

[Figure 7.1:](#) Creating a `SqlConnection` object with Visual Studio .NET

[Figure 7.2:](#) Selecting the provider

[Figure 7.3:](#) Entering the connection details

[Figure 7.4:](#) Entering the advanced connection details

[Figure 7.5:](#) Viewing all the connection details

[Figure 7.6:](#) `sqlConnection1` object's events

[Figure 7.7:](#) The beginning `StateChange` event handler method

[Figure 7.8:](#) The completed `StateChange` event handler method

## **Chapter 8: Executing Database Commands**

[Figure 8.1:](#) A `SqlCommand` object in a form

[Figure 8.2:](#) Adding the `Customers` table to the query using the Add Table dialog

[Figure 8.3:](#) Adding the `CustomerID`, `CompanyName`, and `ContactName` columns to the query using Query Builder

## **Chapter 9: Using *DataReader* Objects to Read Results**

[Figure 9.1:](#) Obtaining the type of a column using Visual Studio .NET's Server Explorer

[Figure 9.2:](#) Adding a `ListView` control to the form

[Figure 9.3:](#) The completed `Form1_Load()` method

[Figure 9.4:](#) The running form

## **Chapter 10: Using *Dataset* Objects to Store Data**

[Figure 10.1:](#) Some of the `DataSet` objects

[Figure 10.2:](#) Using the `AS` keyword

[Figure 10.3:](#) Adding a new Data Set

[Figure 10.4:](#) `MyDataSet.xsd`

[Figure 10.5:](#) `Customers` table added to form

[Figure 10.6:](#) Viewing all the files using the Solution Explorer window

[Figure 10.7:](#) The running form

[Figure 10.8:](#) The Data Adapter Configuration Wizard

[Figure 10.9:](#) Choosing your data connection

[Figure 10.10:](#) Choosing your query type

[Figure 10.11:](#) Generating the SQL statements

[Figure 10.12:](#) Final dialog box for the Data Adapter Configuration Wizard

[Figure 10.13:](#) The new `SqlDataAdapter` object in the tray

[Figure 10.14:](#) The Generate Dataset dialog box

[Figure 10.15:](#) The new `DataSet` object in the tray

[Figure 10.16:](#) The running form

## **Chapter 12: Navigating and Modifying Related Data**

[Figure 12.1:](#) The Relationships tab for `FK_Orders_Customers`

[Figure 12.2:](#) Creating the Windows application

[Figure 12.3:](#) Selecting both the `Customers` and `Orders` tables from Server Explorer

[Figure 12.4:](#) The new objects in the tray beneath the form

[Figure 12.5:](#) The Generate Dataset dialog box

[Figure 12.6:](#) The Schema Editor

[Figure 12.7:](#) The Edit Relation dialog box

[Figure 12.8:](#) The properties of the new relation

## **Chapter 13: Using `DataView` Objects**

[Figure 13.1:](#) Setting the Properties of `dataGridView1`

[Figure 13.2:](#) Setting the Properties of `dataGridView1`

[Figure 13.3:](#) The running form

## **Chapter 14: Advanced Transaction Control**

[Figure 14.1:](#) Running the `Savepoint.sql` script in Query Analyzer

[Figure 14.2:](#) Running the `TransactionIsolation.sql` script in Query Analyzer

[Figure 14.3:](#) Viewing the locks using the Locks/ Process ID node of Enterprise Manager

[Figure 14.4:](#) The transaction on the top part is blocking the transaction on the bottom.

[Figure 14.5:](#) Once the top transaction is committed, the bottom `UPDATE` proceeds.

## **Chapter 15: Introducing Web Applications-ASP.NET**

Figure 15.1: Creating an ASP.NET Web application in Visual Studio .NET

Figure 15.2: Adding TextBox and Button controls to the form

Figure 15.3: The running form

Figure 15.4: The appearance of the final form

Figure 15.5: The ListItem Collection Editor

Figure 15.6: Message from the NameRequired-FieldValidator control

Figure 15.7: Form with a DataGrid

Figure 15.8: The running form

Figure 15.9: The General properties

Figure 15.10: The Columns properties

Figure 15.11: The Paging properties

Figure 15.12: The Format properties

Figure 15.13: The Borders properties

Figure 15.14: Displaying the DataGrid events

Figure 15.15: The running form

Figure 15.16: Form with a DataList

Figure 15.17: The modified header and footer templates with Label controls

Figure 15.18: The Item Templates editor

Figure 15.19: Setting the properties of the table

Figure 15.20: Adding the Label

Figure 15.21: Binding the Label to the ProductID column

Figure 15.22: The running form

Figure 15.23: The running CookieTest.aspx page

Figure 15.24: The running form

Figure 15.25: DataGrid1 properties

Figure 15.26: Buy button properties

Figure 15.27: DataGrid1 with Buy button

Figure 15.28: ShoppingCart DataGrid

## **Chapter 16: Using SQL Server's XML Support**

[Figure 16.1:](#) Running a `SELECT` statement containing a `FOR XML RAW` clause in Query Analyzer

[Figure 16.2:](#) Retrieving rows from the `Employees` table

[Figure 16.3:](#) `Customers.xml` document structure

[Figure 16.4:](#) Viewing `Customers.xml` in Internet Explorer

[Figure 16.5:](#) Viewing `CustomersUsing-Stylesheet.xml` in Internet Explorer

[Figure 16.6:](#) The IIS Virtual Directory Management for SQL Server console

[Figure 16.7:](#) Setting the Virtual Directory Name and Local Path

[Figure 16.8:](#) Setting the authentication details

[Figure 16.9:](#) Setting the data source

[Figure 16.10:](#) Setting the type of access

[Figure 16.11:](#) Setting the virtual name configuration

[Figure 16.12:](#) Selecting customers and displaying results

[Figure 16.13:](#) Adding a new row to the `Customers` table

[Figure 16.14:](#) Running a stored procedure

[Figure 16.15:](#) Running the `Customers.xml` file

[Figure 16.16:](#) Running the `CustomersUsing-Stylesheet.xml` file

[Figure 16.17:](#) Running the `RunAddCustomers.sql` script

## **Chapter 17: Web Services**

[Figure 17.1:](#) Creating a Web service in VS .NET

[Figure 17.2:](#) Adding a new Web service

[Figure 17.3:](#) Accessing a Web service

[Figure 17.4:](#) The Web service test page

[Figure 17.5:](#) Running the `RetrieveCustomers()` method with a `whereClause` of `CustomerID= 'ALFKI'`

[Figure 17.6:](#) Running the `RetrieveCustomers()` method with a `whereClause` of `CustomerID IS NOT NULL`

[Figure 17.7:](#) Form with controls

[Figure 17.8:](#) Northwind Web Service

[Figure 17.9:](#) The new Web reference in Solution Explorer

[Figure 17.10:](#) The running form

[Figure 17.11:](#) The XML Web Services page

[Figure 17.12:](#) The UDDI Web Service Registration page

[Figure 17.13:](#) The UDDI Business Registry Node page

[Figure 17.14:](#) Logging in using a Microsoft Passport account

[Figure 17.15:](#) Entering your email address, name, and phone number

[Figure 17.16:](#) The terms of use page

[Figure 17.17:](#) Setting the business name and description

[Figure 17.18:](#) Setting the Web service details

**Team LiB**

◀ PREVIOUS ▶ NEXT ▶

## List of Tables

### Chapter 2: Introduction to Databases

[Table 2.1:](#) SOME ROWS FROM THE PRODUCTS TABLE

[Table 2.2:](#) MEANING OF THE AVAILABLE PERMISSIONS

[Table 2.3:](#) SQL SERVER DATABASE TYPES

[Table 2.4:](#) DEFINITION FOR THE COLUMNS OF THE Customers TABLE

[Table 2.5:](#) DEFINITION FOR THE COLUMNS OF THE Orders TABLE

[Table 2.6:](#) DEFINITION FOR THE COLUMNS OF THE Order Details TABLE

[Table 2.7:](#) DEFINITION FOR THE COLUMNS OF THE Products TABLE

[Table 2.8:](#) DEFINITION FOR THE COLUMNS OF THE Persons TABLE

### Chapter 3: Introduction to Structured Query Language (SQL)

[Table 3.1:](#) SQL MATHEMATICAL OPERATORS

[Table 3.2:](#) WILDCARD CHARACTERS

[Table 3.3:](#) LOGICAL OPERATORS

### Chapter 4: Introduction to Transact-SQL Programming

[Table 4.1:](#) RETURN VALUES FROM THE @@FETCH\_STATUS FUNCTION

[Table 4.2:](#) FUNCTIONS

[Table 4.3:](#) AGGREGATE FUNCTIONS

[Table 4.4:](#) MATHEMATICAL FUNCTIONS

[Table 4.5:](#) STRING FUNCTIONS

[Table 4.6:](#) DATE AND TIME FUNCTIONS

[Table 4.7:](#) SYSTEM FUNCTIONS

[Table 4.8:](#) ROW ADDED TO THE ProductAudit TABLE BY InsertProductTrigger

[Table 4.9:](#) ROW ADDED TO THE ProductAudit TABLE BY UpdateUnitPriceProductTrigger

[Table 4.10:](#) ROW ADDED TO THE ProductAudit TABLE BY DeleteProductTrigger

### Chapter 6: Introducing Windows Applications and ADO.NET

[Table 6.1:](#) ACCESS MODIFIERS

[Table 6.2: COMMONLY USED WINDOWS FORM CONTROLS](#)

## **Chapter 7: Connecting to a Database**

[Table 7.1: SqlConnection PROPERTIES](#)

[Table 7.2: SqlConnection METHODS](#)

[Table 7.3: SqlConnection EVENTS](#)

[Table 7.4: ConnectionState CONSTANTS](#)

## **Chapter 8: Executing Database Commands**

[Table 8.1: SqlCommand PROPERTIES](#)

[Table 8.2: SqlCommand METHODS](#)

[Table 8.3: CommandType ENUMERATION VALUES](#)

[Table 8.4: METHODS THAT RETRIEVE INFORMATION FROM THE DATABASE](#)

[Table 8.5: CommandBehavior ENUMERATION VALUES](#)

[Table 8.6: SCHEMA COLUMN VALUES](#)

[Table 8.7: FOR XML KEYWORDS](#)

[Table 8.8: THE ExecuteNonQuery\( \) METHOD](#)

[Table 8.9: SqlDbType ENUMERATION MEMBERS](#)

## **Chapter 9: Using *DataReader* Objects to Read Results**

[Table 9.1: SqlDataReader PROPERTIES](#)

[Table 9.2: SqlDataReader METHODS](#)

[Table 9.3: STANDARD C# AND .NET TYPES](#)

[Table 9.4: SQL SERVER TYPES, COMPATIBLE STANDARD C# TYPES, AND GET\\* METHODS](#)

[Table 9.5: Products TABLE COLUMNS, TYPES, AND METHODS](#)

[Table 9.6: Sql\\* TYPES](#)

[Table 9.7: SQL SERVER TYPES, COMPATIBLE Sql\\* TYPES, AND GetSql\\* METHODS](#)

[Table 9.8: Products TABLE COLUMNS, TYPES, AND GetSql\\* METHODS](#)

## **Chapter 10: Using *Dataset* Objects to Store Data**

[Table 10.1: SqlDataAdapter PROPERTIES](#)

[Table 10.2: SqlDataAdapter METHODS](#)

[Table 10.3: SqlDataAdapter EVENTS](#)

[Table 10.4: DataSet PROPERTIES](#)

[Table 10.5: DataSet METHODS](#)

[Table 10.6: DataSet EVENT](#)

[Table 10.7: MissingSchemaAction ENUMERATION MEMBERS](#)

[Table 10.8: XmlWriteMode ENUMERATION MEMBERS](#)

[Table 10.9: XmlReadMode ENUMERATION MEMBERS](#)

## **Chapter 11: Using DataSet Objects to Modify Data**

[Table 11.1: DataTable PROPERTIES](#)

[Table 11.2: DataTable METHODS](#)

[Table 11.3: DataTable EVENTS](#)

[Table 11.4: DataRow PROPERTIES](#)

[Table 11.5: DataRow METHODS](#)

[Table 11.6: DataColumn PROPERTIES](#)

[Table 11.7: SchemaType ENUMERATION MEMBERS](#)

[Table 11.8: DataViewRowState ENUMERATION MEMBERS](#)

[Table 11.9: DataRowVersion ENUMERATION MEMBERS](#)

[Table 11.10: DataRowState ENUMERATION MEMBERS](#)

[Table 11.11: SqlDataAdapter EVENTS](#)

[Table 11.12: FillEventArgs PROPERTIES](#)

[Table 11.13: SqlRowUpdatingEventArgs PROPERTIES](#)

[Table 11.14: ADDITIONAL SqlRowUpdatedEventArgs PROPERTY](#)

[Table 11.15: DataTable EVENTS](#)

[Table 11.16: DataColumnChangeEventArgs PROPERTIES](#)

[Table 11.17: DataRowChangeEventArgs PROPERTIES](#)

## **Chapter 12: Navigating and Modifying Related Data**

[Table 12.1: UniqueConstraint PROPERTIES](#)

[Table 12.2: ForeignKeyConstraint PROPERTIES](#)

[Table 12.3: DataRelation PROPERTIES](#)

[Table 12.4: Rule ENUMERATION MEMBERS](#)

## **Chapter 13: Using *DataView* Objects**

[Table 13.1: DataView PROPERTIES](#)

[Table 13.2: DataView METHODS](#)

[Table 13.3: DataViewRowState ENUMERATION MEMBERS](#)

[Table 13.4: DataRowView PROPERTIES](#)

[Table 13.5: DataRowView METHODS](#)

[Table 13.6: DataViewManager PROPERTIES](#)

## **Chapter 14: Advanced Transaction Control**

[Table 14.1: SqlTransaction PROPERTIES](#)

[Table 14.2: SqlTransaction METHODS](#)

[Table 14.3: SQL Standard Isolation Levels](#)

[Table 14.4: IsolationLevel Enumeration Members](#)

[Table 14.5: SQL Server Lock Types](#)

[Table 14.6: SQL Server Locking Modes](#)

## **Chapter 15: Introducing Web Applications-ASP.NET**

[Table 15.1: WEB FORM CONTROLS](#)

[Table 15.2: Label CONTROLS](#)

[Table 15.3: RadioButtonList ITEMS](#)

[Table 15.4: DropDownList ITEMS](#)

## **Chapter 16: Using SQL Server's XML Support**

[Table 16.1: FOR XML KEYWORDS](#)

[Table 16.2: DIRECTIVES](#)

[Table 16.3: SPECIFYING THE CONTEXT](#)

[Table 16.4: SPECIAL CHARACTERS IN A SQL STATEMENT AND THEIR REPLACEMENTS IN A URL](#)

[Table 16.5: XmlDocument Properties](#)

[Table 16.6: XmlDocument Methods](#)

[Table 16.7: XmlDocument Events](#)

[Table 16.8: XmlDataDocument Methods](#)

**Team LiB**

[!\[\]\(8ae66f640983acb0e5d9c4c8f7d04879\_img.jpg\) PREVIOUS](#)

[!\[\]\(a4b421af71f434127358111e1ab9c2f0\_img.jpg\) NEXT !\[\]\(f144695e4c140afc4410f4b43c43ad28\_img.jpg\)](#)

## List of Listings

### Chapter 1: Introduction to Database Programming with ADO.NET

[Listing 1.1: FIRSTEXAMPLE.CS](#)

[Listing 1.2: OLEDBCONNECTIONACCESS.CS](#)

[Listing 1.3: OLEDBCONNECTIONORACLE.CS](#)

### Chapter 4: Introduction to Transact-SQL Programming

[Listing 4.1: USING CURSORS](#)

[Listing 4.2: DISCOUNTPRICE.SQL](#)

[Listing 4.3: PRODUCTSTOBEREORDERED.SQL](#)

[Listing 4.4: PRODUCTSTOBEREORDERED2.SQL](#)

[Listing 4.5: ADDPRODUCT.SQL](#)

[Listing 4.6: PRODUCTAUDIT.SQL](#)

[Listing 4.7: INSERTPRODUCTTRIGGER.SQL](#)

[Listing 4.8: UPDATEUNITPRICEPRODUCTTRIGGER.SQL](#)

[Listing 4.9: DELETEPRODUCTTRIGGER.SQL](#)

### Chapter 5: Overview of the ADO.NET Classes

[Listing 5.1: SELECTINTODATASET.CS](#)

### Chapter 6: Introducing Windows Applications and ADO.NET

[Listing 6.1: Form1.cs](#)

### Chapter 7: Connecting to a Database

[Listing 7.1: MYSQLCONNECTION.CS](#)

[Listing 7.2: CONNECTIONPOOLING.CS](#)

[Listing 7.3: STATECHANGE.CS](#)

[Listing 7.4: INFOMESSAGE.CS](#)

### Chapter 8: Executing Database Commands

[Listing 8.1: EXECUTESELECT.CS](#)

[Listing 8.2: SINGLEROWCOMMANDBEHAVIOR.CS](#)

[Listing 8.3: SCHEMAONLYCOMMANDBEHAVIOR.CS](#)

[Listing 8.4: EXECUTETABLEDIRECT.CS](#)

[Listing 8.5: EXECUTESCALAR.CS](#)

[Listing 8.6: EXECUTEXMLREADER.CS](#)

[Listing 8.7: EXECUTEINSERTUPDATEDELETE.CS](#)

[Listing 8.8: EXECUTEDDL.CS](#)

[Listing 8.9: EXECUTETRANSACTION.CS](#)

[Listing 8.10: USINGPARAMETERS.CS](#)

[Listing 8.11: ADDPRODUCT.SQL](#)

[Listing 8.12: EXECUTEADDPRODUCT.CS](#)

[Listing 8.13: ADDPRODUCT2.SQL](#)

[Listing 8.14: ADDPRODUCT3.SQL](#)

[Listing 8.15: EXECUTEADDPRODUCT3.CS](#)

## **Chapter 9: Using *DataReader* Objects to Read Results**

[Listing 9.1: USINGCOLUMNORDINALS.CS](#)

[Listing 9.2: STRONGLYTYPEDCOLUMNVALUES.CS](#)

[Listing 9.3: STRONGLYTYPEDCOLUMNVALUESSQL.CS](#)

[Listing 9.4: EXECUTEMULTIPLESELECTS.CS](#)

[Listing 9.5: EXECUTEMULTIPLESQL.CS](#)

## **Chapter 10: Using *Dataset* Objects to Store Data**

[Listing 10.1: POPULATEDATASETUSINGSELECT.CS](#)

[Listing 10.2: POPULATEDATASETUSINGRANGE.CS](#)

[Listing 10.3: CUSTORDERHIST\(\) STORED PROCEDURE](#)

[Listing 10.4: POPULATEDATASETUSINGPROCEDURE.CS](#)

[Listing 10.5: MULTIPLEDATATABLES.CS](#)

[Listing 10.6: MULTIPLEDATATABLES2.CS](#)

[Listing 10.7: MULTIPLEDATATABLES3.CS](#)

[Listing 10.8:](#) MERGE.CS

[Listing 10.9:](#) MYXMLFILE.XML

[Listing 10.10:](#) MYXMLFILE2.XML

[Listing 10.11:](#) MYXMLSCHEMAFILE.XML

[Listing 10.12:](#) WRITEANDREADXML.CS

[Listing 10.13:](#) MAPPINGS.CS

[Listing 10.14:](#) MYDATASET.XSD

## **Chapter 11: Using *DataSet* Objects to Modify Data**

[Listing 11.1:](#) ADDRESTRICTIONS.CS

[Listing 11.2:](#) FILLCHEMA.CS

[Listing 11.3:](#) FINDFILTERANDSORTDATAROWS.CS

[Listing 11.4:](#) ADDPRODUCT4.SQL

[Listing 11.5:](#) UPDATEPRODUCT.SQL

[Listing 11.6:](#) DELETEPRODUCT.SQL

[Listing 11.7:](#) DELETEPRODUCT2.SQL

## **Chapter 12: Navigating and Modifying Related Data**

[Listing 12.1:](#) NESTEDXML.CS

[Listing 12.2:](#) NONNESTEDXMLFILE.XML

[Listing 12.3:](#) NESTEDXMLFILEL.CS

## **Chapter 13: Using *DataGridView* Objects**

[Listing 13.1:](#) USINGDATAVIEW.CS

[Listing 13.2:](#) FINDINGDATAROWVIEWS.CS

[Listing 13.3:](#) ADDMODIFYANDREMOVEDATAROWVIEWS.CS

[Listing 13.4:](#) CREATECHILDDATAVIEW.CS

[Listing 13.4A:](#) USINGDATAVIEWMANAGER.CS

## **Chapter 14: Advanced Transaction Control**

[Listing 14.1:](#) SAVEPOINT.SQL

[Listing 14.2:](#) SAVEPOINT.CS

[Listing 14.3: TransactionIsolation.sql](#)

[Listing 14.4: TransactionIsolation.cs](#)

[Listing 14.5: Block.cs](#)

[Listing 14.6: Deadlock.cs](#)

## **Chapter 15: Introducing Web Applications-ASP.NET**

[Listing 15.1: WebForm1.aspx](#)

[Listing 15.2: WebForm1.aspx.cs](#)

[Listing 15.3: THE WebForm1.aspx FILE](#)

[Listing 15.4: THE WebForm1.aspx FILE](#)

[Listing 15.5: THE WebForm1.aspx FILE](#)

[Listing 15.6: CookieTest.aspx](#)

[Listing 15.7: ViewStateTest.aspx](#)

[Listing 15.8: SessionObjectTest.aspx](#)

[Listing 15.9: ApplicationObjectTest.aspx](#)

## **Chapter 16: Using SQL Server's XML Support**

[Listing 16.1: FORXMLRAW.SQL](#)

[Listing 16.2: FORXMLAUTO.SQL](#)

[Listing 16.3: FORMLEXPLICIT.SQL](#)

[Listing 16.4: FORMLAUTOMIXED.SQL](#)

[Listing 16.5: FORMLAUTOELEMENTS.SQL](#)

[Listing 16.6: FORMLAUTOBINARYBASE64.SQL](#)

[Listing 16.7: CUSTOMERS.XML](#)

[Listing 16.8: CUSTOMERSWITHATTRIBUTES.XML](#)

[Listing 16.9: CUSTOMERSSTYLESHEET.XSL](#)

[Listing 16.10: CUSTOMERSUSINGSTYLESHEET.XML](#)

[Listing 16.11: CUSTOMERSFROMCOUNTRY.SQL](#)

[Listing 16.12: CUSTOMERS.XML](#)

[Listing 16.13: CUSTOMERSSTYLESHEET.XSL](#)

[Listing 16.14:](#) CUSTOMERSUSINGSTYLESHEET.XML

[Listing 16.15:](#) ADDCUSTOMERSXML.SQL

[Listing 16.16:](#) RUNADDCUSTOMERS.SQL

[Listing 16.17:](#) USINGXMLDOCUMENT.CS

[Listing 16.18:](#) USINGXMLDATADOCUMENT.CS

## **Chapter 17: Web Services**

[Listing 17.1:](#) CUSTOMERS.ASMX.CS

[Listing 17.2:](#) CUSTOMERSWEBSERVICE.CS

[Listing 17.3:](#) WEB SERVICE WSDL FILE

**Team LiB**

[!\[\]\(a22a2bdaafd9b6ec263d0d7c0d78a8b0\_img.jpg\) PREVIOUS](#) [!\[\]\(be0f724e9a5ac9be62fe14843ff539e4\_img.jpg\) NEXT](#)

## List of Sidebars

### **Chapter 9: Using *DataReader* Objects to Read Results**

[Using Table Joins](#)

### **Chapter 10: Using *Dataset* Objects to Store Data**

[RETRIEVING FROM MULTIPLE TABLES](#)

### **Chapter 11: Using *DataSet* Objects to Modify Data**

[CALLING THE `Fill\(\)` METHOD OF A `DataAdapter` MORE THAN ONCE](#)