ADO.NET Examples and Best Practices for C# Programmers

WILLIAM R. VAUGHN WITH PETER BLACKBURN

ADO.NET Examples and Best Practices for C# Programmers
Copyright © 2002 by Beta V Corporation and Boost Data Limited

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-012-0
Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Copy Editor: Christina Vaughn
Production Editor: Kari Brooks
Compositor: Diana Van Winkle, Van Winkle Design Group
Artist: Kurt Krames, Kurt Krames Design
Indexer: Carol Burbo
Cover Designer: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.
Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth St., Suite 219, Berkeley, CA 94710.
Email `info@apress.com` or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

CHAPTER 1

Introducing ADO.NET

Hijacked by Bill Vaughn's Inquisitor Peter Blackburn

Ahem! Perhaps I should mention that I needed to tie up Bill Vaughn in order to distill his world-class excellence on ADO.NET for the C# community. I am presently helping with Bill's rehabilitation. ...Now repeat after me, Bill, "C# is the bee's knees!" ... "Hmmmmph! Hmmmmph!" Ah! well yes I can see that we need just a little more assistance; I do hope I'll be able to remove the gag eventually…Wind the rack up a notch, Anders, would you please! ...

This book is all about using ADO.NET with C# (pronounced C sharp), .NET Framework,[1] and to some extent about how Visual Studio .NET helps you build ADO.NET-based applications. The concepts and code discussed and illustrated here apply (in most cases) to .NET Windows Forms and ASP Web Services and other ADO.NET platforms.

To make the transition to .NET easier for you and to clarify how I view this new technology, I start by helping you get familiar with .NET, its new terminology, and the new ways it allows you to access your data. There are many tutorials on .NET, most of which clearly describe the technology, albeit each from a unique and distinct point of view. In this book, my intended target audience is the experienced COM-based ADO developer. I focus strictly on my personal area of .NET expertise: data access and especially, data access with SQL Server. You might sense a bias in favor of Microsoft SQL Server (guilty) and the SqlClient namespace. Perhaps that's because I've had more experience coding, designing, implementing, testing, and teaching SQL Server than any other DBMS system. Again, in most cases, the OleDb and Odbc namespaces implement the System.Data classes (Microsoft.Data classes in the case of Odbc) in much the same way.

---

1. For an in-depth analysis of the .NET Framework check out Dan Appleman's *Moving to VB .NET: Strategies, Concepts and Code,* (Apress) ISBN: 1893115-97-6.

The Odbc .NET Data Provider is not a part of the Visual Studio .NET initial release—you'll need to download it directly from Microsoft's Web site. My informal tests show that the Odbc data provider, which uses Platform Invoke (PI), is faster than the OleDb data provider, which uses COM, although it is roughly twenty percent slower than the SqlClient data provider, which uses Tabular Data Stream (TDS). I talk a little more about this later. Before you decide to close your ears to the OleDb data provider for being the tortoise of the pack, just note that at present this is the *only* data provider that directly supports importing good ol' ADO Recordsets.

For differences and issues, check our Web sites or the Apress Web site[2] for updates sometime after this book hits the streets.

How We Got Here

A number of years ago, Microsoft found itself in yet another tough spot. Overnight (or so it seemed), the Internet had become far more popular than expected and Microsoft was caught without a viable development strategy for this new paradigm. Developers all over the world clamored for ways to get their existing code and skills to leverage Web technology. Even Microsoft's own internal developers wanted better tools to create cutting-edge Web content and server-side executables. These same developers also found that component object model (COM) architectures didn't work very well with or over the Internet—they were never designed to. Sun Microsystems' virtual stranglehold on Java and the ensuing fight over this language made it imperative that Microsoft come up with another way to create fast, light, language-neutral, portable, and scalable server-side executables.

Microsoft's initial solution to this challenge was to reconfigure their popular and well-known Visual Basic interpreter in an attempt to provide server-side (IIS) functionality to the tool-hungry developer community. To this end, VB Scripting Edition sprung to life, aimed at a subset of the four million Visual Basic developers trying to create logic-driven Web content for this new beast called "eCommerce." As many of these developers discovered, an Active Server Page (ASP) created with Visual Basic Script (VBScript) was relatively clunky when compared to "real" Windows-based Visual Basic applications and components. The VBScript language was confined to the oft-maligned Variant datatypes, copious late-binding issues, and interminable recompiles. Despite these issues, a flood of Web sites were built around this technology—probably because they were (loosely) based on a form of a familiar language: Visual Basic.

2. `http://www.betav.com`, `http://www.boost.net`, and `http://www.apress.com`.

Ahem! For those developers who had grown up using C and then its object layer abstraction C++ (these are the scary, awkward languages to the VB community—the ones with the curly braces {}, pointer things ->, and semicolons ;, and in the case of C++, OOP),[3] Microsoft offered JScript—a version of ECMAScript, which from a syntactical viewpoint is closer to C++ and JavaScript than Visual Basic. There were some advantages to be gained by using JScript over VBScript in client-side code, one of which being that, in theory, many other browsers, other than just those Microsoft offered, supported JScript, thereby potentially enabling the code to be browser neutral.

However, Microsoft sought some better way to satiate the needs of millions of Visual Basic developers and their ever-growing interest in the Web without compromising performance or functionality, perhaps providing them, maybe forcing them, to a new world of OOP without the need to learn JScript (or any other curly-brace language)!

It wasn't long before it became clear that Microsoft needed something new—no less than a whole new paradigm, a landslide shift, a new reality with some old familiar concepts, some new concepts, and some borrowed or adapted concepts—in order to accomplish this goal. This was the birth of the .NET platform.

Anders Hejlsberg, a Microsoft Distinguished Engineer,[4] crafted a brand new programming language for this new world reality. This language is C#, which fits with .NET hand in glove, horse and carriage, love and marriage, so to speak. Okay, so I like C#, but it isn't the only language that is now supported in .NET. Syntactically, C# is an OOP, curly-brace language, with semicolons, and thus a language with which C++ and Java developers will quickly feel comfortable and "at home."[5]

You see, Visual Basic just didn't cut it when compared to the heavily object-oriented Java applications with which it was competing. Before this, each new version of VB had inherited language and user interface (UI) supported functionality features from its predecessor. Yes, each new version usually left some unworkable functionality behind, but generally, these "forgotten" features were minor—most developers learned to live without them. When designing VB .NET, however, the Microsoft development team felt that too many of these "legacy" features hobbled Visual Basic's potential by preventing, or at least complicating, easy implementation of more sophisticated features. Thus, the advent of VB .NET.

---

3. OOP: Object-Oriented Programming—IPHO: Many of those who develop without it (as in totally unplanned and unstructured) tend to find that they have lots of places in their code at which they frequently have to exclaim "OOPs!" or other expletives as their code falls over.

4. Not to be confused with "Microsoft Drudge Engineers" who do less theoretical thinking and more real work trying to implement what the "Distinguished" engineers dream up.

5. Gary Cornell, co-founder of Apress and author (with Cay Horstmann) of the two-volume set, *Core Java 2* (Prentice Hall, ISBN: 0-13-089468-0), has been overheard saying that any Java programmer who cannot program proficiently in C# within half an hour of starting C# was probably not a Java programmer to begin with.

Unfortunately, as I see it, more than a few BASIC and Visual Basic developers really expect continued support for much of this "obsolete" functionality. Over the years, VB developers have learned (for better or worse) to depend on a forgiving language and an IDE that supports default properties, unstructured code, automatic instantiation, morphing datatypes, wizards, designers, drag-and-drop binding, and many more automatic or behind-the-scenes operations. More importantly, VB developers pioneered and depended on "edit and continue" development, which permitted developers to change their code after a breakpoint and continue testing with the new code. This was a radical departure from other development language interfaces and, for a decade, put Visual Basic in a class by itself.

IPHO *Microsoft's top engineers tried to move heaven and earth to get "edit and continue" functionality into Visual Basic .NET, but in the end, they just could not get it to work properly; so, it was dropped from the language. I expect that edit and continue is such a core part of the development methodologies used by so many Visual Basic 6.0 developers that Microsoft will be including it—just as soon as they can work out how to do it in a Garbage Collected world.*

Microsoft expects "professional" Visual Basic developers (whoever they are) to wholeheartedly embrace Microsoft's new languages—including the new "Visual Basic"—and (eventually) step away from Visual Basic as we know it today. Consider that a Visual Basic "developer" can be as sophisticated as a front-line professional who writes and supports thousands of lines of DNA code or as challenged as an elementary school student or part-time accountant creating a small application against an Access database. Some of these developers will be skilled enough and motivated enough to adapt to a new language—some will not. Some have the formal training that permits them to easily step from language to language—many (I would venture the majority) do not. Some professional developers, faced with this magnitude of change, will opt to find another language or another seemingly simpler occupation, such as brain surgery.

IMHO *Microsoft continues to complicate the situation by insisting that VB .NET is* ***really*** *just another version of Visual Basic 6.0 and that ADO.NET is just another version of COM-based ADO. They clearly aren't the same—not even close.*

IPHO *Those "professional" Visual Basic developers might very well go just that tiny bit further and take the opportunity to learn and then use C# as their language of choice. The way I look at it, Visual Basic .NET is almost a case-insensitive version of C# without the braces and semi-colons (and an inbuilt default of go-slow…); and for me, alas not Bill (yet), C# "feels" cleaner (more syntactically correct), the block structures of the language are clearer, and I really like the cool automated self-documentation of comments in the code to HTML Web pages.*

I tested the performance of examples in Visual Basic .NET against similar C# examples. If in Visual Basic .NET you remember to set "Option Strict On" (that means take the go-slow default Off), then the MSIL (Microsoft Intermediate Language) produced by the compilers for either language is very, very similar—almost identical, but not quite. If you don't set "Option Strict On" and you leave Visual Basic .NET to its default of loose type checking, then C# is always much faster. In my friendly sparring fights with Dan Appleman, he was able to convince me that Visual Basic could usually get very close to C# performance—at least in our tests to within the region of "noise" (single figures of ticks apart over millions of repetitions). I always found that there was always more "noise" affecting Visual Basic .NET tests than C#.

I think the new Visual Basic .NET language is just that: new. (*Ahem!* V sharp?) While it emulates the Visual Basic language in many respects, it's really not the same. As many of you have heard, I wanted to call it something else—anything else—but my daughter, Fred, told me to keep my mouth shut to prevent her from further embarrassment. I complied, as I don't want to give anyone at Microsoft apoplexy—again.

What Do These Changes Mean?

The Microsoft .NET Framework's system of language(s), tools, interfaces, and volumes of supporting code has been constructed from the ground up with an entirely new architecture. For those of you who remember IBM 3270 technology, you'll find that the .NET Framework tracks many of the same wheel ruts laid into the road during the 1960s. IBM 3270 systems were central processor (mainframe)–driven "smart" (or "dumb") terminal designs. They relied on a user-interface terminal which supported very sparse functionality. The terminal's only function was to display characters at an x-y coordinate and return characters entered into "fields." There were no mice or graphics to complicate things, but a dozen different keyboard layouts made life interesting.

While the industry's current browser technology includes far more intelligence and flexibility at the client, the general design is very similar to the 3270 approach. .NET applications now expect code similar to a browser to render the forms and frames and capture user input, even when creating a Windows Forms application. This means .NET applications will behave and interact differently (at least to some extent) than "traditional" Windows applications.

What's new for server-side executables is the concept of a Web Service. I discuss and illustrate Web Services in Chapter 10, "ADO.NET and XML." This new paradigm finds its roots in Visual Basic 6.0's so-called IIS Applications—better known as Web Classes. Web Services place executable code on your IIS server to be referenced as ASP pages or from other executables such as WinForm applications just as you would reference a COM component running in the middle tier. The big difference is that Web Services do not require COM or DCOM to expose their objects, methods, properties, or events—they are all exposed through SOAP.[6] I explain what this means in Chapter 10.

For the C++ developer moving to C#, these .NET innovations mean that the huge Rapid Application Development (RAD) advantages that Visual Basic developers had over C++ developers are no more, no longer, gone, zip; there is now a level playing field. Previously, C++ Windows Application developers had to do battle fighting with the Microsoft Foundation Classes (MFC), while their Visual Basic developer cousins needed only to tinker with the facile "Ruby" Windows Form Engine. They rarely bothered, cared, or needed to know what a Windows handle or a device context was, but were by far more visibly productive.
This leveling of the playing field has been achieved in part by replacing Visual Basic's "Ruby" forms engine and the accompanying run-time library (VBRUN.DLL) with a new run-time platform and forms engine, as well as a new user interface and development IDE. (If I can use the word "replaced" to mean that the new version does not implement the same functionality.) Saying the Visual Basic run time has been replaced is like saying the diesel engine in a semi-tractor-trailer rig was replaced with a cross-galaxy transport mechanism.

The Visual Basic 6.0 IDE, the Visual InterDev 6.0 IDE, and the Visual C++ 6.0 IDE have been replaced with a new "combined" system that integrates all of the language front ends into one. From the looks of it, Microsoft used the Visual Studio 6.0–era Visual InterDev shell as a base. These changes mean that Visual Basic .NET is *not* just the newest version of Visual Basic. While Visual Basic .NET is similar in some respects to Visual Basic 6.0, it's really a lot more like C# (pronounced C sharp) or C++ (pronounced C hard-to-learn). For the professional, school-trained veterans out there, VB .NET and C# are just other languages. For many, though, they're a big, scary step away from their comfort zone.

---

6. Simple Object Access Protocol. See `http://www.w3.org/TR/SOAP/#_Toc478383486`.

ADO.NET—A New Beginning

This section of the book introduces something Microsoft calls ADO.NET. Don't confuse this new .NET data access interface with what we have grown to know and understand as ADO—I think it's really very different. Yes, ADO.NET and ADOc both open connections and fetch data, however, they do so in different ways using different objects and with different limitations. No, they aren't the same—no matter what Microsoft names them. Yes, ADO.NET has a Connection object, Command object, and Parameter objects (actually implemented by the SqlClient, OleDb, and Odbc .NET Data Providers), however, they don't have the same properties, methods, or behaviors as their ADOc counterparts. IMHO, this name similarity does not help to reduce the confusion you're likely to encounter when transitioning from ADOc to ADO.NET.

NOTE *To avoid confusion, I've coined a new term to help you distinguish the two paradigms; henceforth "ADOc" refers to the existing COM-based ADO implementation and "ADO.NET" refers to the new .NET Framework implementation.*

Actually, the name ADO.NET was not Microsoft's first choice (nor is it mine) for their new data access paradigm. Early in the development cycle (over three years ago),[7] their new data access object library was referred to as XDO (among other things). To me, this made[8] a lot of sense because ADO.NET is based on XML persistence and transport—thus "XML Data Objects" seemed a good choice. Because developers advised Microsoft to avoid the creation of yet another TLA (three-letter acronym)–based data access interface, they were hesitant to use the XDO moniker. I suspect there were other reasons too—mostly concerning the loss of market product name recognition. So, XDO remains one of those words you aren't supposed to mention in the local bar. Later in the development cycle, XDO evolved into ADO+ to match the new ASP+ technology then under construction. It was not until early in 2001 that the name settled on ADO.NET to fit in with the new naming scheme for Windows XP (Whistler) and the newly dubbed .NET Framework.

Microsoft also feels that ADO.NET is close enough to ADOc to permit leveraging the name and making developers feel that ADO.NET is just another version of ADOc. That's where Microsoft and I differ in opinion. The documentation

---

7. Circa AD 1999.
8. I was opposed to another TLA at the time—for some reason that now escapes me.

included ever since the first .NET betas assures developers that ADO.NET is designed to "…leverage current ADO knowledge." While the connection strings used to establish connections are similar (even these are not *exactly* the same as those used in ADOc), the object hierarchy, properties, methods, and base techniques to access data are all very different. Over the past year I often struggled with ADO.NET because I tried to approach solutions to my data access problems using ADOc concepts and techniques. It took quite some time to get over this habit (I joined a twelve-step program that worked wonders). Now my problem is that when someone asks me an ADOc question, I have to flush my RAM and reload all of the old concepts and approaches. I'm getting too old for this.

No matter what you call it, I think you'll also discover that even though ADO.NET is different from ADOc in many respects, it's based on many (many) years of development experience at Microsoft. It's not *really* built from scratch. If you look under the hood you'll find that ADO.NET is a product of many (but not all) of the lessons Microsoft has learned over the last decade in their designing, creating, testing, and supporting of DB-Library, DAO, RDO, ODBCDirect, and ADO, as well as ODBC and OLE DB. You'll also find remnants of the FoxPro and Jet data engines, shards from the Crystal report writer, as well as code leveraged from the ADO Shape, ADOX, and ADOMD providers. Unfortunately, you'll also find that ADO.NET's genes have inherited some of the same issues caused by these technologies—it also suffers from a few "DNA" problems; I discuss these as I go. Most of these issues, however, are just growing pains. I expect there will be a lot of lights left on at night trying to work them out—unless the energy crisis has us working by candlelight by then.

That said, don't assume that this "new" ADO.NET data access paradigm implements all of the functionality you're used to seeing in ADOc. Based on what I've seen so far, there are lots of features—among them many important ones—left behind. I discuss these further in the following chapters.

Comparing ADOc and ADO.NET

Data access developers who have waded into the (generally pretty good) MSDN .NET documentation might have come across a topic that compares ADOc with ADO.NET. IMHO, this topic leaves a lot to be desired; it slams ADOc pretty hard. Generally, it ignores or glosses over features such as support for the Shape provider (which exposes hierarchical data management), pooled connections and intelligent connection management, disconnected Recordsets, serialization, XML functionality, ADOMD, and ADOX. Yes, ADO.NET is a new and innovative data access paradigm, but so is ADOc. In its defense, the documentation does say there are still a number of situations where ADOc is the (only) solution. I suspect that

the Microsoft .NET developers will make ADOc redundant over time—just not right away.

Later in this and subsequent chapters I visit the concept of porting ADOc code over to .NET applications. It's a complex subject full of promise and some serious issues—a few with no apparent resolution. Stay tuned.

IMHO *The job of a technical writer at Microsoft is considerably challenging. I worked on the Visual Basic user education team for about five years and, while some changes have been made, there are still many issues that make life tough for writers, editors, and developers alike—all over the world. One of the problems is that when working with a product as new as .NET, there are few "reliable" sources of information besides the product itself. Unfortunately, the product is a moving target—morphing and evolving from week to week, sometimes subtly, but just as often in radical ways as entire concepts are lopped off or jammed in at the last minute for one reason or another. This problem is especially frustrating when outsiders work with beta versions. To add to Microsoft's problems, they have to "freeze" the documentation months (sometimes six or more) in advance, so it can be passed to the "localizers." These folks take the documentation and translate into French, German, Texan, and a number of other foreign languages. A lot can (and does) happen in the last six months before the product ships. If the product doesn't ship—this has happened on more than one occasion—it is also difficult to keep the documentation in sync.*

Another factor you need to consider is your investment in ADOc training and skills. Frankly, quite a bit of this will be left behind if you choose ADO.NET as your data access interface. Why? Because ADO.NET is that different. This issue will be clearer by the time you finish this book.

Understanding ADO.NET Infrastructure

Microsoft characterizes ADO.NET as being designed for a "loosely coupled, highly distributed" application environment. I'm not sure that I wholly agree with this characterization. I'll accept the "loosely coupled" part, as ADO.NET depends on XML—not proprietary binary Recordsets or user-defined structures—as its persistence model and transport layer. No, ADO.NET does not store its in-memory DataTable objects as XML, but it does expose or transport them as XML on demand. As I see it, XML is one of ADO.NET's greatest strengths, but also one of its

weaknesses. XML gives ADO.NET (and the entire .NET Framework) significant flexibility, which Visual Basic 6.0 applications have to go a long way to implement in code—and C++ applications a little further still. However, XML is far more verbose and more costly to store and transmit than binary Recordsets; granted, with very small data sets, the difference isn't that great. By passing XML instead of binary, ADO.NET can pass *intelligent* information—data and schema and extended properties, or any other attribute you desire—and pass it safely (and securely) through firewalls. The only requirement on the receiving end is an ability to parse XML—and that's now built into the Windows OS.

Understanding ADO.NET's Distributed Architecture

As far as the "highly distributed" part of the preceding ADO.NET characterization, I think Microsoft means that your code for .NET applications is supposed to work in a stand-alone fashion without requiring a persistent connection to the server. While this is true, I expect the best applications for .NET will be on *centralized* Web servers where the "client" is launched, constructed, and fed through a browser pointing to a logic-driven Web page. I think that Microsoft intended to say that ADO.NET is designed primarily for Web architectures.

On the other hand, ADO.NET (in its current implementation) falls short of a universal data access solution—one of ADOc's (and ODBC's) major selling points. The ODBC provider (Microsoft.Data.Odbc) is not included in the .NET Framework but is to be made available through a Web update sometime after .NET is initially released. I don't think one can really interpret this as a policy to back away from the universal data access paradigm—but it would not be hard to jump to that conclusion. I'm disappointed that ODBC is not part of the initial release. But better late than never.

In my opinion, the most important difference between ADO.NET and any other Microsoft data access interface to date is the fact that ADO.NET is multidimensional from the ground up. That is, ADO.NET:

- **Is prepared to handle—with equal acuity—either single or multiple related resultsets along with their relationships and constraints.**
- **Does not try to conjure the intratable relationships—it expects you to define them in code.** But it's up to you to make sure these coded relationships match those defined by your DBA in the database. It might be nice if Visual Studio .NET could read these definitions from the server, but then again, that would take another round trip. Be careful what you ask for…
- **Permits you to (expects you to) define constraints in your application to ensure referential integrity.** But again, it's up to you to keep these in sync with the database constraints.

- **Does not depend on its own devices for the construction of appropriate SQL statements to select or perform updates to the data—it expects you to provide these.** You (or the IDE) can write ad hoc queries or stored procedures to fetch and update the data.

In some ways, this hierarchical data approach makes the ADO.NET disconnected architecture far more flexible and powerful than ADOc—even when including use of the Shape provider in ADOc. In other ways, you might find it difficult to keep component-size relationships and constraints synchronized with their equivalents in the database.

A Brief Look at XML

No, I'm not going to launch into a tutorial on XML, just as I found it unnecessary to bury you in detail about the binary layout of the Recordset (not that I know anything about it). I do, however, want to fill in some gaps in terminology so that you can impress your friends when you start discussing ADO.NET.

XML is used behind the scenes throughout ADO.NET and you ordinarily won't have to worry about how it's constructed until ADO.NET, or an application passing XML to you, gets it wrong. Just remember that the ADO.NET DataSet object can be constructed directly from XML; this includes XML generated by any application that knows how to do it (correctly). The .NET architecture contains root services that let you manage XML documents using familiar programming constructs.

As I said, when you transport your data from place to place (middle tier to client, Web Service to browser), ADO.NET passes the data as XML. However, XML does not describe the database schema by itself—at least not formally. ADO.NET and the .NET IDE know how to define and persist your data's schema using another (relatively new) technology called Extensible Schema Definition (XSD). Accepted as a standard by the W3C[9] standards organization, XSD describes XML data the same way database schemas describe the structure of database objects such as tables. XSD provides a way to not only understand the data contained within a document, but also to validate it. XSD definitions can include datatype, length, minlength, maxlength, enumeration, pattern, and whitespace.[10] Until recently, XML schemas have been typically created in the form of Document Type Definitions (DTDs), but Visual Studio .NET introduces XSD, which has the advantage of using XML syntax to define a schema, meaning that the same parsers can process both data and schemas.

---

9. See `http://www.w3.org` for more information.
10. I expect this list to change (expand, contract) as XSD is nailed down.

IIRC,[11] XSD has been W3C final recommendation status for several months. Visual Studio .NET can generate XSD schemas automatically, based on an XML document. You can then use it to edit the schema graphically to add additional features such as constraints and datatypes. There are also .NET tools that can help construct XSD from a variety of forms including Recordsets, XML data structures, and others.

Later in the book (Chapter 10) I discuss how you can use the XML tools in .NET to manage your data.

ADO.NET—The Fundamentals

For those developers familiar with ADOc and the disconnected Recordset, ADO.NET's approach to data access should be vaguely familiar. The way in which you establish an initial connection to the database is very similar to the technique you used in ADOc—at least on the surface. After that, the similarity pretty much ends.

There are several base objects in ADO.NET. These objects are outlined and briefly described several times in this chapter and discussed in depth in subsequent chapters. Each of the following objects are implemented from base classes in the System.Data namespace by each of the .NET Data Providers:

- **The Connection object**: This works very much like the ADOc Connection object. It's not created in the same way nor is the ConnectionString property exactly the same, but it's close.

- **The Command object**: This works very much like an ADOc Command object. It holds a SQL `SELECT` or action query and points to a specific Connection object. The Command object exposes a Parameters collection that works something like the ADOc Command object's Parameters collection.

- **The DataReader object**: This is used to provide raw data I/O to and from the Connection object. It returns a bindable data stream for WebForm applications and is invoked by the DataAdapter to execute a specific Command.

- **The DataAdapter object**: There is no exact equivalent to this in ADOc; the closest thing is the IDE-driven Visual Basic 6.0 Data Environment Designer. The DataAdapter manages a set of Command objects used to fetch, update, add, and delete rows through the Connection object.

---

11. IIRC: If I recall correctly.

- **The DataTable object**: Again, there is not an ADOc equivalent, but it's similar in some respects to the Recordset. The DataTable object contains a Rows collection to manage the data and a Columns collection to manage the schema. No, DataTables do not necessarily (and should not) be thought of as base tables in the database.

- **The DataSet object**: This is a set of (possibly) related DataTable objects. This interface is bindable in Windows Forms or WebForms. The DataSet also contains Relations and Constraints collections used to define the interrelationships between its member DataTable objects.

A Typical Implementation of the ADO.NET Classes

One approach (there are several) calls for your application to extract some (or all) of the rows from your database table(s) and create an ADO.NET DataTable. To accomplish this, you create a Connection object and a DataAdapter object with its SelectCommand set to an SQL query returning data from a single table (or from several tables using separate SELECT statements in a single Command).

The DataAdapter object's Fill method opens the connection, runs the query through a DataReader (behind the scenes), constructs the DataTable objects, and closes the connection. If you use individual queries, this process is repeated for any related tables—each requiring a round trip, separate queries, and separate DataTable objects. However, if you're clever, you can combine the `SELECT` operations into a single query. ADO.NET is smart enough to build each resultset of a multiple-resultset query as its own DataTable object. I show an example of this in Chapter 5, "Using the DataTable and DataSet."

After the DataTable objects are in place, your code can disconnect from the data source. Actually, this was already done for you; ADO.NET opens and closes the Connection object for you when you use the Fill method. Next, your code can define the primary key/foreign key (PK/FK) relationships and any constraints you want ADO.NET to manage for you. All work on the data takes place in client memory (which could be in a middle-tier component, ASP, or distributed client's workstation).

When working with related (hierarchical) data, you can write a `SELECT` query to extract all or a subset of the customer's table rows into a DataTable object. You can also create queries and code to construct additional DataTable objects that contain rows in the related Orders and Items database tables. Code a single bindable DataSet object to manage all of these DataTable objects and the relationships between them. Behind the scenes, ADO.NET "joins" these DataTable objects in memory based on *your* coded relationships. This joining of DataTable objects permits ADO.NET to navigate, display, manage, and update the DataSet object, the DataTable objects, and ultimately, the database tables behind them when you

use the Update method. After ADO.NET fetches the queried rows to construct the DataSet, ADO.NET (or your code) closes the connection and no longer depends on the database for any further information about the data or its schema.

When called upon to update the database, ADO.NET reopens the connection and performs any needed UPDATE, INSERT, or DELETE operations defined in the DataAdapter as separate Command objects. Your code handles any collisions or problems with reconciliation.

The Visual Studio .NET IDE lets you use drag-and-drop and a number of wizards to construct much of the code to accomplish this. As I discuss in later chapters (see Chapter 4, "ADO.NET DataReader Strategies") you might not choose to avail yourself of this code—it's kinda clunky. As with ADOc's Shape provider, ADO.NET can manage intertable relationships and construct a hierarchical data structure that you can navigate and update at will—assuming you added code to define the relationships and constraints. I show you how to do this in Chapter 5 and in Chapter 8, "ADO.NET Constraint Strategies."

Based on my work with ADO.NET so far, I have a number of concerns regarding the disconnected DataSet approach:

- **The overhead involved in downloading high volumes of data and the number of locks placed on the server-side data rows is problematic at best.** The ADO.NET disconnected DataSet approach might work for smaller databases with few users, but you must be careful to reduce the number of rows returned from each query when dealing with high volumes of data. Sure, it's fast when you test your stand-alone application, but does this approach scale?

- **Assumes that the base tables are exposed by the DBA; in many shops, this is not the case, for security and stability reasons.** While you can (and should) construct DataSet objects from stored procedures, you also need to provide stored procedures to do the UPDATE, DELETE, and INSERT operations. It's not clear if this approach will permit ADO.NET to expose the same functionality afforded to direct table queries—it does not appear to. I have found, however, that it is possible to perform updates against complex table hierarchies, but it requires more planning and work than the simplistic table-based queries often illustrated in the documentation.

- **The Visual Studio .NET drag-and-drop and wizards used to facilitate ADO.NET operations generate (copious) source code.** That's the good news. The bad news is that this source code has to change when the data structures, relationships, or stored procedures used to manage the data change—and this does not happen automatically. This means that you want to make sure your schema is nailed down before you start generating a lot of source code against it. Once inserted, it's often tough to remove this code in its entirety if you change your mind or the schema.

- **The disconnected approach makes no attempt to maintain a connection to the data source.** This means that you won't be able to depend on persisted server-side state. For example, server-side cursors, pessimistic locks, temporary tables, or other connection-persisted objects are not supported.

- **When compared to ADOc, ADO.NET class implementation is fairly limited in respect to update strategies.** As you'll see in Chapter 3, "ADO.NET Command Strategies," and Chapter 7, "ADO.NET Update Strategies," the options available to you are nowhere near those exposed by ADOc—especially in regard to Update Criteria.

ADO.NET .NET Data Providers

A fundamental difference between ADOc and ADO.NET is the latter's use of .NET Data Providers. A .NET Data Provider implements the base System.Data classes to expose the objects, properties, methods, and events. Each provider is responsible for ADO.NET operations that require a working connection with the data source. The .NET Data Providers are your direct portals to existing OLE DB providers (System.Data.OleDb), ODBC drivers (Microsoft.Data.Odbc), or to Microsoft SQL Server (System.Data.SqlClient). ADO.NET (currently) ships with two .NET Data Providers:

- **System.Data.OleDb**: Used to access existing Jet 4.0 and Oracle OLE DB providers via COM interop, but notably not the ODBC (MSDASQL) provider—the default provider in ADOc.[12]

- **System.Data.SqlClient**: Used to access Microsoft (and just Microsoft) SQL Server versions 7.0 and later.

NOTE *The System.Data.SqlClient provider is designed to access Microsoft SQL Server 7.0 or later. If you have an earlier version of SQL Server, you should either upgrade (a great idea),or use the OleDb .NET Data Provider with the SQLOLEDB provider or simply stick with ADOc.*

12. I expect that other .NET Data Providers will appear very soon after .NET ships.

As I said earlier, the Microsoft.Data.Odbc provider was made available via Web download not long after .NET was released to the public. It is used to access most ODBC data sources. No, it's not clear that all ODBC data sources will work with ADO.NET. Initial tests show, however, that this new Odbc .NET Data Provider is twenty percent faster than its COM interop brother, the OleDb .NET Data Provider. This is to be expected because COM is very "chatty," requiring more server round trips than ODBC to get the same data. The Odbc .NET Data Provider uses the more efficient Platform Invoke.

As I said, the ADO.NET OleDb provider uses COM interop to access most existing OLE DB providers—but this does *not* include the ODBC provider (MSDASQL). This also does not mean you can use any existing OLE DB providers with System.Data.OleDb. Only the SQLOLEDB (Microsoft SQL Server), MSDAORA (Oracle), and Microsoft Jet OLEDB.4.0 (Jet 4.0) providers are supported at RTM.[13] Notably missing from this list is MSDASQL—the once-default ODBC provider. In addition, none of the OLE DB 2.5 interfaces are supported, which means that OLE DB providers for Exchange and Internet Publishing are also not (yet) supported in .NET. But, remember that the .NET architecture lends itself to adding additional functionality; I would not be surprised if additional providers appeared before too long.

However, consider that these data access interfaces are very different from the OLE DB or ODBC providers with which you might be accustomed. ADO.NET and the .NET Data Providers implemented so far know nothing about keyset, dynamic, or static cursors, or pessimistic locking as supported in ADOc. Sure, the ADO.NET DataTable object looks something like a static cursor, but it does not share any of the same ADOc `adOpenStatic` properties or behaviors with which you're familiar. They don't leverage server-side state or cursors—regardless of the data source. ADO.NET has its own hierarchical JOIN engine so it doesn't need the server to do anything except run simple (single-table) `SELECT` queries. Whether it makes sense to let ADO.NET do these JOIN operations for you is another question.

A .NET Data Provider is responsible for far more functionality than the low-level ODBC or more sophisticated (and complex/bulky/slow/troublesome) OLE DB data providers in ADOc. A .NET Data Provider implements the System.Data objects I described earlier that are fundamental in the implementation of your ADO.NET application. For example:

- **The Command object**: SqlCommand, OleDbCommand, OdbcCommand

- **The Connection object**: SqlConnection, OleDbConnection, OdbcConnection

---

13. RTM: Release to manufacturing.

- **The DataAdapter object**: SqlDataAdapter, OleDbDataAdapter, OdbcDataAdapter
- **The DataReader object**: SqlDataReader, OleDbDataReader, OdbcDataReader

.NET Data Providers also directly support and implement code to generate Commands, and control the connection pool, procedure parameters, and exceptions. It's clear that .NET Data Providers bear far more responsibility than their ADOc predecessors did. I expect that this also means that the features exposed by one provider might not be supported in the same way or with the same issues (bugs) as another. Of course, this has always been the case with ADOc and its predecessors. Anyone who's worked with ODBC and transitioned to OLE DB in ADOc can bore you with war stories about how "stuff" changed from one implementation to the next. I'm sure we'll see some of the same in ADO.NET.

I think the fact that the .NET Data Provider for SQL Server speaks Tabular Data Stream (TDS) is a *very* important innovation. Not only do I think this will help performance (it will), but it also means Microsoft is not afraid of creating a Microsoft SQL Server–specific interface (no, it does not work with Sybase SQL Server). This opens the door for better, more intimate control of Microsoft SQL Server systems from your code without having to resort to SQLDMO. It also implies that native Oracle, Sybase, and other high-performance native .NET Data Providers are possible. Your guess is as good as mine as to when these will actually appear; for those players who want to stay in the game, I expect sooner rather than later.

Leveraging Existing COM-based ADO Code

The .NET Framework is flexible enough to support more than just the three .NET Data Providers I've mentioned. This adaptability is especially important in light of ADO.NET's architecture, which leaves out a number of data access paradigms that you might find essential to your design. But up to this point, all of you have invested many (many) hours/months/years of work on ADOc code imbedded in all types of applications, middle-tier components, and Web-based executables. The burning question most of you have is "Can I leverage this investment in ADOc in my .NET executables?" The answer is not particularly clear. First, you'll find that you can imbed ADOc code in a .NET executable—while it might not behave the same, .NET applications, components, and Web Services can execute most (but not all) COM-based code.

NOTE *Visual Studio .NET includes an (excellent) conversion utility to take existing ADOc code and convert it. However, it does not convert it to ADO.NET code—it's converted to COM interop-wrapped ADOc code designed to run in a .NET application. While this utility converts the code, it does not convert the architecture or query strategy. These might not be appropriate for your new .NET application.*

Fundamentally, there are two approaches to access existing ADOc objects from .NET executables. First, you can simply reference adodb (the COM interop wrapper around MSADO15.DLL) and include "`using adodb;`" in your solution. In this approach, you access the objects and their properties and methods directly. The problem is that each and every time you reference an ADOc object (or any COM object), property method, or event, the Common Language Runtime (CLR) has to make the reference to and from the COM interop layer. This will slow down the references to some degree and if the interop does not behave, it might impair functionality. We already know this is the case when it comes to executing stored procedures as methods of the ADODB.Connection object—it's no longer supported. There are other issues as well, as I discuss in Chapter 2, "ADO.NET—Getting Connected."

Another approach for accessing existing ADOc objects from .NET executables is to encapsulate your ADOc (or other COM object reference) code in its own wrapper. With this approach, you only access specific methods of the wrapper object, which execute blocks of ADOc code. Few if any properties are exposed. This approach resembles what you do to implement a middle-tier COM component. It also means that you spend far less time in the interop layer—once when you enter the wrapper DLL and once when you return. The problem here is that you often have to reengineer your ADOc code, resulting in some loss of flexibility in coding directly to the ADOc objects.

When importing ADOc code you have to instantiate your objects differently. I walk through several ADOc examples in Chapter 2. There you'll discover that some of the methods work differently—for example, you can't use the GetRows method to return a Variant array, and your simple constants must now be fully qualified—but for the most part, ADOc codes about the same, after all it's just another COM object with properties, methods, and events, and those remain just the same. However, as I said before, you might notice a drop in performance or somewhat different behavior due to COM interop.

Creating DataSets from ADOc Recordset Objects

The .NET developers knew that some of you would want to import ADOc Recordsets from existing COM components and create ADO.NET DataSets; fortunately, this is easy in ADO.NET. The OleDbDataAdapter Fill method directly recognizes ADOc Recordset and Record objects to populate a DataTable in a DataSet. This functionality enables .NET developers to use existing COM objects that return ADO objects without having to rewrite new objects using the .NET Framework. But as of the release of Visual Studio .NET, *only* the OleDb .NET Data Provider supports filling a DataSet from an ADO Recordset or Record object. I illustrate this with an example in Chapter 4.

How COM Interop Affects .NET Applications

As I said before, all "unmanaged" code executed by the CLR must be handled differently from "managed" code. Because of this stipulation between managed and unmanaged code, all of the ADOc and the ADO.NET OleDb .NET Data Provider data I/O operations are processed through a COM interop "wrapper." (The ADO.NET SqlClient .NET Data Provider does not use COM interop.) This extra layer on legacy COM components makes the .NET application think it's communicating to a .NET component and the COM-based code thinks that it's communicating to a COM-based host. Figure 1-1 illustrates this extra layer of protection wrapped around all COM components.

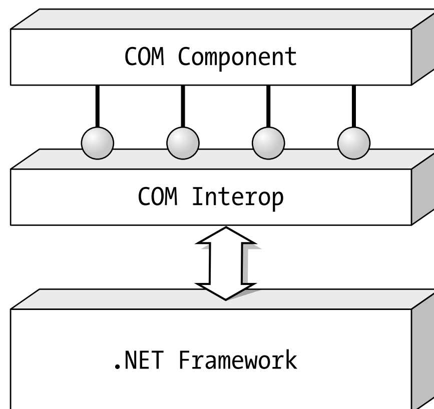

Figure 1-1. COM components access .NET via the COM interop layer.

I suspect we'll see a few side effects caused by this additional translation layer that can't help but hurt performance. COM interop is something like ordering a hamburger from a Spanish-speaking clerk at your local burger palace through a speakerphone. If you don't speak Spanish, the result might have *un poco más cebolla*[14] than you planned on—but for me, that's okay!

One of the major (that should be MAJOR) differences in the .NET Framework is that your .NET application assembly is built using a *specific* version of ADOc DLLs (msado15.dll) and all of the other COM DLLs and components it references. In fact, these DLLs can be (should be?) copied from their common location to the assembly's disk directory. This means you could have the ADO run-time DLLs installed any number of times on your disk—*n* copies of the same ADO DLLs or *n* different versions of the ADO DLLs.

When you start a .NET application, the DLLs used and referenced at design/test/debug/compile time are referenced at run time. This means your application behaves (or misbehaves) the same way it did when you wrote and tested it. Imagine that. If the version of ADO (or any other dependent DLL) gets updated (or deprecated) later, or you deploy to a system with different DLLs, your existing applications still install and load the "right" (older, newer, or the same) version of ADO and your other DLLs. This means that "DLL hell" as we know it has become a specter of the past—at least when all of your applications are based on .NET. I expect DLL hell applications will still be haunting us for decades to come—rattling their chains in the back corridors of our systems and playing evil tricks on unsuspecting tourists.

I walk you through converting and accessing ADOc objects in the next chapter.

ADO.NET and Disconnected Data Structures

ADO.NET constructs and manages DataSet and DataTable objects without the benefit of server-side cursors or persisted state. These objects roughly parallel the disconnected Recordset approach used in ADOc. Remember, ADO.NET provides no support for pessimistic (or any other kind of) locking cursors—all changes to the database are done via optimistic updates. ADO.NET does not include the entire "connected" paradigm supported by every data access interface since DB-Library. Microsoft suggests that developers simply use existing ADOc code wrapped in a COM interop layer for these designs—or stick with Visual Basic 6.0 (*Ahem! or Visual C++—especially for MTS/COM+ ADOc components that use object pooling).*

---

14. A little more onion.

Behind the scenes, ADO.NET's architecture is (apparently) built around its own version of ADOc's Shape provider. It expects the developer to download separate resultsets (Tables) one at a time (or at least in sets). This can be done by using separate round trips to the data source or through multiple-resultset queries. After the DataTable is constructed, you're responsible for hard coding the parent/child relationships between these tables—that is, if you want ADO.NET to navigate, join, manage, display, and update hierarchical data and eventually post batches of updates to the back-end server. All of this is done in RAM with no further need of the connection or the source database. I'm not sure what happens when the amount of available RAM and swap space is exhausted using this approach. There is some evidence to suggest that your system might try to order more from the Web. Just don't be surprised to get a package in the mail addressed to your CPU. I expect that performance and functionality will also suffer to some degree—to say the least. This "in-memory database" approach means that you developers will have to be even more careful about designs and queries that extract too many rows from the data source. But this is not a new rule; the same has always applied to DAO, RDO, and ADOc as well, most especially in client/server circumstances.

The System.Data Namespace

Before I start burrowing any deeper into the details of the .NET System.Data object hierarchy, I'll define a term or two. For those of you who live and breathe object-oriented (OO) concepts, skip on down. For the rest of you, I try to make this as clear as I can despite being a person who's been programming for three decades without using "true" OO.

The .NET Framework is really a set of classes organized into related groups called namespaces. See "Introduction to the .NET Framework Class Library" in .NET Help for the long-winded definition. When you address the specific classes in a namespace you use dot (.) notation—just as you do in COM and did in pre-COM versions of Visual Basic. Thus, "System" is a namespace that has a number of subordinate namespaces associated with it. System.Data.OleDb defines a specific "type" within the System.Data namespace. Basically, everything up to the right-most dot is a namespace—the final name is a type. The System.Data namespace contains the classes, properties, methods, and events (what .NET calls "members") used to implement the ADO.NET architecture. When I refer to an "object," it means an instantiation of a class. For example, when I declare a new OleDbConnection object, I do so by using the `new` constructor on the OleDbConnection class.

```
System.Data.OleDbConnection myConnection = new System.Data.OleDbConnection();
```

Clear? Don't worry about it. I try to stay focused on the stuff you *need* to know and leave the OO purists to bore you with the behind-the-scenes details. See MSDN .NET[15] for more detailed information on the System.Data namespace.

The ADO.NET DataSet Object

The System.Data.DataSet object sits at the center of the ADO.NET architecture. While very different from an ADOc Recordset, it's about as close as you're going to get with ADO.NET. As with the ADOc Recordset, the DataSet is a bindable object supporting a wealth of properties, methods, and events. While an ADOc Recordset can be derived from a resultset returned from a query referencing several database tables, it's really a "flat" structure. All of the hierarchal information that defines how one data table is related to another is left in the database or in your head. Yes, you can use the ADOc Shape provider to extract data from several related tables and manage them in related ADOc-managed (Shape provider–managed) Recordsets. Anyone familiar with the Shape provider will feel comfortable with ADO.NET's DataSet approach. I would characterize the DataSet as a combination of an ActiveX Data Source control,[16] due to its ability to bind data with controls; a multidimensional Recordset, due to its ability to manage several resultsets (DataTable objects) at one time; and the Data Environment Designer or Data Object wizard, in that the DataSet can manage several Command objects used to manage the `SELECT` and action queries.

In contrast to the ADO Recordset, the ADO.NET System.Data.DataSet object is an in-memory data store that can manage *multiple* resultsets, each exposed as separate DataTable objects. Each DataTable contains data from a single data source—a single data query. No, the DataTable objects do not have to contain entire database tables—as you know, that simply won't work for larger databases (or for smaller ones either if you ever expect to upscale). I suggest you code your queries to contain a parameter-driven subset of rows that draw their data from one or more related tables.

Each DataTable object contains a DataColumnCollection (Columns)—a collection of DataColumn objects—that reflects or determines the schema of each DataTable; and a DataRowCollection (Rows) that contains the row data. This is a radical departure from DAO, RDO, and ADOc, where the data and schema information are encapsulated in the same Recordset (or Resultset) object. Consider, however, that the data in the DataTable is managed in XML and the schema in XSD. I discuss and illustrate this layout in Chapter 2.

---

15. http://www.msdn.microsoft.com/library/en-us/cpref/html/cpref_start.asp
16. The ADO Data Control, the Jet Data Control, and your hard-coded data source controls fall into this category.

You can construct your own DataTable objects by query or by code—defining each DataColumn object one-by-one and appending them to the DataColumn-Collection, just as you appended Field objects to an unopened Recordset in ADOc. The DataType property determines or reflects the type of data held by the DataColumn. The ReadOnly and AllowNull properties help to ensure data integrity, just as the Expression property enables you to build columns based on computed expressions. The DataSet is designed to be data agnostic—not caring where or how (or if) the data is sourced or retrieved; it leaves all of the data I/O responsibilities up to the .NET Data Provider.

In cases where your DataSet contains *related* resultsets, ADO.NET can manage these relationships for you—assuming you add code to define the relationships. For example, in the Biblio (or Pubs) database, the Authors table is related to the TitleAuthor and Titles tables. When you build a DataSet against resultsets based on these base (and many-to-many relationship) tables, and you construct the appropriate DataRelation objects; at that point you can navigate between authors and the titles they have written—all under control of ADO.NET. I illustrate and explain this in detail in Chapters 4 and 8.

DataTable objects can manage resultsets drawn directly from base tables or subset queries executed against base tables. The PK/FK relationships between the DataTable objects are managed through the DataRelation object—stored in the DataRelationCollection (Relations) collection. (Is there an echo in here?) When you construct these relationships (and you must—ADO.NET won't do it on its own; but, you can get the Visual Studio IDE to do it for you), UniqueConstraint and ForeignKeyConstraint objects are both automatically created depending on the parameter settings for the constructor. The UniqueConstraint ensures that values contained in a DataColumn are unique. The ForeignKeyConstraint determines what action is taken when a PK value is changed or deleted. I touch on these details again in Chapter 8. No, ADO.NET and the .NET IDE do not provide any mechanisms to construct these PK/FK relationships for you, despite supporting functionality to graphically define these relationships.

The following diagram (Figure 1-2) provides a simplified view of how the DataSet object is populated from a SqlClient .NET Data Provider. It illustrates the role of the bindable DataSet object and the important role of the .NET Data Provider. In this case, the diagram shows use of the Microsoft SQL Server–specific SqlClient .NET Data Provider, which contains objects to connect to the data source (SqlConnection), query the data (SqlDataAdapter), and retrieve a data stream (DataReader). The DataSet object's DataTable objects (Tables) are populated by a single call to the DataSet Fill method.

The DataAdapter also plays a key role here. It contains from one to four Command objects to (at least) fetch the data (SelectCommand) and (optionally) change it (UpdateCommand, InsertCommand, and DeleteCommand). Each of these Command objects are tied to specific Connection objects. When you execute

the DataSet.Update method, the associated DataAdapter executes the appropriate DataAdapter Command objects for each added, changed, or deleted row in each of the DataTable objects.

Once constructed, the DataSet need not remain connected to the data source because all data is persisted locally in memory—changes and all. I drill deeper into DataSet topics in Chapter 4.

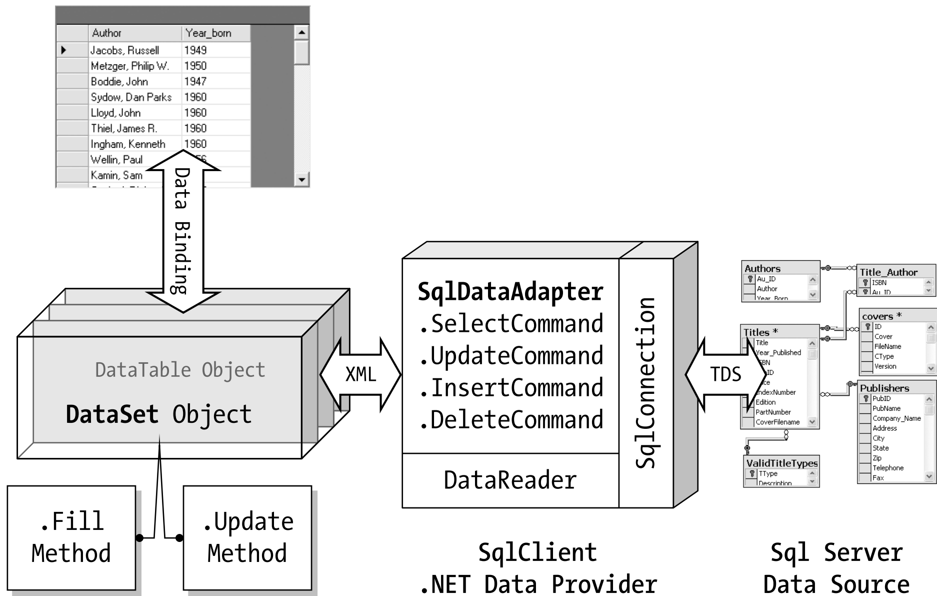

Figure 1-2. ADO.NET Data Access using the DataSet object.

The DataSet object supports a DataTableCollection (Tables) collection of DataTable objects, which contain a DataRowCollection (Rows) collection of DataRow objects. Each DataRow object contains the DataColumnCollection (Columns) of DataColumn objects, which contain the data and all of the DDL properties. Remember that, like the ADOc Recordset, the DataTable object can be bound by assigning it to the DataSource property of data-aware (bindable) controls.

Figure 1-3 illustrates the look of the System.Data.DataSet in a hierarchical diagram. Note the difference in the .NET naming convention. In COM, we expect a collection of objects to be named using the plural form of the object. For example, a collection of Cat objects would be stored in the Cats collection. In .NET, most (but not all) collections are named using the singular object name followed by "Collection," as in DataTableCollection. I found this very confusing until I started to code. It did not take long to discover that ADO.NET uses *different* names for

each of these collections. These "real" names are shown in parentheses in the preceding paragraph and in Figure 1-3. I'm sure there's a good OO reason for this—I just have no idea what it is.

I explore each of these objects in more detail in subsequent chapters.

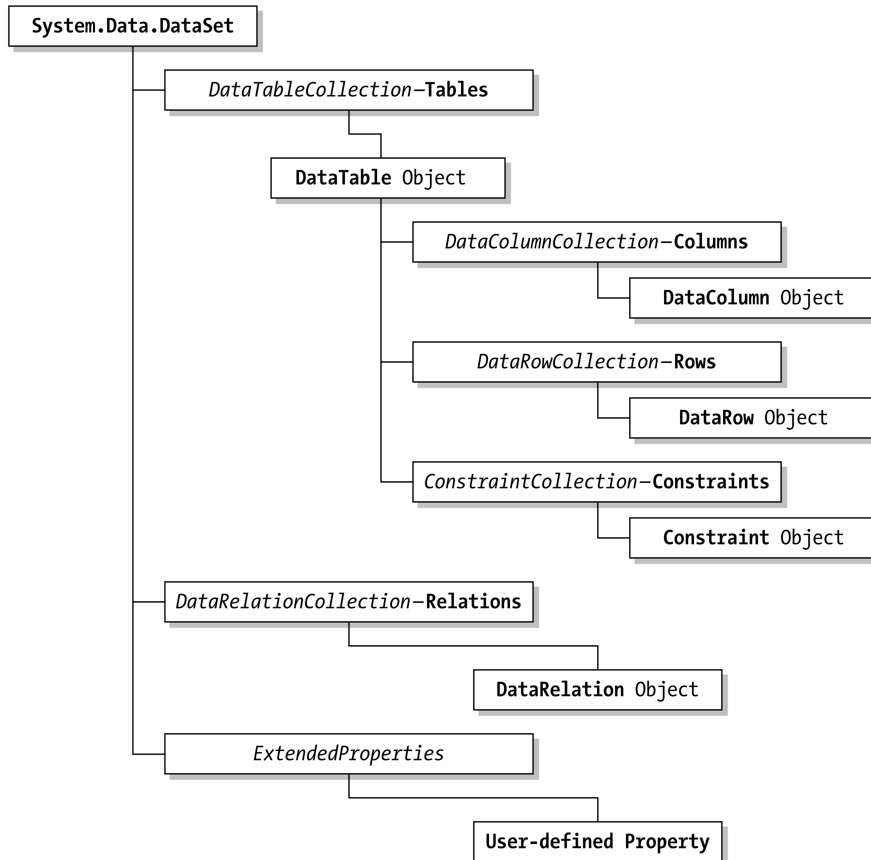

Figure 1-3. DataSet object hierarchy.

So, what should you know about this new ADO.NET structure? The DataSet:

- **Is a memory-resident structure constructed by the DataAdapter Fill method.**
- **Contains zero or more DataTable objects.**
- **Is logically tied to a DataAdapter object used to fetch and perform action queries as needed.**

- **Contains Constraints and Relations collections to manage inter-DataTable relationships.**
- **Is data-source agnostic, stateless, and can function independently from the data source.** All data, schema, constraints, and relationships to other tables in the DataSet are contained therein.
- **Is transported through XML documents via HTTP.** This means a DataSet can be passed through firewalls and used by any application capable of dealing with XML.
- **Can be saved to XML or constructed from properly formatted XML.**
- **Can be created programmatically**. DataTable by DataTable and DataColumn by DataColumn—along with DataRelation objects and Constraints.

It's clear that the DataSet was designed to transport "smart" data (and schema) between a Web host (possibly implemented as a Web Service) and a client. In this scenario, a client application queries the Web Service for specific information, such as the number of available rooms in hotels given a specific city. The Web Service queries the database using parameters passed from the client application and constructs a DataSet, which might contain a single DataTable object or multiple DataTable objects. If more than one table is returned, the DataSet can also specify the relationships between the tables to permit the client to navigate the room selections from city to city. The client can display and modify the data—possibly selecting one or more rooms—and pass back the DataSet to the Web Service, which uses a DataAdapter to reconcile the changes with the existing database.

Descending the System.Data Namespace Tree

I think pictures and drawings often make a subject easier to understand—especially for subjects like object hierarchies. So, I'm going to begin this section with a series of diagrams that illustrate the layout of the System.Data namespace.

ADOc has a relatively easy-to-understand and easily diagrammed object hierarchy. ADO.NET's System.Data namespace, however, is far more complex. As it currently stands, there are dozens upon dozens[17] of classes and members in the .NET Framework. Few of the complexities of the OO interfaces have been hidden—at least not in the documentation. Fortunately, there is a fairly easy way to climb through the object trees and get a good visual understanding of the hierarchies—basically what goes where and with what: Use the object browser in Visual Studio .NET. You can launch it from the **View | Other Windows** submenu. Figure 1-4 illustrates how the object browser depicts the System.Data namespace (unexploded). Throughout this section of the book, I walk through these object trees one at a time. By the time I'm done, you should either be thoroughly familiar with the System.Data namespace or thoroughly sick of it.

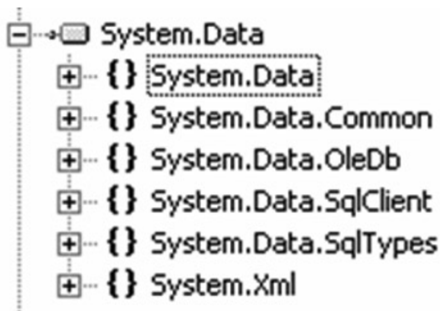

Figure 1-4. The System.Data namespace.

System.Data Namespace Exploded

The exploded System.Data namespace has over forty members—the top dozen or so are shown in Figure 1-5. I hope that we won't have to learn and remember how to use *all* of these objects, properties, methods, and events to become productive ADO.NET developers. Table 1-1 lists and describes the most important of these objects—the ones you'll use most often (at least at first).

17. I tried to count all of the objects in System.Data but lost count … sorry.

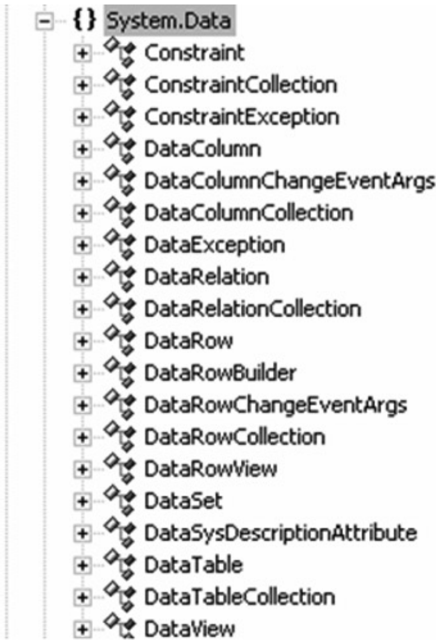

Figure 1-5. System.Data objects.

Table 1-1. Selected Members of the System.Data Namespace

Object	Description
Constraint and ConstraintCollection (Constraints), ForeignKeyConstraint, UniqueConstraint	Represents referential integrity constraints. Used to specify unique keys or PK/FK constraints and what to do when they change. Used to prevent duplicate rows from being added to the current dataset. No equivalent in ADOc. Hard coded by your application.
DataColumn and DataColumnCollection (Columns)	Represents a single data column schema associated with a DataTable object and the collection used to manage the columns. Similar to the ADOc Field object and Fields collection—but without the Value property. Automatically generated from the resultset.
DataException (and various other exception objects)	Represents the various exceptions thrown when an ADO.NET error is triggered. Similar to the ADOc Error object.

(continued)

Table 1-1. Selected Members of the System.Data Namespace (continued)

Object	**Description**
DataRelation and DataRelationCollection (Relations)	Represents table/column/table relations. Hard coded by your application. Specifies the tables and columns used to interrelate parent/child tables. No equivalent in ADOc.
DataRow and DataRowCollection (Rows)	Represents the *data* in a table row. Generated automatically.
DataRowView	Permits customized views of data rows based on changes applied during editing. Original, Proposed, and Current versions of a data row are exposed.
DataSet	Represents an in-memory data store consisting of DataTable, DataRelation, and Constraint objects.
DescriptionAttribute	Permits definition of code-specified properties for properties, events, or extenders.
DataTable and DataTableCollection (Tables)	Represents in-memory rows and columns of data returned from a data source or generated in code.
DataView, DataViewManager, DataViewSetting, DataViewSettingCollection (DataViewSettings)	Permits viewing one or more subsets of a DataTable. Similar to ADOc Recordsets after the Filter property is applied. Several DataView objects can be created against the same DataTable.
PropertyCollection (Properties)	Permits definition and retrieval of code-defined properties.
(and several others)	There are several other objects, event enumerations, and support objects exposed by the System.Data namespace.

Instantiating System.Data Objects

Your .NET application should be fairly specific about the libraries it expects to reference. In .NET, the ADO.NET .NET Data Providers (roughly equivalent to the ODBC and OLE DB providers accessed by ADOc) are built into the System.Data namespace so you don't have to add an explicit reference to use them. An exception is the Odbc .NET Data Provider that must be installed and registered separately—and so it is not part of System.Data namespace, rather it's part of the Microsoft.Data namespace. The Solution Explorer is a handy way to see what namespaces are already referenced for your application's assembly, as shown in Figure 1-6.

Figure 1-6. The Solution Explorer showing a newly created WinForm application.

Depending on the ADO.NET data access provider you choose, you'll want to use the using[18] directive with either System.Data.OleDb or System.Data.SqlClient (or in unusual situations, both), or Microsoft.Data.Odbc to make sure your code correctly references these libraries. Actually, the CLR, which sits at the core of .NET, won't permit name collisions, but adding a namespace to the using list makes coding easier by providing "shorthand" syntax for commonly used objects. Although not required for ADO.NET, the using directive signals the compiler to

---

18. For Visual Basic developers converting from Visual Basic .NET, the using directive is equivalent to the `Imports` directive in Visual Basic .NET and (very) loosely similar to the inclusion of header files in C and C++.

search the specified namespace referenced in your code to resolve any ambiguous object names. Basically, using helps the compiler resolve namespace references more easily. The using statement should be positioned first in your code—above all other declarations. For example, to add the OleDb .NET Data Provider namespace, place the following at the start of your code module:

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

Similarly for the SqlClient .NET Data Provider namespace, add the following to your code module:

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

Because you used the `using` directive with the System.Data.SqlClient .NET Data Provider, you can code

```
SqlConnection cn = new SqlConnection();
```

However, the downside to this approach is potential object collisions and failed compiles. Again, some pundits feel that it's best to explicitly reference declared objects. You can also reference your ADO.NET objects explicitly if you don't mind typing a lot (or if you are paid by the word). For example, you can create a new ADO.NET Connection object this way:

```
System.Data.SqlClient.SqlConnection cn =
            new System.Data.SqlClient.SqlConnection();
```

However, I try not to use this approach in my examples or sample code. I provide more examples of object and variable declarations as I go—and there is a long way yet to travel.

Introducing the ADO.NET DataAdapter

Think of the DataAdapter as a "bridge" object that links the data source (your database) and a Connection object with the ADO.NET-managed DataSet object through its `SELECT` and action query Commands. All of the .NET Data Providers implement their version (instance) of the System.Data.DataAdapter class; OleDbDataAdapter, OdbcDataAdapter, and SqlClientDataAdapter all inherit from the base System.Data class. Each .NET Data Provider exposes a SelectCommand property that contains a query that returns rows when the DataSet Fill method is executed. The SelectCommand is typically a `SELECT` query or the name of a stored

procedure. Each Command object managed by the DataAdapter references a Connection[19] object to manage the database connection through the Command object's Connection property. I discuss the Connection object in Chapter 2.

The invocation of the DataSet Update method triggers the execution of the DataAdapter object's UpdateCommand, InsertCommand, or DeleteCommand to post changes made to the DataSet object. I discuss updating in Chapter 7. The figure shown earlier (Figure 1-2) also illustrates the working relationship between the DataSet and the DataAdapter.

Constructing DataAdapter Command Queries

If the query set in the SelectCommand is simple enough (references a single table and not a stored procedure), you can (usually) ask ADO.NET to generate the appropriate action queries for the DataAdapter UpdateCommand, InsertCommand, and DeleteCommand using the CommandBuilder object. If this does not produce suitable SQL syntax, you can manually fill in the action queries using queries of your own design—even calling stored procedures to perform the operations. I discuss the construction of these commands in Chapter 3.

Coding the DataAdapter

I expect you'd like to see some code that demonstrates how all of this is implemented. Because I haven't discussed the Connection object yet, this will be a little tough, but let's assume for a minute that you know how to get connected in ADO.NET. Let me walk you through a small example.[20] (Don't worry about the code I don't explain here—I discuss many of these points again in the next chapter.)

First, make sure that your application can see the SqlClient namespace. It's already part of the .NET Framework, but not part of your application's namespace.

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

Next, within the address range of your Form's[21] class, define the objects and variables to be used.

---

19. Actually, the name of the Connection object is SqlConnection, OleDbConnection, OdbcConnection, or <ProviderSpecific>Connection in the case of other vendors' .NET Data Provider namespace's Connection object.
20. Located in the "\Examples\Chapter 01\Data Adapter" folder on the CD.
21. The default architecture in most examples (before I get to Chapter 10) is Windows Forms.

NOTE *In C#, objects and variables have private scope by default. I will have a little more to say on constructors later on—like where and how best to deal with them—but those familiar with VB .NET should just note here that* `strConnect` *is declared here as a* `const`*. This means that it is effectively a read-only field—a constant. Why? Well, we use this string in the constructor argument for the new SqlConnection object. So what? Well in C#, non-static instance fields cannot be used to initialize other instance fields outside of a method, and this is quite different from VB .NET.*

```
public class Form1 : System.Windows.Forms.Form
{
    const string strConnect = "data source=.;database=biblio;uid=admin;pwd=pw";
    string strQuery   =
        "Select Title, Price from Titles where Title like 'Hit%'";
    SqlConnection cn = new SqlConnection(strConnect);
    SqlDataAdapter da = new SqlDataAdapter();
    DataSet ds = new DataSet();
……
}
```

In the Form1_Load event handler, you set the DataAdapter object's SelectCommand string to a `SELECT` query that returns a few rows from the Titles table. Actually, you shouldn't have to open the connection explicitly, because, if the connection is not already open, the Fill method automatically opens it and then closes it again. If you use this auto-open technique, you need to be prepared for connection errors when you execute the Command. I'm using this approach because it's more familiar to ADOc developers. I illustrate how to get the Fill method to manage connections in the next chapter and a simpler, more ADO.NET-centric approach later in this chapter.

Notice the use of C#'s `try` and `catch` error handler.[22] In the `catch` statement, you reference the System.Data.SqlClient.SqlException object simply as `SqlException` (remember that you placed the `using System.Data.SqlClient;` statement in earlier so that you could make these "shorthand" references). `SqlException` exposes a Message and Error number (and more) that can be used to figure out what went wrong. The simplest way to provide **all** of the `SqlException` object information to a developer during debugging is to cast it to a string with a call to the `ToString()` method, sending this to the Debug output window via the

22. Error handling is discussed in Chapter 9, "ADO.NET Error Management Strategies." The ADO.NET concepts I use apply universally in most cases.

Debug.WriteLine() method. To use the Debug object, you must reference the System.Diagnostics namespace by using System.Diagnostics. This is helpful when the intended recipient of an exception message is a developer, but not necessarily so useful for your program to spew it all out to a user while committing hari-kari. Fortunately, the Debug object is automatically stripped from release builds. Your user doesn't care much for which line in your code triggered the self-disembowelment—but more on exceptions in Chapter 9.

So here, if the cn.Open(); statement does not work, the next statement is never executed and the catch block will deal with the exception, depositing its remains to the console/debug output window.

```
...
using System.Diagnostics;
...
private void Form1_Load(object sender, System.EventArgs e)
{
    try
    {
        cn.Open();
        da.SelectCommand = new SqlCommand(strQuery, cn);
    }
    catch(SqlException ex)
    {
        Debug.WriteLine(ex.ToString());
    }
}
```

In the Button click-event (did I say there are both DataGrid and Button controls on the form?) you use the DataAdapter Fill method to "run" the SelectCommand query in the specified DataAdapter. The results are fed to the DataSet object. By default, the Fill method names the DataSet "Table" (for some reason). I would have preferred "Data" or "DataSet" to discourage confusion with database tables. The Fill method is very (very) flexible as it can be invoked in a bevy of ways, as I describe in Chapter 4. The options I chose in this example name the resulting DataTable "TitlesandPrice." In the next statement, I bind the DataSet to the DataGrid control.

```
private void Button1_Click(object sender, System.EventArgs e)
{
        da.Fill(ds, "Titles and Price"); // Defaults to "Table"
        DataGrid1.DataSource = ds.Tables["Titles and Price"];
}
```

The result? Well, this code opens a connection, runs a query, and fills a grid with the resulting rows; but what's missing? To start with: error handlers. This code does not deal with bad connections (except to print a debug message), bad queries, empty queries, or the fact that most applications will want to create a parameter-based query instead of a hard-coded SELECT statement. However, baby steps come before running—especially in *this* neighborhood.

As I wrote this example, I was reminded of a few lessons:

- **The** `using System.Data.SqlClient` **directive helps.** Statement completion did not show the objects I was referencing nearly as quickly (if at all) until I added the `using` directive.
- **The DataSet object is suitable for binding.** That is, it can be assigned to the DataGrid or any bindable control for display. In my example, I bind the DataSet to a DataGrid control's DataSource property.
- **It helps to bind to a specific DataTable.** If you bind to the DataSet, the data in the DataGrid isn't immediately shown. This requires the user to drill down into a selected DataTable. It's better to bind to a specific DataTable in the DataSet Tables collection.
- **Use the form's constructor method to initialize instance variables**. It is not a good idea to initialize instance variables at class level declaration since they can't be encapsulated in `try/catch` blocks to deal with any exceptions arising in the initialization.

TIP *This is a practice I picked up years ago: Install crude error handlers from the very beginning. I encourage you to do the same. The crudest, of course, is a simple catch and casting of the exception object to a string that is sent to the debug output window. This can save you an extra ten minutes as you try to figure out what went wrong.*

A Simpler Example

Okay, now that I have shown you an example based on how an ADOc developer might code, take a look at the same problem using the new ADO.NET approach. You should notice that there is no explicit call to open the connection—that is taken care of here silently by the `Fill()` method; I talk more about this later.

Also, I am specifying properties for the class' constructors to use when instantiating. Most .NET classes have one or several constructors used to set different combinations of properties as the objects are being instantiated. After you understand these, you'll really appreciate how they make your code easier to write.

```
private void btnRunQuery_Click(object sender, System.EventArgs e)
{
    try
    {
        SqlConnection cn =
            new SqlConnection("data source=.;database=biblio;uid=admin;pwd=pw");
        SqlDataAdapter da =
            new SqlDataAdapter(
            "Select Title, Price from Titles where Title like 'Hit%'", cn);
        DataSet ds = new DataSet();
        da.Fill(ds, "Titles and Price");
        DataGrid1.DataSource = ds.Tables["Titles and Price"];
    }
    catch (SqlException ex)
    {
        Debug.WriteLine(ex.ToString());
    }
}
```

ADO.NET's Low-Level Data Stream

By this time you know that by default, ADOc Recordsets are created as RO/FO[23] firehose data structures. This low-level data stream permits data providers to return resultsets to the client as quickly as the LAN can carry them. While fast, the default firehose ADOc Recordset does not support record count, cursors, scrolling, updatability, caching, filters, sorting, or any costly overhead mechanism that could slow down the process of getting data back to the client.

ADO.NET also supports this firehose functionality, but in a different way. After you establish a connection, you can stream data back to your application using the ADO.NET .NET Data Provider's DataReader class (SqlDataReader, OdbcDataReader, or OleDbDataReader) through the provider's Command class (SqlCommand, OleDbCommand, or OdbcCommand).

23. RO/FO: read-only/forward-only

Although the ADO.NET data stream is RO/FO, the fundamental data access technique is different from ADOc in a number of respects. Let's walk through a simple example[24] as an illustration. The following section of code declares Connection, DataAdapter, DataReader, and Command objects using the SqlClient .NET Data Provider. As you'll see throughout this section, C# permits you to declare and initialize selected properties of the declared objects in a single line of code; although, as I said earlier, departing from VB .NET, C# does not permit you to initialize selected properties with other non-static instance fields outside of a method.

If you look carefully at this code snippet you'll notice that the `SqlCommand cmd` object is declared in the Form1 class outside of a method. By default this gives the `cmd` object private scope, ensuring that it is accessible to all methods within the class. I have, however, placed the constructor code

```
cmd = new SqlCommand(strSQL, cn);
```

for the `cmd` object within Form1's constructor method `Form1()` after the call to `InitializeComponent()`; since this depends on other non-static instance fields `strSQL` and `cn`.[25]

```
…
using System.Data.SqlClient;
…
public class Form1 : System.Windows.Forms.Form
{
   …
    SqlConnection cn =
        new SqlConnection("data source=.;database=biblio;uid=admin;pwd=pw");
    SqlDataAdapter da = new SqlDataAdapter();
    string strSQL = "Select Title, PubID from Titles where Title like ";
    SqlDataReader Dr;
    SqlCommand cmd;
   …
public Form1()
{
   …
    InitializeComponent();
   …
    cmd = new SqlCommand(strSQL, cn);
}
   …
```

24. Located in the "\Examples\Chapter 01\Data Stream" folder on the CD.
25. Well I suppose we could declare strSQL and cn as static but then that would force them to be common across **all** concurrent instances of Form1.

This next routine is fired when a button is clicked on the form. The ExecuteReader method is executed—instantiating a SqlClient.DataReader. When first opened, the DataReader does *not* expose a row of the resultset because its current row pointer is positioned before any rows (as in a Recordset when BOF = True). To activate the first and each subsequent row one at a time, you have to use the DataReader object's Read method, which returns False when there are no (additional) rows available. Once read, you can't scroll back to previously read rows—just as in the FO resultset in ADOc.

As each row is read, the code moves data from the columns exposed by the DataReader to a ListBox control. Note that you have to use the `Add()` method to add members to any collection—including the ListBox and ComboBox controls' Items collections. You also have to be very careful about moving the data out of the DataReader columns; each column must be specifically cast as you go—converting each to a datatype suitable for the target. In order to code these conversions correctly, your code will have to know what datatypes are being returned by the resultset or use the GetValue method. .NET is pretty unforgiving when it comes to automatically morphing datatypes.

TIP *I use the ListBox BeginUpdate and EndUpdate methods to prevent needless painting while I'm filling it.*

```
private void Button1_Click(object sender, System.EventArgs e)
{
    cn.Open(); //The connect string was defined when the object was created
    cmd.CommandText = strSQL + "'" + TextBox1.Text + "%'";
    Dr = cmd.ExecuteReader();
    ListBox1.Items.Clear();  // clear the listbox
    ListBox1.BeginUpdate();  // Prevent the listbox from painting

    while (Dr.Read()) // get the first (or next) row
    {
        ListBox1.Items.Add(Dr.GetString(0) + " - " + Dr.GetInt32(1).ToString());
    }

    ListBox1.EndUpdate();  // Let the listbox paint again

    Dr.Close();   // close the data reader.
}
```

Index

Symbols

? parameter marker
  lack of support for by SqlClient .NET Data Provider, 85–86
  support for in ADO.NET, 82–83
@ReturnValue parameter, creating and adding to stored procedures, 125–126
+ operator, use of in .NET, 91

A

AcceptRejectRule, ForeignKeyConstraint, 290
action queries, using Visual Studio to generate, 251–258
Add() method, using to construct parameters, 115
Add Web Reference dialog, opening, 333–334
ad hoc queries, executing to perform updates, 280
ADOc
  comparing to ADO.NET, 7–9, 142–144
  update strategies, 238–239
ADOc code
  instantiation of objects when importing, 18
  leveraging existing in your .NET executables, 17–18
ADOc connection objects
  accessing from Visual Basic .NET, 77
  establishing a connection with, 39
ADOc ConnectionString vs. ADO.NET ConnectionString, 46
ADOc Field object, compared with DataSet, 172
ADOc Fields collection vs. the ADO.NET DataColumnCollection, 171
ADOc namespace, instantiating, 76–77
ADOc objects, accessing from .NET executables, 18
ADOc properties vs. ADO.NET properties, 182–183
ADOc Recordset
  vs. the ADO.NET DataReader processing loop, 146
  vs. ADO.NET DataTable structure, 170
  vs. the DataTable, 157–196
  importing into an ADO.NET data structure, 168–172
ADOc Recordset objects, creating DataSets from, 19
ADOc Recordsets, comparing to DataTable objects, 169–172
ADO.NET
  adding new DataRows to a DataTable, 224–225
  vs. ADOc data stream, 36–38
  capturing the SELECT-generated rowsets in, 129–132
  command strategies, 79–140
  comparing to ADOc, 7–9, 142–144
  Connection object properties, 53–54
  connection pooling, 71–74
  data access using the DataSet object, 24
  DataReader strategies, 141–156
  DataSet object hierarchy, 159
  and disconnected data structures, 20–21
  error management strategies, 301–311
  establishing connections using different providers, 39–77
  forms of Find implemented by, 213
  the fundamentals, 12–15
  getting to manage a pessimistic locking update, 222
  handling of duplicate DataTable columns by, 181
  how it implements constraints, 285–295
  how it passes data versions to update parameters, 264
  how we got here, 2–6
  implementing foreign key constraints, 289–291
  implementing unique constraints, 287–289
  introduction to, 1–38
  low-level data stream, 36–38
  .NET Data Providers, 15–20
  a new beginning, 7–9
  vs. other Microsoft data access interfaces, 10–11
  possible problems with, 14–15
  reviewing the generated stored procedures, 251–254
  and SOAP, 342
  support of Parameter constructors, 112–114

understanding its distributed architecture, 10–11
understanding the infrastructure, 9–15
update strategies, 221–282
updating a DataView in, 210
using "O'Malley" rule with, 199
using the CommandBuilder class, 240–250
using to execute ad hoc queries, 84–85
and XML, 313–342
XML support in, 314–315
ADO.NET and ADO Examples and Best Practices for VB Programmers, 2nd Edition, discussion of Da Vinci Tools in, 81
ADO.NET classes, a typical implementation of, 13–15
ADO.NET CommandBuilder
how the Update() method manages Command objects generated by, 242
using to construct commands, 81
ADO.NET commands, understanding, 79–136
ADO.NET Connection objects, creating, 42–51
ADO.NET connections
closing, 68–75
opening in code, 65–66
ADO.NET ConnectionString vs. ADOc ConnectionString, 46
ADO.NET ConnectionString property, building, 44–51
ADO.NET DataAdapter, introducing, 31–36
ADO.NET DataAdapter commands, 80
ADO.NET DataColumnCollection vs. the ADOc Fields collection, 171
ADO.NET DataReader vs. the ADOc Recordset processing loop, 146
ADO.NET DataSet
object hierarchy, 159
using with the DataTable, 157–196
ADO.NET DataSet object, 22–26
ADO.NET data structure, importing an ADOc Recordset into, 168–172
ADO.NET DataTable structure vs. ADOc Recordset, 170
ADO.NET garbage collector, managing, 68–69
ADO.NET .NET Data Provider, choosing the right one, 41–42
ADO.NET .NET Data Providers. *See also* .NET Data Provider, 40–42
use of XML by, 41
valid "name" arguments and how they are used, 48–50
ADO.NET .NET Data Providers CommandBuilder vs. Visual Studio's, 253–254
ADO.NET parameter queries, introduction to, 82–84
ADO.NET properties vs. ADOc properties, 182–183
Advanced SQL Generations Options dialog, using to generate code for UpdateCommand, 90–91
Application Name, Connection string argument, 48
arguments
passing multiple to the DataView object Find() method, 218
passing to the DataView object Find() method, 217–218
ASP.NET security model, function of in connection security, 70–71
AttachDBFilename, Connection string argument, 48
Authors table, example of updating, 266–272
autonumber, identity, or GUIDs, retrieving, 258–262

B

"before" block, for DiffGram, 320
BeginEdit() method, using, 228
BeginTransaction() method
Connection object, 55
creating a Transaction object with, 59
Biblio database
copying from the CD, 343–344
and its intertable relationships, 296
BiblioServiceException class, for Web Service, 328
binding, to DataSets, 195–196
Boolean properties, setting using yes and no instead of true or false, 50
bulk INSERT statement, using to improve update performance, 264–265

C

C#
development of, 3
introduction to using ADO.NET with, 1–38
using constructors in declarations in, 43–44
CancelEdit() method, undoing changes to DataRow or DataRowView objects with, 227
Cascade action, understanding, 291
case sensitivity, of DataSets, 186
Caspol.exe, modifying security policy with, 62

ChangeDatabase() method, Connection object, 55
ChangeDatabase() method, using, 46–47
ChangeState event, fired on the Connection object, 56
check constraints, how ADO.NET implements them, 285
Class View, of generated DataSet, 100–101
CloseConnection, ExecuteReader CommandBehavior argument, 134
Close() method
  Connection object, 55
  for DataReader, 148
CLS. *See* Common Language Specification (CLS)
Code Access Security Policy Tool (Caspol.exe), modifying security policy with, 62
Columns collection
  accessing, 182–186
  creating your own, 186
COM-based ADO, as a .NET Data Provider, 75–77
COM Interop, how it affects .NET applications, 19–20
CommandBuilder
  in ADO.NET, 81
  constructing the Parameters collection in, 248–249
  generating the DeleteCommand with, 245
  generating the InsertCommand with, 243–244
  generating the UpdateCommand with, 244–245
  how it deals with concurrency issues, 249–250
  how the Update() method manages Command objects generated by, 242
CommandBuilder class (cb)
  initiating, 243–248
  rules and restrictions, 240–241
  using, 240–250
Command constructor, code for in ADO.NET, 80
Command Execute() DataReader methods, exploring, 133–134
Command object, in ADO.NET, 12
Common Language Specification (CLS), 183
ComputeRowCounts subroutine, for showing row counts, 270–272
Connection Lifetime (SqlClient)
  Connection string argument, 48
  ConnectionString keyword, 73
Connection object
  in ADO.NET, 12
  closing, 128–129
  constructing a connection string for opening, 65–66
  importance of explicitly closing, 62
Connection object events, sinking, 56–58
Connection object methods, examining, 55
Connection object properties, ADO.NET, 53–54
Connection objects
  ADO.NET vs. ADOc, 42–43
  creating ADO.NET, 42–51
Connection.Open() method, acceptance of arguments by, 44–45
connection pool
  cleaning, 72
  debugging, 75
  monitoring with performance counters, 74
connection pooling
  ADO.NET, 71–74
  OleDb and Odbc, 75
  SqlClient .NET Data Provider, 71–72
Connection Reset
  Connection string argument, 48
  keyword for SqlClient.SqlConnection object, 73
connections
  adding to your project, 66–68
  information and security issues, 69–71
ConnectionString, constructing and assigning to a SqlConnection object, 107
Connection string arguments, list of valid and their uses, 48–50
ConnectionString keywords, for SqlClient.SqlConnection object, 73–74
ConnectionString property
  ADO.NET, 54
  building an ADO.NET, 44–51
Connection Timeout, Connection string argument, 48
ConnectionTimeout property, ADO.NET, 54
Connect Timeout, Connection string argument, 48
ConstraintCollection collection, managing, 286
ConstraintException, reason thrown, 306
constraints
  how ADO.NET implements, 285–295
  postponing enforcement of, 228
  types of, 285–286

constraints collection, managing, 286
constructor methods, using in declarations, 43–44
ContinueAfterError property, using, 274–275
ContinueUpdateOnError property, for Update statements, 273–276
CreateCommand() method, Connection object, 55
CTRL+ALT+5, opening the Server Explorer window with, 66–67
Current Language, Connection string argument, 48
current row pointer (CRP), positioning, 195

D

DACW. *See* DataAdapter Configuration Wizard (DACW)
data, merging and transporting, 281–282
DataAdapter
- coding, 32–36
- Command objects contained in, 23
- constructing command queries, 32
- introduction to in ADO.NET, 31–36
- lessons learned when coding, 35
- properties for managing operational queries, 239

DataAdapter Configuration Wizard (DACW)
- changing the DataAdapter created by, 92
- generating stored procedures with, 93–95
- information needed to construct a DataAdapter, 87–88
- renaming the DataAdapter or Command objects created by, 97
- reviewing the code generated by, 254–258
- reviewing the stored procedures generated by, 251–254
- Update mapping of parameter to source column, 97
- using, 86–98
- using existing stored procedures with, 95–98
- using SQL statements with, 89–93
- using stored procedures with, 93
- using to set up a typical query, 87–88

DataAdapter Fill() method, using to build DataTables, 173–177
DataAdapter object, in ADO.NET, 12
DataAdapter Update() methods, for SqlClient .NET Data Provider, 263
Database (SqlClient only), Connection string argument, 48
database connections, importance of closing as soon as you can, 69
Database property, ADO.NET, 54
data binding, using the DataView in, 205
DataColumnCollection (Columns), accessing, 182–186
DataColumn object
- public properties, 182–183
- supported DataType settings, 183–186

data constraints, postponing enforcement of, 228
DataException, reason thrown, 306
Data|Generate DataSet menu, 99
DataGrid control
- showing rows that failed to update, 273
- working with an updatable, 268

DataReader
- cleaning up after, 155–156
- common methods inherited from parent object, 148–149
- creating to stream in a rowset, 128–129
- FieldCount property, 147
- Get methods common to all providers, 149–150
- Get methods unique to SqlClient provider, 150–151
- implementation of by ADO.NET's data providers, 144–145
- importance of closing as soon as you can, 69
- important difference from the DataSet, 145
- methods for instantiating and executing a query, 145–146
- operational methods, 147–148
- RecordsAffected property, 147
- vs. the Recordset processing loop, 146
- understanding, 144–151
- using Get methods to retrieve data, 151–155
- using instead of a DataSet, 127–129

DataReader Get methods, using to retrieve data, 151–155
DataReader methods, exploring, 133–134
DataReader object, in ADO.NET, 12
DataReader properties, typical, 147
DataReader strategies, in ADO.NET, 141–156
DataRelationCollection, 161
DataRelation objects, creating, 295–300
DataRow, RowState values, 233
DataRow.AcceptChanges() method, effect on DataRowVersion, 237
DataRow.BeginEdit() method, effect on DataRowVersion, 237

DataRow.CancelEdit() method, effect on DataRowVersion, 237
DataRowCollection (Rows), accessing, 187–191
DataRow.EndEdit() method, effect on DataRowVersion, 237
DataRow errors, indicating, 193–194
DataRow Item, determining the data version for, 234–237
DataRow object
- BeginEdit() method, 228
- dumping an array of to a TextBox control, 202–203
- Item property, 188–190
- understanding, 187

DataRow properties, 187
DataRow.RejectChanges() method, effect on DataRowVersion, 237
DataRow RowError() method, using to set an error string, 193
DataRows
- adding new to a ADO.NET DataTable, 224–225
- undoing a delete operation, 227

DataRowVersion, impact of editing on, 237
DataRow versions, effect of data changes on, 237
DataSet
- Class View of generated, 100–101
- compared with ADOc Field object, 172
- creating a strongly typed, 102–105
- creating with the .NET IDE, 98–101
- important difference from the DataReader, 145
- object hierarchy, 159
- undoing a delete operation, 227
- using a DataReader instead of, 127–129
- using with the DataTable, 157–196

DataSet and DataTable structure, understanding, 159–168
DataSet Collections, 161
DataSet data, importing from an XML document, 316–319
DataSet Fill() method, extracting rows from the data source with, 106–107
DataSet Merge() method, merging DataSets with, 281–282
DataSet objects
- addressing data in, 163
- ADO.NET, 13, 22–26
- constructing directly from FOR XML queries, 82
- fetching and saving as XML, 321–323
- hierarchy, 25
- using for ADO.NET data access, 24

DataSet objects and XML, understanding, 316–325
DataSet rows, deleting from a DataTable, 226–227
DataSets
- accessing data in, 194–195
- addressing the Value property with typed, 224
- addressing the Value property with untyped, 223–224
- binding to, 195–196
- building DataTable objects within, 177–179
- case sensitivity of, 186
- constructing and saving an existing schema, 164
- constructing strongly typed, 163–165
- creating from ADOc Recordset objects, 19
- creating updatable, 239–240
- deleting a specific row from, 227
- filtering, sorting, and finding data in, 197–219
- generating using the XSD.exe command-line tool, 164–165
- implementing typed, 161
- manually assembling strongly typed, 165–167
- merging and transporting, 281–282
- merging XML data with, 321
- moving DataTable objects from one to another, 178
- passing updated back to a Web Service, 336–337
- typed vs. untyped, 161–168
- using untyped, 167–168

DataSet XML methods, 322
DataSource property, ADO.NET, 54
Data Source, Server, Addr, Address, or Network Address, Connection string argument, 48
DataTable
- adding new DataRows to in ADO.NET, 224–225
- vs. the ADOc Recordset, 157–196
- creating with its associated Rows collection, 189–190
- using with the DataSet, 157–196

DataTable.AcceptChanges() method, effect on DataRowVersion, 237
DataTableCollection, 161
DataTable.Constraint, how it is used, 286

DataTable objects
in ADO.NET, 13
building within a DataSet, 177–179
comparing to ADOc Recordsets, 169–172
contents of, 22–23
moving from one DataSet to another, 178
DataTableRow_Changed event handler, coding example, 230–232
DataTables
accessing data in, 181–194
building, 172–181
building with the constructor, 173
creating from multiple resultsets, 178–179
the fundamentals of changing data in, 223–238
populating a DataSet with several at once, 178–179
undoing a delete operation, 227
using the DataAdapter Fill() method to build, 173–177
DataTable Select() method
additional DataView advantages over, 205
example code for using to filter and sort, 199–203
filtering and sorting with, 198–204
filtering, sorting, and finding data with, 197–219
sorting with, 199–203
DataTable.Update, effect on DataRowVersion, 237
DataType properties, referencing in your code, 185–186
DataType settings, supported by DataColumn object, 183–186
data validation
building into the DataSet, 229
coding event handlers, 229–233
data versions
code for showing the available, 269–270
determining, 234–237
DataView
advantages vs. DataTable Select() method, 205
sorting, 206–207
updating, 210
DataView filter properties, setting, 206–210
DataViewManager object
ADO.NET, 204–205
working with, 211–212
DataView object
adding to a form or component, 205
filtering and sorting with, 204–210
Find() method, 216–218
using the sort(), filter(), and find() methods, 197–219
DataView object Find() method
passing arguments to, 217–218
passing multiple arguments to, 218
DataView objects, creating with Visual Studio, 210
Da Vinci Tools, new version available in the .NET IDE, 81
DBConcurrencyException, reason thrown, 306
DBConcurrencyException trap, coding, 279–280
DDL query, example of, 242–243
declarations, using constructor methods in, 43–44
DefaultView, example code for using, 207–210
DefaultViewManager property, creating custom setting for DataTables with, 212
DeleteCommand
generated by the DACW, 257–258
generating with the CommandBuilder, 245
DeleteCommand object, ADO.NET DataAdapter, 80
DeletedRowInaccessibleException, reason thrown, 306
Delete() method, marking a selected row for deletion with, 226–227
DeleteRule, ForeignKeyConstraint, 290
DELETE stored procedure, generated by the DACW, 253
Depth property, of a DataReader, 147
DeriveParameters() method, using to construct the Parameters collection, 248–249
DiffGram
“before” block, 320
function of, 319–321
InfoRequest block, 320
valid settings for hasChanges tag, 321
XML header, 319
XmlWriteMode option argument, 323
.disco XML discovery file, from creation of example Web Service, 326–327
Dispose() method, Connection object, 55
dot (.) notation, using when addressing specific classes in a namespace, 21

drag-and-drop (D&D)
reason not to use for ADO.NET data objects, 56–57
using with the .NET IDE, 102–105
Driver property, ADO.NET, 54
DSN=(Odbc only), Connection string argument, 48
DuplicateNameException, reason thrown, 307

E

EnforceConstraints property, postponing constraint enforcement with, 228
Enlist (SqlClient only)
Connection string argument, 48
keyword for SqlClient.SqlConnection object, 73
Environment.TickCount() method, capturing tick values with, 137–140
Equals() method, inherited from parent object, 148
error handlers, creating custom, 341–342
error handling, types supported by .NET Framework, 301–302
error handling strategies, strategies when executing the Update() method, 273–276
error management, strategies for in ADO.NET, 301–311
error messages, displaying, 311
EvaluateException, reason thrown, 307
event handlers, coding to validate your data as it is entered, 229–233
Exception class properties, 308
exception handling
trapping specific vs. general exceptions, 306–310
for Update() methods, 277–280
ExecuteNoQuery() method, purpose of, 133
ExecuteReader CommandBehavior arguments, 134–135
ExecuteReader() method, purpose of, 133
ExecuteScalar() method
purpose of, 133
using, 135–136
ExecuteXMLReader, purpose of, 133
ExtendedProperties, 161
Extensible Schema Definition (XSD), 11–12

F

FieldCount property, of a DataReader, 147
File name= (OleDb only), Connection string argument, 48
Fill() method
extracting rows from the data source with, 106–107
importing an ADOc Recordset into ADO.NET with, 168–172
using to name a table, 176
using to populate a DataSet with DataTables, 178–179
FillSchema() method
rules for configuring PrimaryKey and Constraints properties, 180–181
using, 294–295
using to retrieve DDL, 180–181
FilterExpression argument, compared to an SQL WHERE clause, 198–199
filtering, on RowState and version, 204
filter properties, setting for DataView object, 206–210
Find() method, for DataView object, 216–218
Find() methods, using, 212–219
ForeignKeyConstraint
DeleteRule actions, 290–291
rule properties for, 290
ForeignKeyConstraint objects, creating, 293–294
foreign key constraints
how ADO.NET implements them, 286
implementing, 289–291

G

Gacutil.exe tool, 60
garbage collection, how .NET handles it, 55
garbage collector, managing the ADO.NET, 68–69
Generate DataSet dialog, 99–100
Generate the SQL statements dialog, Advanced Options ... button on, 89
GetBoolean() method, 149
GetByte() method, 149
GetChanges() method, using to manage modified rows, 272
GetChar() method, 149
GetChars() method, 149
GetData() method, 149
GetDataTypeName() method, 149
GetDateTime() method, 149
GetDecimal() method, 149
GetDouble() method, 149
GetFieldType() method, 149
GetFloat() method, 150
GetGuid() method, 150
GetHashCode() method, inherited from parent object, 148
GetInt16() method, 150

GetInt32() method, 150
GetInt64() method, 150
GetLifetimeService() method, inherited from MarshalByRefObject, 148
GetName() method, for DataReader, 148
GetOrdinal() method, for DataReader, 148
GetSchemaTable() method, for DataReader, 148
GetSqlBinary() method, 150
GetSqlBoolean() method, 150
GetSqlByte() method, 150
GetSqlDateTime() method, 150
GetSqlDecimal() method, 150
GetSqlDouble() method, 150
GetSqlGuid() method, 150
GetSqlInt16() method, 150
GetSqlInt32() method, 151
GetSqlInt64() method, 151
GetSqlMoney() method, 151
GetSqlSingle() method, 151
GetSqlString() method, 151
GetSqlValue() method, 151
GetSqlValues() method, 151
GetString() method, 150
GetType() method, inherited from parent object, 148
GetUpperBound() method, nuances of, 199
GetValue() method, 150
GetValues() method, 150
  using to fetch a row, 154–155
GetXml, DataSet XML() method, 322
GetXml() method, writing a DataSet to a string with, 322
GetXmlSchema, DataSet XML() method, 322
GetXmlSchema() method, extracting the XML (XSD) schema with, 322–323
GetXXCommand() methods, using, 243

H

HasVersion property, determining presence of data version values with, 234–237
Hejlsberg, Anders, C# crafted by, 3
HelpLink Exception class property, 308

I

IDE. *See* .NET IDE
identity values, retrieving, 258–261
IgnoreSchema option argument, XmlWriteMode, 323
Indexer property, using to access DataRow object data values, 188–190
inference rules, InferXmlSchema() method, 325
InferXmlSchema, DataSet XML() method, 322
InferXmlSchema() method, 324–325
InfoMessage event, fired on the Connection object, 56
InfoRequest block, for DiffGram, 320
Initial Catalog (OleDb and SqlClient only), Connection string argument, 48
InitializeComponent() method, using, 56–57
InitializeLifetimeService() method, inherited from MarshalByRefObject, 149
InnerException Exception class property, 308
InRowChangingEventException, reason thrown, 307
InsertCommand
  generated by the DACW, 255–256
  generating with the CommandBuilder, 243–244
InsertCommand object, ADO.NET DataAdapter, 80
INSERT stored procedure, generated by the DACW, 252
Integrated Security or Trusted_Connection, Connection string argument, 48
IntelliSense™. *See* Visual Studio IntelliSense™
InvalidExpressionException, reason thrown, 307
IsClosed property, of a DataReader, 147
IsDBNull() method, for DataReader, 148
IsNull() method, using with the DataRow or DataColumn objects, 192–193
Isolation Level (OleDb only), Connection string argument, 49
ItemArray() method, using, 190–191
ItemArray property, assigning data values to a row with, 191
Item property
  of a DataReader, 147
  using to access DataRow object data values, 188–190

J

just-in-time connections, examples of, 107–108

K

KeyInfo, ExecuteReader CommandBehavior argument, 134

M

managed providers. *See* .NET Data Provider; ADO.NET .NET Data Providers
Master Detail Drill-Down sample, 300
  information request example, 337
Max Pool Size (SqlClient only)
  Connection string argument, 49
  keyword for SqlClient.SqlConnection object, 73
MDAC SDK, installing, 45
merging DataSets, 281–282
Message Exception class property, 308
Microsoft.Data.Odbc, establishing a connection with, 39
Microsoft Jet ODBC Driver, compatible with Odbc .NET Data Provider, 60
Microsoft ODBC Driver for Oracle, compatible with Odbc .NET Data Provider, 60
Microsoft SQL ODBC Driver, compatible with Odbc .NET Data Provider, 60
Min Pool Size (SqlClient only)
  Connection string argument, 49
  keyword for SqlClient.SqlConnection object, 74
MissingPrimaryKeyException, reason thrown, 307
multiple-resultset queries, managing, 129–132
multiple-table queries, managing, 116–121

N

name arguments, list of valid, 48–50
.NET. *See also* ADO.NET
  handling of garbage collection by, 55
.NET applications, how COM Interop affects them, 19–20
.NET Cache object, 110
.NET constraints, understanding, 283–285
.NET constructors, initializing variables with, 108–110
.NET Data Providers
  ADO.NET, 15–20
  choosing the right one, 41–42
  COM-based ADO as, 75–77
  currently shipped with ADO.NET, 15
  establishing connections using different, 39–77
  use of XML by, 41
  values passed for setting options, 42–43
.NET Framework system, what the changes mean to programmers, 5–6
.NET IDE
  creating a DataSet with, 98–101
  using drag-and-drop with, 102–105
  using to get connected, 66–68
.NET object reference performance, comparing, 137–140
.NET Server Explorer. *See* Server Explorer
.NET Server Explorer window. *See* Server Explorer window
Network Library or Net
  Connection string argument, 49
  default setting for, 51
New Project dialog, Visual Studio .NET, 328
NextResult() method, for DataReader, 148
NoNullAllowedException, reason thrown, 307
null constraints, how ADO.NET implements them, 285
NULL values, working with, 192–193

O

object cleanup, how .NET handles, 55
OdbcCommand object, supported by Odbc .NET Data Provider, 61
OdbcConnection instance object variable, declaring, 63
OdbcConnection object
  managing, 61–64
  supported by Odbc .NET Data Provider, 61
Odbc connection pooling, debugging the connection pool, 75
ODBC connection strings, examples of valid, 64
OdbcDataAdapter object, supported by Odbc .NET Data Provider, 61
OdbcDataReader object, supported by Odbc .NET Data Provider, 61
ODBC Driver Manager utility, tracing and connection pooling handled by, 60
ODBC drivers, compatible with the Odbc .NET Data Provider, 60
Odbc .NET Data Provider
  connecting with, 59–64
  downloading, 2
  establishing a connection with, 39
  installing, 40
  objects supported by, 61
  ODBC drivers compatible with, 60
  use of Platform Invoke (PI) by, 59
  using, 60–61
  Web site address for downloading, 39
OdbcPermissionAttribute object, verifying adequate code permissions with, 62

OdbcPermission object, creating security demands with, 62
OleDb connection pooling, debugging the connection pool, 75
OleDbDataAdapter Fill() method, using to import an ADOc Recordset, 168
OleDbException class, 310
OleDb .NET Data Provider, establishing a connection with, 39
"O'Malley" rule, using, 199
one-to-many relationships, database diagram of, 292
Open() method, Connection object, 55
optimistic locking strategies
  importance of designing systems to support, 222
OUTPUT parameters
  capturing for stored procedures, 123–127
  creating, 126

P

Packet Size, Connection string argument, 49
PacketSize property, ADO.NET, 54
Parameter object constructors
  reviewing, 112–114
  using a datatype enumeration as alternative, 113–114
Parameter object properties, constructor for defining in a single line of code, 114
parameter queries
  introduction to ADO.NET, 82–84
  managing, 111–115
parameters
  setting the correct version for, 237–238
  using the Add() method to construct, 115
Parameters collection
  constructing using the DeriveParameters() method, 248–249
  construction of, 111–112
parent/child primary key/foreign key relationships, understanding, 292–294
Password, or Pwd, Connection string argument, 49
performance counters, monitoring the connection pool with, 74
Persist Security Info, Connection string argument, 49
pessimistic locking, 221–222
  using in your ADO.NET application, 249
Platform Invoke (PI), use of by Odbc .NET Data Provider, 59
Pooling (SqlClient only)
  Connection string argument, 50
  keyword for SqlClient.SqlConnection object, 74
primary key constraints, how ADO.NET implements them, 285
PrimaryKey property, setting for Rows collection Find() method, 213–214
Prompt (OleDb only), Connection string argument, 50
Provider (OleDb only), Connection string argument, 50
Provider property, ADO.NET, 54

Q

queries
  constructing SQL for, 85–98
  executing ad hoc to perform updates, 280
  executing traditional, 105–110
  managing for stored procedures, 121–129
  managing multiple resultset, 129–132
  managing multiple table, 116–121
  testing for results, 191–194
  using ADO.NET to execute ad hoc, 84–85
Query Analyzer, using to deal with multiple resultsets, 119–120
Query Builder
  building SQL text with in DACW, 89–93
  improvements in Visual Studio .NET IDE, 86
Query Builder window, opening in Visual Studio, 262

R

Read() method
  for DataReader, 148
  importance of using with DataReader objects, 146
ReadOnlyException, reason thrown, 307
ReadXml(), DataSet XML method, 322
ReadXML() method
  of DataSet object, 316–319
  XmlReadMode arguments, 317–319
ReadXmlSchema() method
  DataSet XML, 322
  extracting schema from an XML document with, 324
RecordsAffected property, of a DataReader, 147

RecordsAffected value, capturing for multiple-resultsets or stored procedures, 132
Recordset vs. the DataReader processing loop, 146
referential integrity constraints, understanding, 283–285
RejectChanges() method, undoing a delete operation with, 227
RemoveAt() method, deleting DataSet rows with, 226
Resultset columns, binding to, 136–140
resultsets, construction of hierarchical, 90
Return Value, capturing for stored procedures, 123–127
RO/FO (read-only/forward only), 33, 36
row counts, subroutine for showing, 270–272
RowError() method, using to set an error string, 193
RowNotInTableException, reason thrown, 307
Rows collection, testing the Count property of, 192
Rows collection Contains() method, using, 219
Rows collection Find() method, 213–216
  setting the PrimaryKey property for, 213–214
Rows.Find() method, executing, 214–216
Rows.Remove() method, deleting a specific Row object with, 226
RowStateFilter property, supported by the DataView, 206
RowState property, checking the RowState with, 233

S

Sceppa, David, 254, 283
SchemaOnly, ExecuteReader CommandBehavior argument, 134
security issues, when working with connections, 69–71
SelectCommand
  creating for use in a DataAdapter, 63–64
  executing individual, 116–118
  generated by the DACW, 254
  setting up in ADO.NET, 121–123
SelectCommand objects
  ADO.NET DataAdapter, 80
  executing multiple resultset, 118–121
SELECT-generated rowsets, capturing in ADO.NET, 129–132
Select() method, filtering and sorting with, 198–204
SELECT query, soliciting the SQL text for in DACW, 89–93
SELECT stored procedure, generated by the DACW, 251
SequentialAccess, ExecuteReader CommandBehavior argument, 134
Server Explorer
  adding a connection to, 66–68
  extracting a known-working ConnectionString from, 67–68
Server Explorer window, opening, 66–67
ServerVersion property, ADO.NET, 54
SetColumnError() method, saving an error string associated with a specific column with, 193–194
SingleResult, ExecuteReader CommandBehavior argument, 135
SingleRow, ExecuteReader CommandBehavior argument, 135
SOAP and ADO.NET, 342
Solution Explorer
  showing a new WinForm application, 30
  Update Web Reference drop-down in, 335
sorting
  in ascending or descending order, 206
  with the DataTable Select() method, 199–203
SourceColumn property, setting for Parameter objects, 238
Source Exception class property, 308
SourceVersion property, setting for Parameter objects, 238
SQL (Structured Query Language), constructing for your queries, 85–98
SqlClient, ConnectionString keywords for, 73–74
SqlClient connection pooling, 71–72
SqlClient .NET Data Provider
  connecting with, 39, 51–53
  lack of ? parameter marker support by, 85–86
  speed of vs. OleDb or Odbc .NET Data Providers, 52
SqlClient.SqlConnection object, ConnectionString keywords for, 73–74
SqlCommand object properties, code for setting up, 107–108
SqlConnection, code used to set up, 102–104
SqlConnection.State property, verifying closure of, 45
SqlDataAdapter, code used to set up, 102–104

SqlDataAdapter1 Properties page, changing the names and properties of a DataAdapter in, 97–98
SqlError class, severity levels and their causes, 310
SqlException class, 308–309
SqlException namespace, properties exposed in, 308–309
SqlParameter constructors
- arguments passed to by DACW-generated code, 255
- using, 112–114

SQL punctuation, dealing with imbedded, 247–248
SqlSelectCommand, code used to set up, 102–104
SQL Server, Tabular Data Stream (TDS) language used by, 51–53
SQL Server 2000 instance (SS2K), notation used to address, 65
SQL statements, using with the DACW, 89–93
SqlUpdateCommand, code used to set up, 102–104
SQL WHERE clause, compared to a FilterExpression argument, 198–199
SQLXML, release of, 314
StackTrace Exception class property, 308
State property, ADO.NET, 54
stored procedure queries, managing, 121–129
stored procedures
- capturing the Return Value and OUTPUT parameters for, 123–127
- executing, 126–127
- generating with the DACW, 93–95
- previewing the command scripts, 95
- selecting for SELECT and action queries, 96
- using existing with the DACW, 95–98
- using with the DACW, 93

strongly typed DataSets
- constructing, 163–165
- manually assembling, 165–167

StrongTypingException, reason thrown, 307
structured exception handling, 301–302
SysntaxErrorException, reason thrown, 307
System.Data.DataSet object, ADO.NET, 22–26
System.Data exceptions, thrown by ADO.NET, 306–307
System.Data namespace, 21–36
- exploded view of, 27–28
- the layout of, 26–29
- selected members of, 28–29

System.Data objects
- exploded view of, 28
- fundamental in implementation of ADO.NET applications, 16–17
- instantiating, 30–31

System.Data.OleDb
- currently shipping with ADO.NET, 15
- establishing a connection with, 39

System.Data.SqlClient
- currently shipping with ADO.NET, 15
- establishing a connection with, 39
- introducing, 51–53

T

Tabular Data Stream (TDS)
- establishing a connection with, 39
- low-level language used by SQL Server, 51–53

TargetSite Exception class property, 308
ToString() method, inherited from parent object, 149
Transaction object, creating from ADO.NET, 59
transactions
- managing from your ADO.NET application, 59
- support for, 73

transmogrify, defined, 313
Trusted_Connection or Integrated Security, Connection string argument, 48
try/catch blocks
- effect of cradling declaration statements in, 109–110
- variable scope in, 305

try/catch exception handlers
- nesting, 305–306
- standard exceptions, 303–304
- syntax, 303
- understanding, 303–306
- use of in ADO.NET, 301–311

try/catch scope, exceptions that fire outside of, 305
TypedDataSetGeneratorException, reason thrown, 307
typed DataSets
- built-in type checking, 162
- implementing, 161
- vs. untyped DataSets, 161–168

U

unique constraints
- how ADO.NET implements them, 285
- implementing, 287–289

unstructured exception handling, 301
untyped DataSets, using, 167–168

UpdateCommand
generated by the DACW, 256–257
generating with the CommandBuilder, 244–245
understanding the plan, 246–247
UpdateCommand object, ADO.NET DataAdapter, 80
Update exceptions, 278
Update() methods
data row changes fired by, 232–233
error handling strategies, 273–276
exception handling, 277–280
typical errors encountered, 278–279
understanding, 262–264
what happens when they are executed, 250
what to do when they fail, 276
UpdateRule, ForeignKeyConstraint, 290
UPDATE stored procedure, generated by the DACW, 252
update strategies, performance of, 264–266
UserID or UID, Connection string argument, 50

V

Value property
addressing with typed DataSets, 224
addressing with untyped DataSets, 223–224
variables, using .NET constructors to initialize, 108–110
VB .NET, development of, 3–4
VersionNotFoundException, reason thrown, 307
Visual Basic .NET, accessing ADOc objects from, 77
Visual Studio
creating DataView objects with, 210
opening a Query Builder window in, 262
using to create Command objects, 261–262
using to generate action queries, 251–258
Visual Studio IntelliSense™, exposing DataSet members, 162
Visual Studio .NET
New Project dialog, 328
test harness HTML page, 331
Visual Studio .NET Toolbox, Data tab, 87
Visual Studio's CommandBuilder vs. ADO.NET .NET Data Providers', 253–254
VSDisco discovery file, example of typical, 327

W

Web Reference, updating, 334–336
Web Service
AuthorByISBN parameter prompt, 332
BiblioServiceException class, 328
code for changing the namespace, 329
code to post changes to the database, 338–341
creating as an XML data source, 326–336
creating custom error handlers for, 341–342
debugging, 336
functions to expose AuthorByISBN() and TitlesByAuthor() methods, 329–330
passing an updated DataSet back to, 336–337
testing, 331–336
Web site address
for downloading Odbc .NET Data Provider, 2, 39
for information about SQLXML, 314
Windows 2000 Performance Monitor, accessing performance counters in, 74
Workstation ID, Connection string argument, 50
WorkstationID property, ADO.NET, 54
WriteSchema option argument, XmlWriteMode, 323
WriteXml() method, DataSet XML, 322
WriteXmlSchema() method
DataSet XML, 322
exporting a DataSet objects schema with, 325

X

XML (Extensible Markup Language)
and ADO.NET, 313–342
a brief look at, 11–12
fetching and saving DataSet objects as, 321–323
loading schema from, 324–325
performing bulk INSERTs with, 265–266
support in ADO.NET, 314–315
use of by ADO.NET and .NET Data Providers, 41
XML and DataSet objects, understanding, 316–325
XML data, merging with your DataSets, 321
XML DataSet, browser rendered for Web Service, 333

XML data source, creating a Web Service as, 326–336
XML documents, importing DataSet data from, 316–319
XML formats, standardizing, 315–316
XML header, for DiffGram, 319
XmlReadMode arguments, for ReadXML() method, 317–319
XmlReadMode Auto argument, function of, 317
XmlReadMode DiffGram option, function of, 318
XmlReadMode Fragments option, function of, 318–319
XmlReadMode IgnoreSchema option, function of, 318
XmlReadMode InferSchema option, function of, 318
XmlReadMode ReadSchema option, function of, 318
XML schema, persisting, 325
XmlWriteMode option arguments, 323
XSD. *See* Extensible Schema Definition (XSD)
XSD.exe, creating a XSD-specific DataSet class with, 164
XSD schemas, automatic generation of by Visual Studio .NET, 12