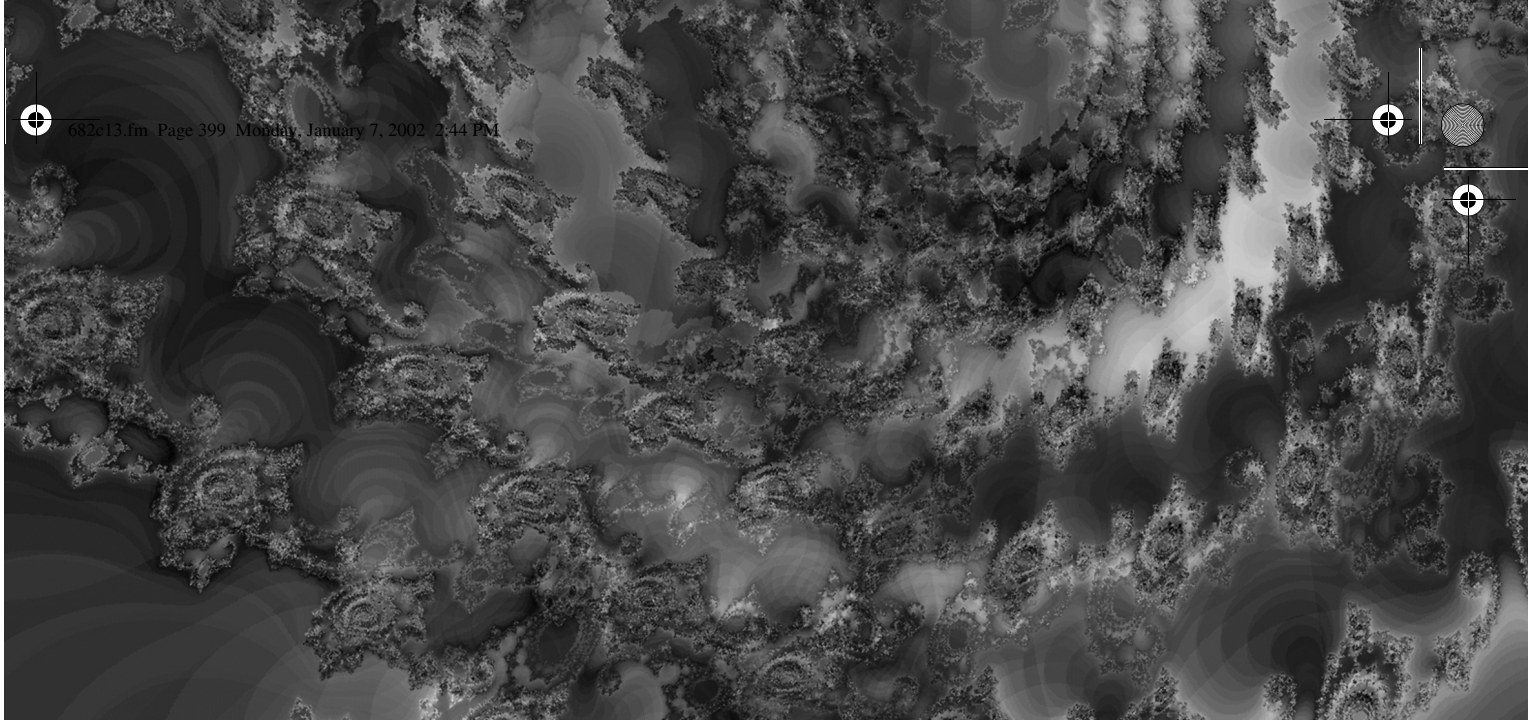


Part Two

ADO.NET and the .NET Framework

CHAPTER 13

Introducing ADO.NET

UP TO THIS POINT this book has been about COM-based ADO; from this point forward it's all about the .NET Framework,[1] ADO.NET, 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 WinForms 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. In this second half of the book, 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 namespace implements the System.Data classes in much the same way. Sure, I point out areas where there seem to be differences between the provider implementations; but no, I won't be getting into the Odbc .NET Data Provider. Because of Microsoft's hesitancy to keep this provider up to date during the beta cycle, my technical editors and I were unable to include much more than a passing reference to this provider. Check my Web site[2] for an update sometime after this book hits the streets for differences and issues.

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

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.
2. `http://www.betav.com`

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 System's 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.

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. It wasn't long before it became clear that Microsoft needed something new—entirely new—in order to accomplish this goal. 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.

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.

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 and that ADO.NET is just another version of COM-based ADO. They clearly aren't the same—not even close.*

A poll of developers taken in the fall of 2001 resulted in a pretty even division. Almost half feel that Visual Basic needs to be reborn anew without the deadwood—especially in light of the ever-evolving technology we're now facing. The other half feel that they have too much invested in training, infrastructure, code, and tears to abandon Visual Basic as they know it. The few outliers are just uncertain and are thinking about moving to Florida or Illinois where their choice does not seem to make much difference.

Frankly, over the last year, I've grown to like Visual Basic .NET—but at a price. I found it tough to figure out how to do some of the "basic" things Visual Basic developers have taken for granted for a decade. These tasks include things like adding items to a list box, managing variable scope, setting up an error handler, managing a control array, constructing a UI component, declaring arrays and classes, and knowing what property to set when I wanted something to show up in a control. Most of these little things have changed so even an accomplished Visual Basic developer will (at least initially) feel lost and encounter a degree of frustration. After this learning period is over and the Visual Basic 6.0 developer feels comfortable with the language, he or she is then faced with learning how to use ADOc or the all-new ADO.NET in this new language. That's what this book is for. For no extra charge I'll even throw in some Visual Basic .NET tips and tricks along the way to make the overall transition easier.

I think the new Visual Basic .NET language is just that: new. 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?

Microsoft's answer to the demands made by developers working in this new arena is the Microsoft .NET Framework. This 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 (WinForms) 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 22, "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.[3] I explain what this means in Chapter 22.

For the Visual Basic 6.0 developer, these .NET innovations mean a new runtime platform and forms engine, as well as a new user interface and development IDE. Both the existing VBRUN.DLL and the accompanying "Ruby" Windows Form Engine have been replaced (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.

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

The Visual Basic 6.0 IDE has 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 Basic 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 is just another language. For others, it's a big, scary step away from their comfort zone.

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—at least in the ADO.NET chapters—"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[4]), their new data access object library was referred to as XDO (among other things). To me, this made[5] 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

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

of "Visual Basic" 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 the 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 included with the .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, 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 on 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 it to 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. 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 (System.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[6] 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.[7] 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

6. See `http://www.w3.org` for more information

7. I expect this list to change (expand, contract) as XSD is nailed down.

XML syntax to define a schema, meaning that the same parsers can process both data and schemas.

IIRC,[8] 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 22) 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.

8. 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 WinForms 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 17, "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.

Visual Basic .NET's 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 16) 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 17 and in Chapter 20, "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 amount 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 15, "ADO.NET Command Strategies" and Chapter 19, "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 (System.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.[9]

- **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.*

As I said earlier the System.Data.Odbc provider is scheduled to be made available via Web download not long after .NET is 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

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

Provider is twenty percent faster than its COM interop brother the OleDb .NET Data Provider.

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.[10] 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
- **The DataAdapter object**: SqlDataAdapter, OleDbDataAdapter, OdbcDataAdapter
- **The DataReader object**: SqlDataReader, OleDbDataReader, OdbcDataReader

10. RTM: Release to manufacturing.

.NET Data Providers also directly support and implement code to generate Commands, 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 drop your ADOc code into your .NET code and register ADO 2.x in your solution. This gets .NET to generate a COM interop wrapper around MSADO21.DLL and include it 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 14, "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 14. 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. However, as I said before, you might notice a drop in performance or somewhat different behavior due to COM interop.

That said, while your existing ADOc logic has to be recoded (at least to some degree) to run in .NET, the basic functionality should work (about) the same. You should be able to use the same flow, the same error handlers, and the same methods, properties, and events as you did in Visual Basic 6.0—at least that's the goal for the Microsoft .NET development team. If you need to access pessimistic cursors, server-side cursors, manage and maintain server-side state, or clone functionality implemented in Visual Basic 6.0 applications, you need to keep using ADOc to do so. None of these features are supported—at least not yet—in ADO.NET.

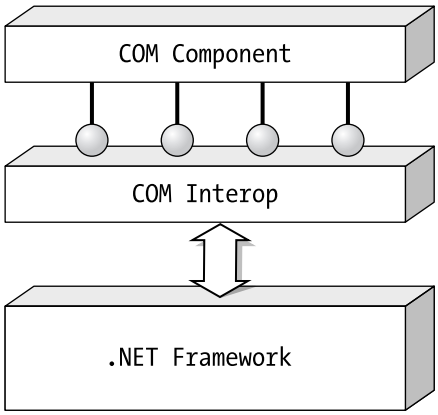
Creating DataSets from ADOc Recordset Objects

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 DataSet Fill method directly recognizes ADOc Recordset

and Record objects. This functionality enables .NET developers to use existing COM objects that return ADO objects without having to rewrite new objects using the .NET Framework. Both the OleDb and SqlClient .NET Data Providers support filling a DataSet from an ADO Recordset or Record object. I illustrate this with an example in Chapter 16.

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 13-1 illustrates this extra layer of protection wrapped around all COM components.



All COM object, property, method, and event references pass through the COM interop layer.

Figure 13-1. COM components access .NET via 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 burger 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*[11] than you planned on—but for me, that's okay!

11. A little more onion.

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 its 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.

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 backend 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.

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 breath 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.

```
Dim myConnection as 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[12] 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

12. http://www.msdn.microsoft.com/library/en-us/cpref/html/cpref_start.asp

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: a Visual Basic 6.0 Data Source control,[13] 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 14.

You can construct your own DataTable objects by query or by code—defining each DataColumn object one-by-one and appending them to the DataColumnCollection, 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

13. The ADO Data Control, the Jet Data Control, and your hard-coded data source controls fall into this category.

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 16 and 20.

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 a 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 20. 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 13-2) 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) is 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 16.

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.

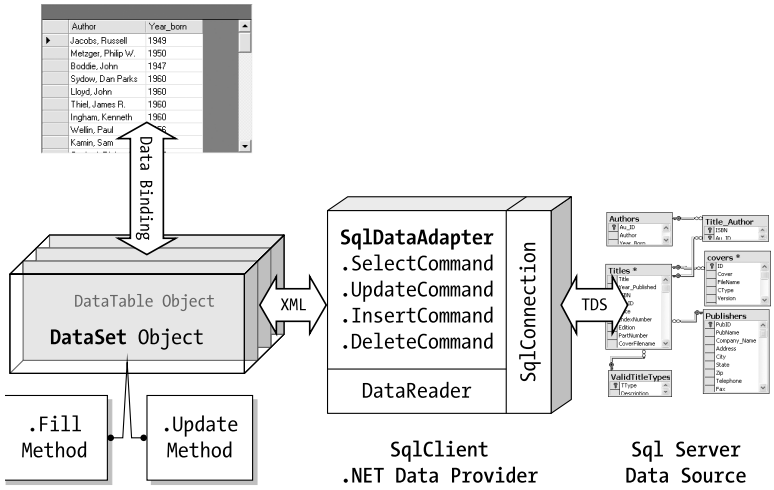


Figure 13-2. ADO.NET Data Access using the DataSet object.

Figure 13-3 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 13-3. 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.

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.

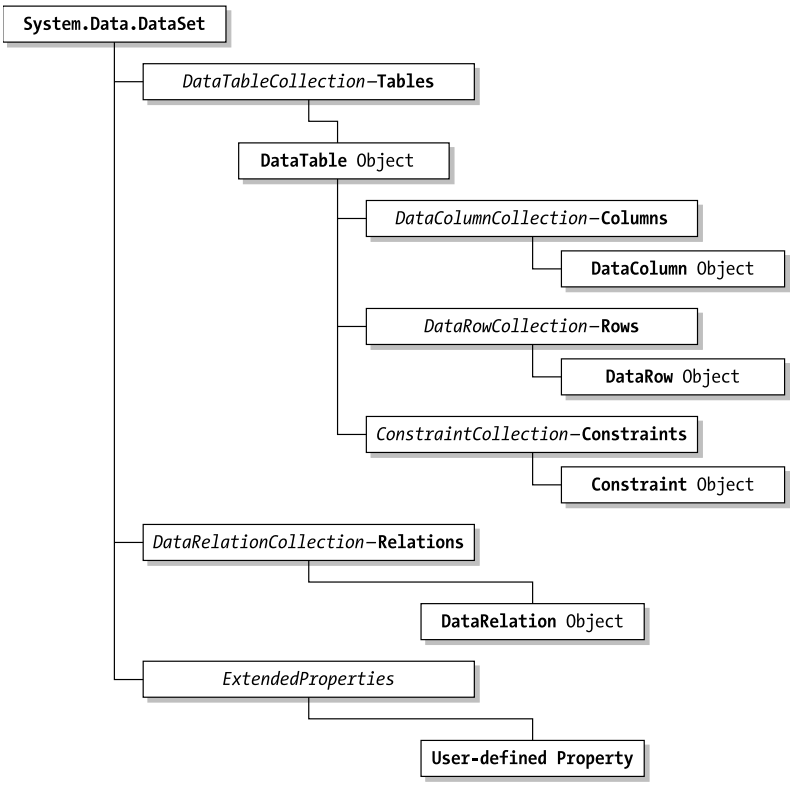


Figure 13-3. DataSet object hierarchy.

- **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 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 diagramed object hierarchy. ADO.NET's System.Data namespace, however, is far more complex. As it currently stands, there are dozens upon dozens[14] 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 Basic .NET. Figure 13-4 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 be thoroughly sick of it.

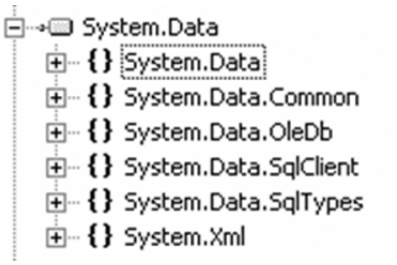


Figure 13-4. The System.Data namespace.

System.Data Namespace Exploded

The exploded System.Data namespace has over forty members—the top dozen or so are shown in Figure 13-5. 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 13-1 lists and describes the most important of these objects—the ones you'll use most often (at least at first).

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

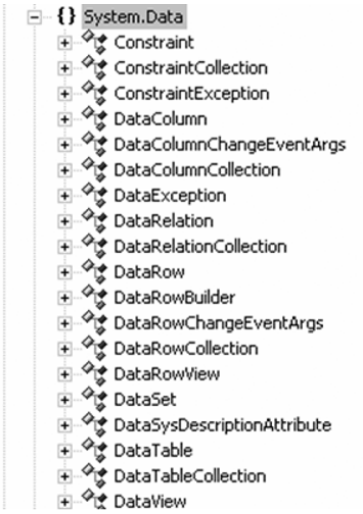


Figure 13-5. System.Data objects.

Table 13-1. 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. Contains information about the error. Similar to the ADOc Error object.
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.

Table 13-1. Selected Members of the System.Data Namespace (Continued)

Object	Description
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—just as in Visual Basic 6.0. Fortunately, the .NET IDE works pretty much like Visual Basic 6.0, but 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. If you aren't using the Odbc provider when you create a new solution, the Solution Explorer references show that System.Data is part of the base namespace. The Solution Explorer is a handy way to see what namespaces are already referenced for your application's assembly, as shown in Figure 13-6.

Depending on the ADO.NET data access provider you choose, you'll want to use the Imports operator with either System.Data.OleDb or System.Data.SqlClient (or in unusual situations, both), or System.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 Imports list makes coding easier by providing "shorthand" syntax for commonly used objects. Although not required for ADO.NET, the Imports operator signals the compiler to search the specified namespace referenced in your code to resolve any ambiguous object names. Basically, using Imports helps the compiler resolve namespace references

Figure 13-6. The Solution Explorer showing a newly created WinForm application.

more easily. The Visual Basic .NET Imports statement should be positioned first in your code—above all other declarations, but after any Option settings. For example, to import the OleDb .NET Data Provider, add the following to your code module:

```
Imports System.Data.OleDb
```

To import the SqlClient .NET Data Provider, add the following to your code module:

```
Imports System.Data.SqlClient
```

Because you used the Imports operator with the System.Data.SqlClient .NET Data Provider, you can code:

```
Dim Cn as 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—especially in the Dim statement. 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 using:

```
Dim Cn as New System.Data.SqlClient.SqlConnection( )
```

However, I don't 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. Both (well, 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[15] object to manage the database connection through the Command object's Connection property. I discuss the Connection object in Chapter 14.

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 19. The figure shown earlier (Figure 13-2) 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 15, "ADO.NET Command Strategies."

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.[16] (Don't worry about the code I don't explain here—I discuss many of these points again in the next chapter.)

15. Actually, the name of the Connection object is SqlConnection or OleDbConnection.

16. Located on the CD in the "\Examples\Chapter13\Data Adapter" folder.

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.

```
Imports System.Data.SqlClient.
```

Next, within the address range of your Form's[17] class, define the objects and variables to be used. Visual Basic .NET permits you to define a value for a variable or an object's default property (there are no default properties in .NET). In this case, you're defining a ConnectString value and assigning it to the SqlConnection object's ConnectString property. As you'll discover in the next chapter, this has its drawbacks. I am also intentionally using the New operator with the Dim statement.[18]

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim strConnect As String =  _
      "data source=.;database=biblio;uid=admin;pwd=pw"
    Dim strQuery As String = "Select Title, Price from Titles where ... "
    Dim cn As New SqlConnection(strConnect)
    Dim da As New SqlDataAdapter()
    Dim ds As New DataSet()
```

In the Form1_Load event, 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 the new .NET Try/Catch error handler.[19] In the Catch statement, you reference the System.Data.SqlClient.SqlException object simply as `SqlException` (remember, you used the Imports System.Data.SqlClient statement 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. If the Cn.Open statement does not work, the next statement is never executed. If there is an error, you use the familiar Debug object with a new method: WriteLine.

17. The default architecture in most examples (before I get to Chapter 22) is WinForms. The ADO.NET concepts I use apply universally in most cases.

18. No, `Dim xx as New` is no longer evil in Visual Basic .NET.

19. Error handling is discussed in Chapter 21, "ADO.NET Error Management Strategies."

```
+ Windows Form Designer Code  ' (Black-box code used to handle your form)

 Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
 Handles MyBase.Load
        Try
            cn.Open()
            da.SelectCommand = New SqlCommand(strQuery, cn)
       Catch ex As SqlException
            Debug.WriteLine(ex.Message)
        End Try

    End Sub
```

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 16. The options I chose in this example name the resulting DataTable "TitlesandPrice." In the next statement, I bind the DataSet to the DataGrid control—just as you would have in Visual Basic 6.0, but without the Set statement (it's no longer supported or needed in Visual Basic .NET).

```
Private Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button1.Click
        da.Fill(ds, "TitlesandPrice") ' Defaults to "Table"
        DataGrid1.DataSource = ds
    End Sub

End Class
```

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 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:

- **Using Imports System.Data.SqlClient helps.** Statement completion did not show the objects I was referencing nearly as quickly (if at all) until I added the Imports statement.

- **I keep forgetting to use `Dim xx As New yy`.** I've disciplined myself (no, not with leather flails) to avoid their use in Visual Basic 6.0—but now I have to learn again when it's best to use `Dim xx As New yy`.

- **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.

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. This can save you an extra ten minutes as you try to figure out what went wrong. Until Visual Basic .NET is really working, I expect that many of the "Unhandled exception" errors will make our lives tough. The default name for the "Filled" DataSet is "Table". You want to override that in many cases.*

A Simpler Example

Okay, now that I showed you an example based on how an ADOc developer might code, take a look at the same problem using the new ADO.NET approach. In this case, I used the Dim constructors to set specified properties in the classes as they were being instantiated; I used the Fill method to manage the connection for me. There's nothing to do at Form_Load time and I bound to the first DataTable in the DataSet object's Tables collection. Think of a constructor as simply a macro used to instantiate an object and set one or more properties. 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.

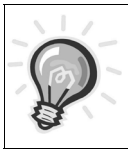
```
Dim cn As New SqlConnection("data source=.;database=biblio;" _
  & "uid=...")[200]
Dim da As New SqlDataAdapter("Select Title, Price " _
  & "  from Titles where… ",cn)
Dim ds As New DataSet()
Private Sub btnRunQuery_Click(ByVal sender As System.Object,
   ByVal e As System.EventArgs) Handles btnRunQuery.Click
    da.Fill(ds, "TitlesAndPrice")
    DataGrid1.DataSource = ds.Tables("Titles and Price")
End Sub
```

ADO.NET's Low-Level Data Stream

By this time you know that by default, ADOc Recordsets are created as RO/FO 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).

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[21] 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, .NET permits you to declare and initialize selected properties of the declared objects in a single line of code. I discuss the subtleties of this as I get into the details of these methods a little later.



TIP *Note the difference in syntax between the declarations that use the* `New` *operator and those that don't. As a rule of thumb, don't use parentheses unless you use* `New`.

20. Note that some of the code samples are shortened for brevity. All examples in the book are organized by chapter and provided on the companion CD.

21. Located in the "\Examples\Chapter13\Data Stream" folder on the CD.

```
 Imports System.Data.SqlClient
…
Dim cn As New SqlConnection("data source=.;database=biblio;uid=admin;…
Dim da As New SqlDataAdapter()
Dim cmd As New SqlCommand("Select Title, PubID from Titles …",cn)
Dim Dr As SqlDataReader
```

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. The new syntax used to add items to the ListBox is shown in the example. Note that you have to use the Add method in Visual Basic .NET to add members to any collection—including ListBox and ComboBox control Items collections.[22] 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—at least when compared to BASIC.

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

22. You'll find many surprises when working with .NET. Many of the bread-and-butter controls and functions have "evolved."

```
Private Sub Button1_Click(ByVal sender As System.Object,  _
ByVal e As System.EventArgs) Handles Button1.Click
 Cn.Open
 Dr = cmd.ExecuteReader()
    With ListBox1
      .Items.Clear()  ' clear the listbox
      .BeginUpdate()  ' Prevent the listbox from painting
      While Dr.Read() ' get the first (or next) row
        .Items.Add(Dr.GetString(0) & " - " _
           & CStr(Dr.GetInt32(1)))
      End While
      .EndUpdate()    ' Let the listbox paint
    End With
  Cn.Close            ' Close the Connection
  Dr.Close()          ' Close the data reader.
End Sub
```