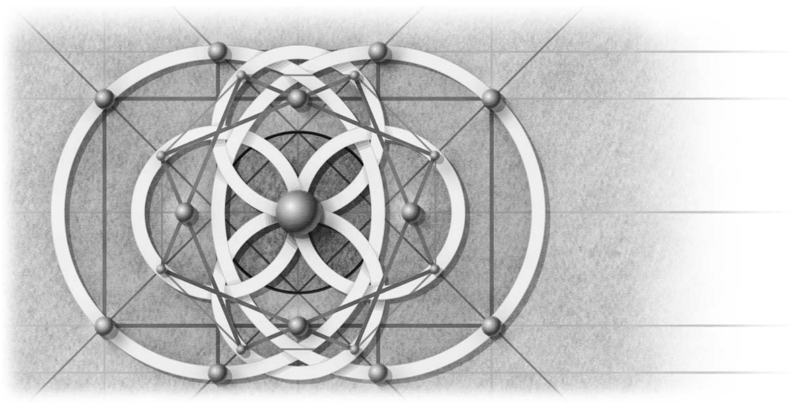
20



Moving from ADO to ADO.NET

Let's face it—most Microsoft Visual Basic applications have some sort of data access. If your application uses ActiveX Data Objects (ADO), you probably want to know what improvements you can make now that you are upgrading to Visual Basic .NET. The .NET Framework offers a whole new set of controls and services, including ADO.NET. Naturally, you will want to take advantage of some or all of these new features. From the developer's standpoint, you have three options available:

- Adapt your usage of ADO to better interoperate with the .NET Framework objects.
- Upgrade your ADO code to ADO.NET.
- Do both.

This chapter introduces the concepts and class hierarchy of ADO.NET. Then it provides an overview of how to use your existing ADO code within your .NET application to bind data to controls and support ADO *Recordset*s with XML Web services. For the more adventurous among you, the last part of this chapter offers some guidance on how to migrate your existing ADO code to ADO.NET.

ADO.NET for the ADO Programmer

It is obvious from the get-go that ADO.NET isn't the traditional ADO. The data object model provided by ADO.NET is substantially different from that of ADO and requires a different approach when designing and implementing solutions.

While some of the concepts remain familiar to ADO developers (such as using *Connection* and *Command* objects to manipulate a data source), others will require some adjustment. This discussion gives only a short overview, so we can quickly move on to the more interesting stuff.

Overview of ADO.NET

While ADO.NET presents a new data object model, parts of it are quite similar to the ADO we all know and love. Think of ADO.NET as the result of the integration of both classic ADO and the Microsoft XML technologies in a single, coherent framework. Unlike ADO, ADO.NET was designed from the ground up to support a more general data architecture that is abstracted from an underlying database. In other words, ADO.NET was designed to work with a myriad of data sources and can function as a data store in and of itself. This framework provides a series of data container objects that support functions similar to those used in databases (indexing, sorting, and views) without requiring an actual physical database behind the scenes.

ADO.NET also provides an interface that is consistent with other .NET stream-based data classes. This is a huge step forward and a wonderful design win. In Visual Basic 6, ADO uses a totally different object model compared to the *FileSystemObject* or the Microsoft XML libraries. ADO.NET, on the other hand, is designed to be consistent with other .NET Framework classes. This new compatibility enables developers to move between vastly different kinds of classes with relative ease. Figure 20-1 provides an overview of the ADO.NET class structure.

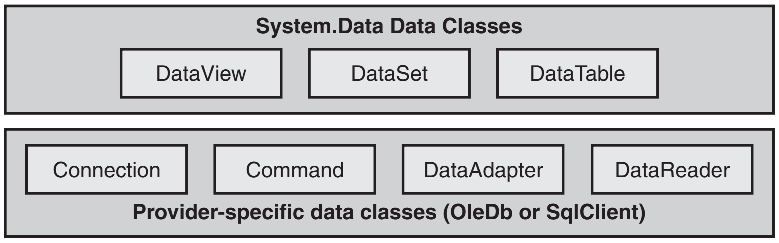


Figure 20-1 Overview of ADO.NET classes.

While the *Connection* and *Command* objects are still required for most data access, the distinction between the two objects has been made literal, as the *Connection* object now solely represents a database connection, while the *Command* object handles all queries.

The classic ADO *Recordset* functionality has been split into smaller and more specialized objects: the *DataSet* and *DataReader* classes. Forward-only "firehose" *Recordset*s map directly to *DataReader*s, while disconnected *Recordset*s map best to *DataSet*s. *DataSet* is a powerful new class designed solely to address the disconnected data model problem. It was designed specifically to fill the gaps in the way ADO handles disconnected data, which was never very satisfactory. Given the new kinds of applications that Visual Basic .NET targets, improving and enhancing data access was a primary goal. ADO.NET goes still further by providing strong integration with XML, for which ADO had only minimal support.

*DataSet*s

*DataSet*s are probably the most foreign of the new objects in the ADO.NET arsenal and thus require a bit more of an introduction than the rest. A *DataSet* is an in-memory, named cache of data retrieved from a generic data container. What this means is that the original data source is not important and does not have to be a database. *DataSet*s are extremely flexible for that reason. They can be populated from various sources, including data adapters, local data, or XML.

In essence, a *DataSet* contains collections of *DataTable*s and *DataRelation*s. The *DataTable* class conceptually corresponds closely to a database table, while the *DataRelation* class is used to specify relations between the *DataTable*s. *DataSet*s can also contain user-specific information with the ExtendedProperties collection. Contained data and the current schema (or both) can be directly extracted as XML streams, making the information highly mobile. Figure 20-2 illustrates the structure of the *DataSet* class and its relationship to the data-specific ADO.NET classes.

Important Any relationships between *DataTable*s need to be specified by the programmer. The current version of ADO.NET does not support propagating preexisting relationships from the data source to the *DataSet*. Therefore, any relationships that are defined in your SQL database (or other relational databases) need to be redefined in your *DataSet*.

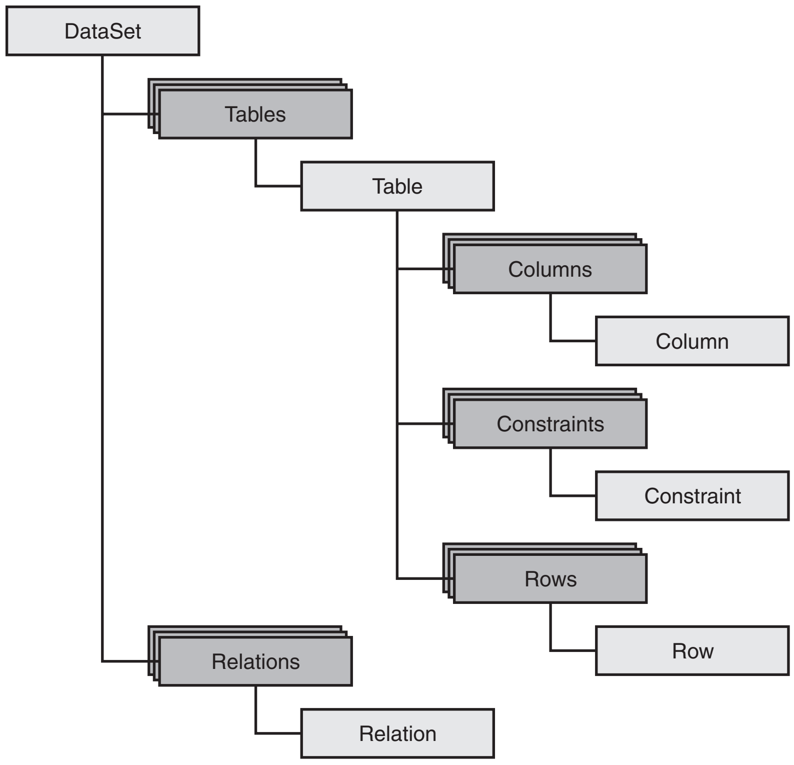


Figure 20-2 *DataSet* class hierarchy.

Using a *DataAdapter* in conjunction with a *DataSet* enables you to explicitly manage changes made to all of the tables contained within the *DataSet* and propagate those changes back to the originating data source. The *DataAdapter* is a new component in ADO.NET. It is designed to operate as an intermediary between disconnected forms of data (*DataSet*s, *DataTable*s) and the underlying data store. The simplest use of a *DataAdapter* is to fill a *DataSet* or *DataTable* with the results of a database query, but it is able to go much further than that. The *DataAdapter* is an intermediary in the truest sense of the word, in that it supports bidirectional data transfers. Using the *DataAdapter*, you can define how changes to a *DataSet* or *DataTable* are merged back into the source database and vice versa. In its most advanced use, the *DataAdapter* can specify how to handle conflict resolution when merging, including insertions and updates.

The .NET Framework ships with two types of *DataAdapter*s: the *OleDbDataAdapter* and the *SqlDataAdapter*. Although these *DataAdapter*s target different data sources, they can be used interchangeably, allowing you to move between data sources with minimal disruption. As you will see later in this chapter, the *OleDbDataAdapter* can also be used as a bridge between ADO and ADO.NET.

A Quick Note on Providers

You will see **managed providers** mentioned throughout this chapter. Two kinds of providers ship with Visual Basic .NET: the OleDb providers and the SqlClient providers, which live in the System.Data.OleDb and System.Data.SqlClient namespaces, respectively. The majority of the examples in this chapter use the OleDb managed provider because it maps the most directly to ADO. The *OleDbConnection* object can use the same connection string that ADO uses. The SqlClient managed provider is designed to talk directly to a Microsoft SQL server and uses a slightly different connection string. (There is no need to specify a provider in a connection string for the *SqlConnection* object.)

Integrating Your ADO Code into a Visual Basic .NET Application

Believe it or not, ADO has certain capabilities that enable it to be compatible with .NET applications and the .NET Framework. It is possible to preserve your existing investment in ADO while still taking advantage of .NET controls and performing such tasks as data binding and marshaling *Recordset*s through XML Web services. To this end, the .NET Framework includes some features that support transforming the old ADO *Recordset* into something consumable by a Visual Basic .NET application. This section discusses those features.

ADO with ADO.NET: Is It the Best Architecture?

Although it is certainly possible to use ADO within your Visual Basic .NET application, your mileage will vary as far as performance is concerned. This approach may not give the best performance, but it is functional and quick to implement, and it allows you to reuse your existing business objects. Later you may choose to change to a solution based solely on ADO.NET. The compatibility features discussed here span the chasm between the two technologies and make it possible to combine ADO with ADO.NET.

Binding Your ADO *Recordset* to .NET Controls

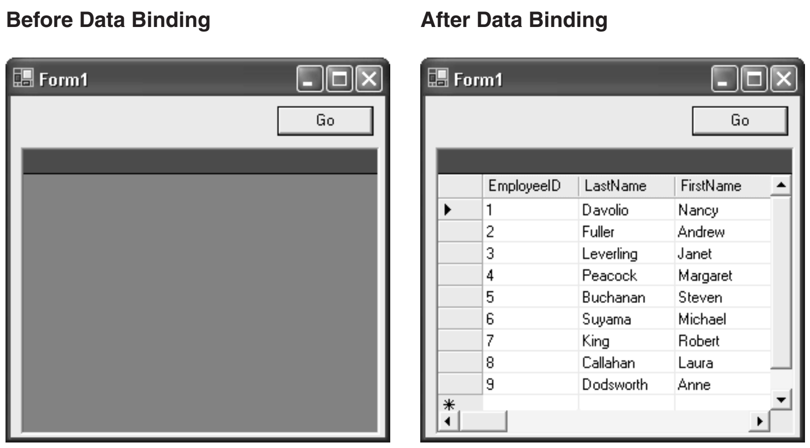
.NET provides a wide range of controls for both Web applications and Microsoft Windows–based form applications. Many of these controls support data binding with ADO.NET data sources. The question is how to take advantage of your existing ADO infrastructure and still use the new controls. The answer is simple: all you need to do is convert your *Recordset* to either a *DataSet* or a *DataTable*. Once you have one of these managed data objects, data binding is trivial.

As we mentioned previously, the *OleDbDataAdapter* has the ability to handle this task. Check out the following example:

```
'Here is the familiar ADO stuff
Dim cn As New ADODB.Connection()
Dim rs As New ADODB.Recordset()
cn.Open("Provider=SQLOLEDB.1;Integrated Security=SSPI;" & _
   "Persist Security Info=False;" & _
   "Initial Catalog=Northwind;Data Source=bart")
rs.Open("Select * From Employees", cn)

'This is where things start getting interesting. Here we declare not
'only the DataAdapter, but also the DataTable the data is destined for.
Dim olead As New Data.OleDb.OleDbDataAdapter()
Dim dt As New Data.DataTable()

'We use the DataAdapter to fill a DataTable
olead.Fill(dt, rs)

'Now we bind the DataTable to the DataGrid
Me.DataGrid1.DataSource = dt

'This is cleanup. Always close your objects as
'soon as you are done with them.
rs.Close()
cn.Close()
```

In this simple example, an ADO *Recordset* is bound to a DataGrid control on a Windows form. Figure 20-3 shows the form with the DataGrid control before and after data binding. The *OleDbDataAdapter.Fill* method is capable of populating a *DataTable* object with data from an ADODB *Recordset*.

In this way it is possible to take any *Recordset* and convert it to a managed data structure (a *DataTable*). Once you have done the conversion, you can bind the *DataTable* directly to a control, or you can add it to a *DataSet* and work with the table just as you would any other *DataTable* (including creating a *DataView* for the data, which we will get to a bit later on).



EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven
6	Suyama	Michael
7	King	Robert
8	Callahan	Laura
9	Dodsworth	Anne

Figure 20-3 DataGrid before and after data binding.

Using ADO with XML Web Services

XML Web services are the cornerstone of Microsoft's .NET initiative. They offer a simplified mechanism for implementing distributed, consumable services using SOAP, an XML-based protocol that transmits over HTTP. SOAP has significant advantages over technologies such as Remote Data Service (RDS) and Distributed Common Object Model (DCOM). XML Web services will go through Internet firewalls with no problems, and the applications are loosely coupled, eliminating issues related to component registration or changing GUIDs. These features can lead to a more robust system that is not as sensitive to changes as a COM-based system. For more information on using XML Web services, consult Chapter 21.

Suppose you've decided that XML Web services are cool and you want to replace your middle tier with one. If you use the middle tier to handle data access and pass *Recordset*s, however, you have a problem: you cannot pass any COM object through an XML Web service. COM objects lack the self-descriptive capabilities necessary to support XML serialization. How can you get the middle tier to work with Visual Basic .NET?

You actually have two options available, as Figure 20-4 illustrates. The one you choose depends entirely on your application's architecture.

- Option 1: Leave your existing architecture in place and serialize your *Recordset* to and from XML. This is a good idea if you have a lot of ADO manipulation on both the client and the server tiers.
- Option 2: Convert your *Recordset* to a *DataTable* or *DataSet* (as appropriate) and pass that back through the XML Web service. (The marshaling will be handled for you.) This option is good if the client

code is used mainly for presentation or data binding and can readily be replaced with managed code.

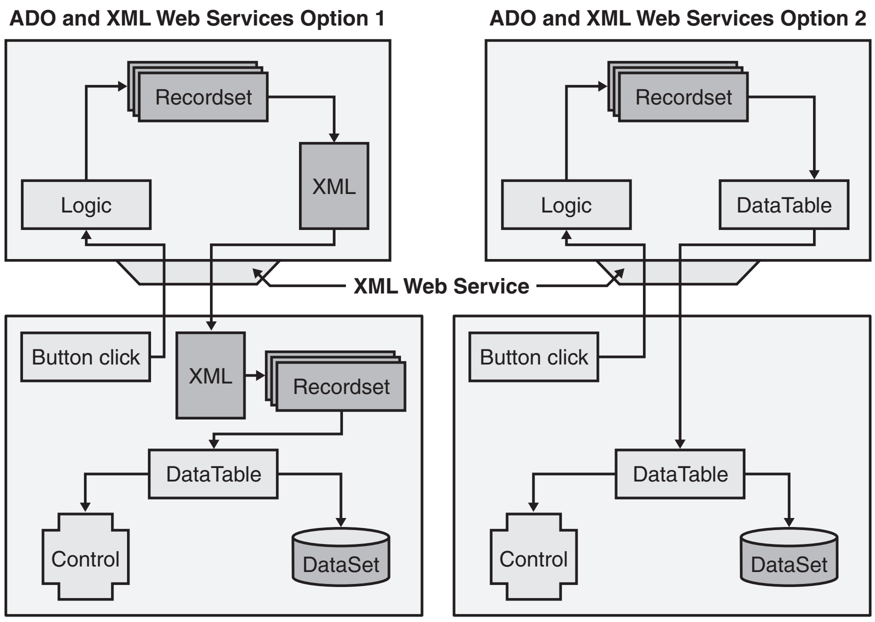


Figure 20-4 Options for implementing XML Web services with *Recordsets*.

Serializing Your *Recordset* to XML

The standard way to pass managed objects back and forth through XML Web services is by serializing the objects to and from XML (which is usually done for you). It is also possible to do this with *Recordset*s, although you have to handle the serialization yourself—sort of. Check out the following two Visual Basic .NET functions:

```
Function RsToString(ByVal rs As ADODB.Recordset) As String
   Dim stm As New ADODB.Stream()
   rs.Save(stm, ADODB.PersistFormatEnum.adPersistXML)
   Return stm.ReadText()
End Function

Function StringToRs(ByVal s As String) As ADODB.Recordset
   Dim stm As New ADODB.Stream()
   Dim rs As New ADODB.Recordset()
   stm.Open()
   stm.WriteText(s)
   stm.Position = 0
   rs.Open(stm)
   Return rs
End Function
```

By using the built-in ability to serialize an ADODB *Recordset* to an XML string, and by using the ADODB *Stream* object, you can freely pass *Recordset*s between tiers in your application. In addition you get the added benefits of using XML Web services, which is a versatile replacement for DCOM, especially when you need to communicate over the Internet and build loosely coupled distributed applications. Again, if you are interested in this topic, turn to Chapter 21 for details on how to implement the XML Web service itself.

Mapping ADO Objects to ADO.NET

Building on the previous discussion, let's now see how to adapt your existing code to use the new managed data objects in place of ADO. On the whole, it is a fairly simple process. We'll start by going over generic technology mappings to give you an idea of where to begin, and then we'll move on to some implementation details.

In general, the most direct mapping for ADO is to the OleDb namespace under System.Data, although if you expect to be using Microsoft SQL Server, the SqlClient providers will offer the best performance. An early beta version of the .NET Framework had an ADO provider namespace. It was decided, however, that this name would be too confusing, since the only real commonality was the use of the underlying OLEDB database providers. Therefore, the namespace was renamed to OleDb. This change was meant to emphasize to developers that the OleDb provider is to be used to access the various OLE DB database providers (as was ADO). Remember, however, that ADO is a COM-based interface to OLE DB data providers. The OleDb namespace contains managed interfaces to OLE DB data providers and actually lacks certain features found in ADO (hierarchical *Recordset*s come to mind).

For the purposes of this discussion, we will limit comparisons to those between ADO and OleDb. Keep in mind, however, that every example using classes from the OleDb namespace has a direct equivalent in the SqlClient namespace (which is specific to SQL Server). So if you are using a SQL Server database, moving between the two managed providers is often as easy as changing the class name's prefix from OleDb to Sql (that is, OleDbCommand becomes SqlCommand). Easy, right?

Connection and *Command* Objects

The *Connection* and *Command* objects still exist in ADO.NET. There are, however, significant differences between the ADO and ADO.NET classes. In ADO the *Connection* object not only represents your connection to a database but also can be used to execute database commands. In addition, other ADO objects (*Command* and *Recordset*) had *ActiveConnection* properties that accepted either an existing *Connection* object or a provider string, which would cause the object to create a new *Connection* object. This arrangement led to confusion because it was possible to execute database commands using all three of these objects, leading to wildly different coding practices from method to method (let alone from project to project).

ADO.NET is much more straightforward. *Connection* objects are provider specific (such as *OleDbConnection*) and are used only to represent a connection to a database. There is no way to use them to execute a statement against a database. That is the job of the *OleDbCommand* class. This class is comparable to the ADO *Command* class in that it can be used to execute run-time queries and stored procedures. Setting properties such as *CommandText* and *CommandType* should be instantly familiar to the seasoned ADO developer in this context. There is an important difference, however, in that it is possible to be explicit about the kind of query being performed. The simple *Execute* method of the ADO *Command* object has been replaced with four separate methods that indicate expectations regarding any given query. The methods specify what data type you want back, if any:

- **ExecuteReader** Execute a query and return a *DataReader* object.
- **ExecuteScalar** Execute a query and return a single value.
- **ExecuteXmlReader** Execute an XML query and return an *Xml-Reader* object.
- **ExecuteNonQuery** Execute a query with no results.

What this change means is that if you are using the *Execute* method on your ADO *Connection* objects to perform queries you will need to create an *OleDbCommand* object to do the actual work. This separation of functionality will hopefully lead to a more consistent coding style across applications. Although it can be painful to make the necessary adjustments in your code, we consider this an improvement over classic ADO. The following code illustrates what changes are needed to upgrade a simple ADO example:

```
' Visual Basic 6 Method Using ADO
Sub ExecuteSql( connStr As String, statement As String )
   Dim conn as New ADODB.Connection
   Dim rs as ADODB.RecordSet
   conn.Open connStr
   Set rs = conn.Execute( sqlStatement )
End Sub
```

```
' Visual Basic .NET Equivalent Method Using ADO.NET
Sub ExecuteSql( connStr As String, statement As String )
   Dim conn as New OleDbConnection( connStr )
   Dim cmd as New OleDbCommand( statement, conn )
   conn.Open()
   Dim dr As OleDbDataReader = cmd.ExecuteReader()
End Sub
```

*Recordset*s

Not all *Recordset*s are created equal. Depending on the settings you use, they can take several different forms, leading to various consequences for application performance. As we discussed in the previous section, ADO.NET seeks to separate the different types of functionality into specialized classes, both so that there is never any question as to the kind of data you are working with (disconnected, forward-only "firehose," and so on) and so that the classes can be optimized for a particular kind of data, rather than having to be general-purpose.

Forward-Only *Recordset*s

The forward-only *Recordset* is by far the most common type. It is the kind of *Recordset* that ADO generates by default (which is probably why it is the most common, performance advantages aside). Often this type of *Recordset* is referred to as the firehose *Recordset* because the data is read in a continuous stream, and once it has been read, it's gone.

ADO.NET has a direct equivalent to the firehose: the *DataReader* object. The *DataReader* is always forward only, and unlike a *Recordset*, a *DataReader* can never be disconnected. Also, it doesn't support random access under any circumstances. To simplify reading data from streams and to standardize on a common way to read data, the *DataReader* shares many interfaces and implementation details with other stream reader objects in the .NET Framework.

There are other implementation differences as well that can cause some initial confusion for developers. When a *DataReader* object is first returned, the record pointer starts before the first record in the data stream (instead of starting at the first record in the stream). A call to the *Read* method must be made before the first record is loaded, and successive calls to *Read* are necessary to continue moving through the data stream until the end is reached (signaled when *Read* returns *False*). Here is a quick comparison of the use of ADO firehose *Recordset*s and the ADO.NET *DataReader* (note the differences in the loop structures):

```
' Visual Basic 6 Forward-Only Recordset Access
Sub VB6RecordSetExample(connStr As String, statement As String)
   Dim conn As New ADODB.Connection
   Dim rs As ADODB.Recordset
   conn.Open (connStr)
```

(continued)

```
   Set rs = conn.Execute(statement)
   Do Until rs.EOF
      Dim str As String
      Dim i As Integer
      For i = 0 To rs.Fields.Count - 1
         str = str & rs.Fields(i).Value
      Next
      Debug.Print str
      rs.MoveNext
   Loop
End Sub

' Visual Basic .NET DataReader Example
Sub VBNETDataReaderExample(ByVal connStr As String, _
   ByVal statement As String)
   Dim conn As New OleDbConnection(connStr)
   Dim cmd As New OleDbCommand(statement, connStr)

   conn.Open()
   Dim reader As OleDbDataReader = cmd.ExecuteReader()

   While reader.Read()
      Dim str As String = "", i As Integer
      For i = 0 To reader.FieldCount - 1
         str &= reader(i)
      Next
      Debug.WriteLine(str)
   End While
End Sub
```

Disconnected *Recordset*s and Dynamic Cursors

A disconnected *Recordset* is retrieved in much the same way as a forward-only *Recordset*. So is a *Recordset* that uses a dynamic cursor. This is not the case with a *DataSet*, which requires, at least in the following example, a *DataAdapter* to handle the data population. Thus, while ADO had common methods for creating different kinds of *Recordset*s, ADO.NET has specific methods for creating different forms of data repositories.

```
' This is a Visual Basic 6 Dynamic Cursor Recordset Example
Function VB6DynamicRSExample(connStr As String, _
   statement As String) As Recordset
   Dim rs As New ADODB.Recordset
   rs.Open statement, connStr, adOpenDynamic

   Set rs = conn.Execute(statement)
   Set VB6DynamicRSExample = rs
   Set rs = Nothing
End Function
```

```
' This is a Visual Basic .NET DataSet Example
Function VBNETDataSetExample(ByVal connStr As String, _
   ByVal statement As String) As DataSet
   Dim adapter As New OleDbDataAdapter(statement, connStr)
   Dim ds As New DataSet()

   adapter.Fill(ds)
   Return ds
End Function
```

The other major difference between the disconnected *Recordset*s and *DataSet*s lies in how iteration is handled. The disconnected or dynamic *Recordset*s can use any of the *Move* functions (*Move*, *MoveFirst*, *MoveLast*, *MoveNext*, and *MovePrevious*) or can access rows using an ordinal reference or column name. A *DataTable* (contained within a *DataSet*) can be iterated using either an ordinal reference or *For Each*, as in the following example.

```
Dim row As DataRow
For Each row In ds.Tables(0).Rows
   ' Do stuff
Next

Dim i As Integer
For i = 0 To ds.Tables(0).Rows.Count - 1
   row = ds.Tables(0).Rows(i)
   ' Do other stuff
Next
```

Using *DataView*s

A *DataView* is an object that allows you to create multiple views of your data and that can be used for data binding. The concept is fairly simple, but its importance cannot be overstated. The flexibility it introduces is impressive because a *DataView* does not actually change the *DataTable* from which it is generated. Instead, it offers a filtered window to the data. The following code shows how you can use the DataView control to filter rows based on a column value (in this case, the FirstName and LastName columns):

```
Function DataViewTest()
   Dim adapter As New OleDbDataAdapter("Select * From Customers", _
   connStr)
   Dim ds As New DataSet()

   adapter.Fill(ds)
```

(continued)

```
    Dim view1 As DataView = ds.Tables("Customers").DefaultView
    Dim view2 As DataView = ds.Tables("Customers").DefaultView

    view1.RowFilter = "'LastName' Like 'O%'"
    view2.RowFilter = "'FirstName' Like 'E%'"

    Dim i As Integer

    Debug.WriteLine("All LastNames starting with 'O'")
    For i = 0 To view1.Count - 1
       Debug.WriteLine(view1(i)("LastName"))
    Next

    Debug.WriteLine("All FirstNames starting with 'E'")
    For i = 0 To view2.Count - 1
       Debug.WriteLine(view1(i)("FirstName"))
    Next

End Function
```

Data Binding

Earlier in this chapter, we demonstrated data binding using a *Recordset* (imported into a *DataTable*). However, that was only a small example of what is possible. This section looks at data binding in more detail, covering data binding in different environments, Web Forms controls vs. Windows Forms controls, and the kinds of objects that are supported for binding.

Binding to Windows Forms Controls

The .NET Framework brings a wide variety of controls to the developer. The Windows Forms controls are found in the System.Windows.Forms namespace, and the vast majority of them support data binding. In general, data binding to a control is far from challenging. It requires that you use an in-memory data store and can usually accept either a *DataTable* or a *DataSet*.

The mechanism is very simple. Take a look at the following example, using a *DataTable*:

```
Dim adapter As New OleDbDataAdapter("Select * From Customers", _
   connStr)
Dim dt As New DataTable()

' Import the query results into the DataTable
adapter.Fill(dt)

' Bind the DataTable to the DataGrid
DataGrid1.DataSource = dt
```

The *DataSet* works in almost exactly the same way:

```
Dim adapter As New OleDbDataAdapter("Select * From Customers",
connStr)
Dim ds As New DataSet()

' Import the query results into the DataSet
adapter.Fill(ds)

' Bind a DataTable from the DataSet to the DataGrid
DataGrid1.DataSource = ds.Tables("Customers")
```

Notice in this example that instead of passing the entire *DataSet* to the DataGrid, we've chosen to bind only a single table. This is often necessary with controls for which multiple sets of tables don't make sense. The DataGrid is not one of these controls, however. It is capable of a multitable display (try it) and likes the *DataSet* just fine.

Binding with Web Forms Controls

Data binding with Web Forms controls requires an additional step. Web Forms controls reside under the System.Web.UI namespace, and because they are destined to run only within the ASP.NET world, some of their behaviors are, by necessity, different from those of their System.Windows.Forms counterparts. Data binding is just one example of such a behavior. The Windows Forms controls examples apply to Web Forms controls, but with a caveat: you need to explicitly tell the control to bind to the data source you specified, as follows:

```
Dim adapter As New OleDbDataAdapter("Select * From Customers", _
   connStr)
Dim ds As New DataSet()

' Import the query results into the DataSet
adapter.Fill(ds)

' Bind a DataTable from the DataSet to the Web Form DataGrid
DataGrid1.DataSource = ds.Tables("Customers")

DataGrid1.DataBind()
```

Notice that this example is exactly the same as the Windows Forms *DataSet* example just given, with one additional line of code. Calling *DataBind* on the Web Form DataGrid forces the binding to occur—a necessary step, without which the control would not be bound to the specified data.

A Note About Performance

Performance is tricky. Tweaking an application to get it running under heavy load is an expensive and time-consuming task. From the standpoint of data access, however, it is possible to ensure a high level of performance as long as you follow these two coding conventions:

- Open connections as late as possible.
- Close them as soon as possible.

These two guidelines seem simple, yet their importance cannot be overstated. Connections to databases represent real physical resources on your machine, and hogging those resources will inevitably result in poor performance. For performance-sensitive applications, it is absolutely essential that you explicitly close all of your database connections as soon as you are done with them. This is necessary for two reasons. First, the connection will not automatically be closed for you as soon as the reference goes out of scope. Garbage collection is an indeterministic process, and it is impossible to guarantee the timing of a collection (unless you force it, which is a horribly expensive solution). Second, if a connection is left to be collected, it might not make it back into the connection pool, requiring the creation of a new connection. This adds more overhead to an application. Ideally, you want to get a connection from the connection pool (by calling *Open* on your connection object) and then return it to the pool immediately when you are done with it (by calling *Close*).

On another performance note, you should make sure that you choose your managed data provider carefully. Although the OleDb providers enable you to access virtually any database or data source (including SQL), Microsoft has also developed a SQL-specific managed provider that does not use OLE DB. In fact, the SqlClient namespace provides significant performance gains when accessing SQL Server–based databases because it is a network-level provider and communicates directly with a SQL server through low-level network access. The OleDb managed provider uses the OLE DB database providers, not only introducing an additional layer for OLE DB but also running it through the COM interop layer, further increasing overhead. When performance is critical, use only the OleDb provider for data sources other than Microsoft SQL Server (unless other companies have produced their own managed database providers by the time you read this).

Using *Open* and *Close* Effectively

If you need to do a lot of database work (such as making multiple sequential queries), it is important to make sure that you do not hold on to your database connections throughout the series of calls. If possible, you should release the connections between queries. This does add some additional overhead to your method (because it has to get connections from and return them to the connection pool more frequently), but it increases churn in the pool and ensures that connections will always be available to handle your application's needs. Take it from us—when your application runs out of connections, its performance craps out big time (yes, that is a technical term), and that's putting it mildly. The following example illustrates good database etiquette:

```
Function DBTest()
   Dim conn As New SqlConnection(connStr)
   Dim cmd As New SqlCommand("Select * From Customers", conn)

   ' Open the connection and execute the command
   conn.Open()
   Dim dr As SqlDataReader = cmd.ExecuteReader()

   While dr.Read()
      ' Do some processing
   End While

   ' Close the reader and the connection
   dr.Close()
   conn.Close()

   ' Get ready to do your next query
   cmd.CommandText = "Select * From Employees"

   ' Open the connection and execute the next command
   conn.Open()
   dr = cmd.ExecuteReader()

   While dr.Read()
      ' Do some other processing
   End While

   ' Close the reader and the connection
   dr.Close()
   conn.Close()
End Function
```

Conclusion

We covered a lot of ground in this chapter, from using ADO within your Visual Basic .NET application, to data binding and getting *Recordset*s to work with XML Web services, to moving to ADO.NET. Remember that you have many choices regarding how you implement your application. If your investment in ADO is too heavy at this time for you to consider upgrading to ADO.NET, you can still continue to improve and develop your application with an eye to switching to ADO.NET in the future. On the other hand, when you make the plunge into ADO.NET and the various managed providers, you will reap significant performance gains over pure ADO.