Chapter 31

Using ADO from ADO.NET
Although ADO.NET is more capable than ADO, ADO.NET is also substantially different from ADO. Migrating your existing applications to ADO.NET is no small undertaking. There’s no silver bullet that will quickly and painlessly convert all that ADO code into ADO.NET code. Although Microsoft does supply a migration wizard with Visual Studio.NET that’s designed to migrate your existing COM applications to the .NET Framework, you only have to try to use it a couple of times before discovering that it leaves all of the heavy lifting to you. The reality is that if you have a significant number of COM-based ADO applications running, in all likelihood you’re going to be keeping them for a while. Although rewriting applications may be quicker than writing them in the first place, a conversion from COM to .NET and ADO to ADO.NET definitely requires a full application rewrite. Most businesses already have projects underway that don’t allow them to drop everything and rewrite their existing applications for the sake of taking advantage of new technology. One way of beginning to merge your existing code into the .NET Framework is by using the .NET Framework’s COM interoperability functions to call COM-based ADO from .NET applications. In this chapter, you’ll see how you can incorporate ADO into your new .NET applications. The first part of this chapter illustrates how to import the msado15.dll, which contains ADO, into your .NET application. The second part of this chapter shows how to use the ADO Recordset object. First, by populating and navigating through the recordset, and then by using an ADO Recordset object to load an ADO.NET DataSet. The chapter concludes by discussing some of the important ADO–ADO.NET coexistence issues that you need to be aware of when using ADO from the .NET Framework.

**Importing ADO into the .NET Framework**

Although ADO and ADO.NET are very different, they do have some overall similarities. They both use a Connection object to connect to the data source. They both issue commands to the database and are capable of executing stored procedures and retrieving resultsets. And they both provide mechanisms for navigating the returned results as well as binding them to the objects in the client application. For ADO, the underlying connection to the data source is made using an OLE DB Provider. The ADO.NET equivalent to this is the .NET Data Provider. The ADO Connection and Command objects are essentially equivalent to the ADO.NET OleDbConnection and OleDbCommand objects if you’re using the .NET Framework Data Provider for OLE DB. In the ADO framework, resultsets are surfaced using the ADO Recordset object, which is closest to the ADO.NET DataTable. However, the new ADO.NET DataTable provides a host of functionality that isn’t available in the ADO Recordset object. Using ADO from a .NET application bears a strong resemblance to using ADO.NET, and coding those ADO calls will definitely be familiar territory. The first step in using ADO for the .NET Framework is to import the msado15.dll, which contains the COM-based ADO functionality, into your .NET application as is described in the following section.
Referencing the msado15.dll in Visual Studio.NET

In order to use ADO from a .NET Framework application, you must first have a .NET class wrapper for the ADO COM library. .NET applications aren’t able to directly call COM objects like ADO. Instead, .NET assemblies can only call other .NET assemblies. To call a COM object from a .NET application, you need to use the .NET System.Runtime.InteropServices namespace in your application. This namespace provides the classes required to interface with the COM objects. Fortunately, you don’t need to do this manually for the ADO COM library. Microsoft has provided a COM InterOp assembly for the ADO msado15.dll called ADODB, which by default is located under the C:\Program Files\Microsoft.NET\Primary Interop folder.

In spite of its name, the msado15.dll isn’t ADO 1.5. All of the recent versions of ADO have been contained in the msado15.dll dynamic link library.

In order to use this assembly in your application, you must first create a reference for it in your Visual Studio project. To add a reference to your Visual Studio.NET project, you must first open up the Project menu and then select the Add Reference option. The Add References dialog box shown in Figure 31-1 will be displayed.

To add a new reference for the ADODB assembly, scroll through the list of components until you see the ADODB component. Select the ADODB component from the list by clicking it then clicking Select. As you can see in Figure 31-1 the ADODB component name will be copied into the Selected Components list shown at the bottom of the Add Reference dialog box. When you’ve finished selecting components, clicking OK adds a reference to the selected components to your Visual Studio.NET project.

Manually Creating the COM InterOp Assemblies

If you can’t locate the ADODB assembly on your system, or if you have other COM objects you’ve created that you want to make use of in your .NET applications, you can use the tlbimp (Type Library Importer) utility that Microsoft has provided with the .NET Framework. The tlbimp.exe program allows you to import COM DLLs into .NET assemblies. You can run the tlbimp.exe from the Visual Studio Command Console as follows:

tlbimp.exe "C:\Program Files\Common Files\System\ado\msado15.dll"

Running the tlbimp.exe program with this argument will result in the creation of the ADODB.DLL in the C:\Program Files\Microsoft.NET\Primary Interop folder. After creating the ADODB.DLL, you can then add a reference to the ADODB component using the steps described in the preceding section.
Importing the ADODB DLL

After you've created a .NET assembly for the COM object you want to use in your .NET application, you can optionally import that assembly into your .NET application. Importing the assembly into the application isn't a requirement. However, doing so allows you to refer to the objects in your application without needing to use the fully qualified name. For instance, the ADODB component contains the Connection, Recordset, and Command objects. Without adding an import directive, you would need to reference each component using its fully qualified name: ADODB.Connection, ADODB.Recordset, and ADODB.Command. When an import directive is added to your project, you have the option to drop the ADODB qualifier and refer to the objects using their base name much like in standard ADO. In this case: Connection, Recordset, and Command.

To add an import directive for the ADODB.DLL assembly to a VB.NET project, you would add the following code to the declarations section of your source file:

```vbnet
Imports ADODB
```
To add an import directive for the ADODB assembly to a C# project, the code would appear as follows:

```csharp
using ADODB;
```

### Using an ADO Recordset

After adding an import directive to your code, you’re ready to begin using the different ADO objects contained in the ADODB assembly. The first thing that you’ll probably want to do is to retrieve data from the data source and populate an ADO Recordset object. The Recordset is the primary ADO object that’s used to retrieve data. The following ADORecordset subroutine illustrates opening an ADO Recordset object that selects those rows in the Customers table of the Northwind database where the Country column contains the value of Denmark:

```csharp
Private Sub ADORecordset(ByVal sServer As String, ByVal sDB As String)
    Dim ADOcn As Connection = New ADODB.Connection()
    Dim ADOrs As Recordset = New ADODB.Recordset()
    Dim ADOfld As Field

    ADOcn.Open("PROVIDER=SQLOLEDB;DATA SOURCE=" & sServer & ";INITIAL CATALOG=" & sDB & ";INTEGRATED SECURITY=SSPI;")
    ' Create a recordset
    Dim sSQL As String = "SELECT CustomerID, CompanyName, " & "Country FROM Customers WHERE Country = 'Denmark'"
    ADOrs.Open(sSQL, ADOcn, ADODB.CursorTypeEnum.adOpenForwardOnly, ADODB.LockTypeEnum.adLockReadOnly, 0)
    'Setup column heading in the list
    lstResults.Items.Add(ADOrs.Fields("CustomerID").Name & vbTab & "CompanyName" & vbTab & "Country"
    ' The recordset can't be bound so read one row at a time
    Do Until ADOrs.EOF
        lstResults.Items.Add(ADOrs.Fields("CustomerID").Value & vbTab & "CompanyName" & vbTab & "Country"
        ADOrs.MoveNext()
    Loop
    ' Close the Recordset
```
The C# version of the ADORecordset subroutine is shown here:

```csharp
private void ADORecordset(string sServer, string sDB)
{
    ADODB.Connection ADOcn = new ADODB.Connection();
    ADODB.Recordset ADOrs = new ADODB.Recordset();
    ADOcn.Open("PROVIDER=SQLOLEDB;DATA SOURCE=" + sServer + ";INITIAL CATALOG=" + sDB + ";INTEGRATED SECURITY=SSPI;", "", ",**,=-1");
    // Create a recordset
    string sSQL = "SELECT CustomerID, CompanyName, Country " + 
    + "FROM Customers WHERE Country = 'Denmark';";
    ADOrs.Open(sSQL, ADOcn, ADODB.CursorTypeEnum.adOpenForwardOnly, 
    ADODB.LockTypeEnum.adLockReadOnly, 0);
    // Setup column heading in the list
    lstResults.Items.Add(ADOrs.Fields["CustomerID"].Name + '\t' + 
    + ADOrs.Fields["CompanyName"].Name + '\t' + '\t' + 
    + ADOrs.Fields["Country"].Name);
    // The recordset can't be bound so read one row at a time
    while (!ADOrs.EOF)
    {
        // Add the Recordset fields to a list and get the next row
        lstResults.Items.Add(\n            (string)ADOrs.Fields["CustomerID"].Value + '\t' + 
            '\t' + (string)ADOrs.Fields["CompanyName"].Value + 
            '\t' + '\t' + (string)ADOrs.Fields["Country"].Value);
        ADOrs.MoveNext();
    }
    // Close the Recordset
    ADOrs.Close();
    ADOcn.Close();
}
```
In these examples, the sServer variable contains the name of a SQL Server system and the sDB variable contains the name of Northwind—setting the default database to the Northwind sample database included with SQL Server.

At the beginning of this subroutine, you can see where instances of an ADO Connection and Recordset object named ADOcn and ADOrs are created. Like its name implies, the ADO Connection object is used to open a connection to the data source, whereas the ADO Recordset object is used to send a SQL select statement to the target database and hold the resultset that is returned. The Recordset object can provide navigation, data access, and update capabilities. Next, the ADOcn Connection object’s Open method is used to initiate an active connection with the database system. The first parameter of the Open method takes an OLE DB connection string. In this case, the Provider keyword indicates that the OLE DB Provider for SQL Server will be used. The Data Source keyword specifies that the target database is identified by the sServer variable. The Initial Catalog keyword sets the default database to the database name contained in the sDB variable. Finally, the Integrated Security keyword indicates that Windows authentication will be used to connect to the target database. For more information about OLE connection strings, you can refer to Chapter 6.

In C#, you need to explicitly define all of the parameters that are required by a given method. VB.NET does not have this requirement. That’s why the Open method in the C# version of this code requires three extra parameters. In the case of the Connection object’s Open method, these parameters are optionally used to pass in the user ID, password, and options flag. The user ID and password are passed in as empty strings; the options value of –1 in the fourth parameter indicates that the connection is opened synchronously.

After a connection has been opened to the data source, a string variable named sSQL is assigned with a SQL Select statement that will retrieve the CustomerID, CompanyName, and Country column from the Customers table where the contents of the Country column contains the value of Denmark. Then the Open method of the ADOrs Recordset object is used to populate the Recordset object. The first parameter of the ADOrs Recordset object’s Open method accepts the SQL statement that will be sent to the data source. The second parameter accepts the name of an active ADO Connection object—in this case, the ADOcn object that was just opened. The third and fourth parameters indicate the type of connection that will be opened. The value of CursorTypeEnum.adForwardOnly indicates that the Recordset object will use a forward-only cursor that provides high performance but is not scrollable. The value of LockTypeEnum.adLockReadOnly in the fourth parameter indicates that the Recordset will be read-only, so no updates are allowed. The value of 0 in the fourth parameter indicates that the type of command used in the first parameter is a SQL statement.

The next section of code outputs the contents of the ADOrs Recordset object to a .NET List Box named lstResults. An ADO Recordset object cannot be directly bound to a .NET data-aware interface component, so the contents of the Recordset must be
manually parsed. First, the column headings of the columns retrieved in the Recordset object are written to the top line in the lstResults ListBox. You add items to the .NET Framework’s ListBox by invoking the Add method that’s supplied by the Items collection of the lstResults ListBox object. In this case, the ListBox’s Add method is passed a string parameter that concatenates the value contained in the Name property of each ADO Field object that’s contained in the Fields collection. You should note that the name of each column is used as the index to identify the desired ADO Field object in the Fields collection. Tab characters are inserted into the string just to help the column heading line up better within the ListBox.

The .NET Framework does not provide support for using default properties. This means that when using ADO objects, you cannot use the Bang notation (e.g., ADOrs!CustomerID) that was supported by VB 6.0 and VC++. Instead, you need to explicitly define the properties you want to work with.

Next, all of the row values in the Recordset are added to the ListBox by setting up a looping structure that is repeated until the ADOrs.EOF property contains the value of True—indicating the end of the rows in the Recordset. In this instance, the ListBox’s Add method is passed a string parameter that concatenates the value contained in the Value property of each ADO Field object. Again, Tab characters are inserted just to help the results line up better in the ListBox. Then the ADOrs Recordset object’s MoveNext method is used to move the pointer to the next row in the ResultSet.

At the end of this subroutine, you can see that the ADOrs Recordset object and the ADOcn Connection object are explicitly closed using each respective object’s Close method. Explicitly calling the Close method is an essential technique in .NET applications to ensure that the object resources are released.

Loading an ADO.NET DataSet from an ADO Recordset

One key step in migrating your ADO applications to ADO.NET is beginning to take advantage of the native ADO.NET objects. In addition to being able to use the native ADO objects such as the Recordset object from .NET applications, you can also create interfaces between these ADO objects and the native ADO.NET objects. The following example illustrates how you can load an ADO.NET DataSet from the data contained in an ADO Recordset object. This can be advantageous because the DataSet can be bound to other .NET interface objects like the DataGrid. In the following example, you can see how to open an ADO Recordset and then use the .NET DataAdapter’s Fill method to populate a DataSet using the ADO Recordset. Filling a DataSet with the contents of an ADO object is a one-way operation. In other words, once you move the data from the ADO Recordset object into the ADO.NET DataSet object, the Recordset object is closed, and all of the data operations such as navigation and updates are performed using the DataSet object.
Private Sub ADOLoadDataSet(ByVal sServer As String, ByVal sDB As String)
    Dim ADOcn As Connection = New ADODB.Connection()
    Dim ADOrs As Recordset = New ADODB.Recordset()
    ADOcn.Open("PROVIDER=SQLOLEDB;DATA SOURCE=" & sServer & 
    ";INITIAL CATALOG=" & sDB & ";INTEGRATED SECURITY=SSPI;")
    ' Create a recordset. Bear in mind that
    Dim sSQL As String = "SELECT * FROM Customers " & 
    "WHERE Country = 'USA'"
    ADOrs.Open(sSQL, ADOcn, CursorTypeEnum.adOpenForwardOnly, _
    LockTypeEnum.adLockReadOnly, 0)
    Dim da As New OleDbDataAdapter()
    Dim ds As New DataSet()
    ' Load the Recordset into a DataTable object
    ' Fill closes the Recordset upon completion
    da.Fill(ds, ADOrs, "ADOrsTable")
    ' Bind the DataTable to a grid
    grdResults.DataSource = ds.Tables("ADOrsTable")
    ADOcn.Close()
End Sub

The C# version of the ADOLoadDataset subroutine follows:

private void ADOLoadDataSet(string sServer, string sDB)
{
    ADODB.Connection ADOcn = new ADODB.Connection();
    ADODB.Recordset ADOrs = new ADODB.Recordset();
    string sSQL = "SELECT * FROM Customers WHERE Country = 'USA';
    ADOcn.Open("PROVIDER=SQLOLEDB;DATA SOURCE=" + sServer
    + ";INITIAL CATALOG=" + sDB + ";INTEGRATED SECURITY=SSPI; ",
    ",",",",-1);
    ADOrs.Open(sSQL, ADOcn, CursorTypeEnum.adOpenForwardOnly, 
    LockTypeEnum.adLockReadOnly, 0);  
    // Converts a Recordset into a DataTable object
    OleDbDataAdapter da = new OleDbDataAdapter();
    DataSet ds = new DataSet();
    da.Fill(ds, ADOrs, "ADOrsTable");
    // The recordset is implicitly closed and is unusable
    // Bind the DataTable to a grid
    grdResults.DataSource = ds.Tables["ADOrsTable"];  
    ADOcn.Close();
}
At the beginning of the ADOLoadDataSet subroutine, you can see where an ADO
Connection object named ADOcn and an ADORecordset object named ADOrs are created.
Next, the ADOcn Connection object’s Open method is used to initiate an active connection
with the database system. The first parameter of the Open method uses an OLE DB
connection string that sets the OLE DB Provider to SQLOLEDB, the SQL Server OLE
DB Provider. The database server and default database are set to the values contained
in the sServer and sDB variables, and the authentication method is set to use integrated
Windows security.

Next, a SQL statement that retrieves all of the columns from the Customers table
where the Country column contains the value USA is assigned to the sSQL variable. The
sSQL variable is then passed as the first parameter of the ADOrs Recordset object’s Open
method. The remaining parameters specify that the ADOrs Recordset object will be
forward-only and read-only.

After the ADOrs Recordset object is populated, the next two lines create an
OleDbDataAdpater object and a DataSet object named da and ds, respectively. Then
the OleDbDataAdapter’s Fill method is used to create a new DataTable object within the
DataSet. This overloaded instance of the OleDbDataAdapter’s Fill method accepts three
parameters. The first parameter identifies the DataSet that is the target of the Fill operation.
The second parameter identifies the ADO Recordset object that is the data source. And the
third parameter optionally names the DataTable object that will be created in the DataSet.
In this example, a DataTable named ADOrsTable will be created in the ds DataSet
based on the data in the ADOrs Recordset object. The Fill method implicitly closes the
ADOrs Recordset object after it completes—making it impossible to use the Recordset
object again unless it is explicitly reopened. Once the DataTable has been created, it’s
bound to a DataGrid object named grdResults and the ADOcn Connection object is
closed.

While the Recordset is closed following the completion of the OleDbDataAdapter’s Fill,
you can still post data changes back to the database from the DataSet via the DataAdapter.

### Updating Data with an ADO Recordset Object

The previous examples illustrated how to retrieve data into a .NET application using the
ADO Recordset object. However, the ADO Recordset object isn’t restricted to being used in
a read-only mode. In fact, it’s completely capable of performing all of the same functionality
from a .NET application as it is from a legacy COM application. In the next example, you’ll
see how you can create an updateable ADO Recordset object from a .NET application and
use it to add rows to a table. Here, the Recordset object is opened using an updateable
keyset cursor, which allows forward and backward scrolling, as well as bookmarking
rows for direct navigation.

In order to run the following example, you need to create the department table
using the following SQL:
CREATE TABLE [dbo].[Department] (
    [DepartmentID] [int] NOT NULL ,
    [DepartmentName] [char] (25)
) ON [PRIMARY]

The VB.NET code to update an ADO Recordset object follows:

Private Sub ADOUpdateRecordset(ByVal sServer As String, _
    ByVal sDB As String)
    Dim ADOcn As Connection = New ADODB.Connection()
    Dim ADOrs As Recordset = New ADODB.Recordset()
    Dim ADOfld As Field
    ADOcn.Open("PROVIDER=SQLOLEDB;DATA SOURCE=" & sServer _
        & ";INITIAL CATALOG=" & sDB & ";INTEGRATED SECURITY=SSPI;")
    ' Create a recordset
    Dim sSQL As String = "SELECT DepartmentID, DepartmentName" & _
        & "FROM Department"
    ADOrs.Open(sSQL, ADOcn, CursorTypeEnum.adOpenKeyset, _
        & "FROM Department"
    ADOrs.Open(sSQL, ADOcn, CursorTypeEnum.adOpenKeyset, _
        & "FROM Department"
    Dim i As Integer
    For i = 1 To 10
        ' Create anew buffer
        ADOrs.AddNew()
        ' Set the column values
        ADOrs.Fields("DepartmentID").Value = i
        ADOrs.Fields("DepartmentName").Value = _
            "Department" & CStr(i)
        ' Add the row
        ADOrs.Update()
    Next
    ' Display the updated recordset in a grid
    Dim da As New OleDbDataAdapter()
    Dim ds As New DataSet()
    ' Load the Recordset into a DataTable object
    ' Fill closes the Recordset upon completion
    da.Fill(ds, ADOrs, "ADOrsTable")
    ' Bind the DataTable to a Grid
    grdResults.DataSource = ds.Tables("ADOrsTable")
    ADOcn.Close()
End Sub
The C# version of the ADOUpdateRecordset subroutine is shown here:

```csharp
private void ADOUpdateRecordset(string sServer, string sDB)
{
    ADODB.Connection ADOcn = new ADODB.Connection();
    ADODB.Recordset ADOrs = new ADODB.Recordset();
    ADOcn.Open("PROVIDER=SQLOLEDB;DATA SOURCE=" + sServer + ";INITIAL CATALOG=" + sDB + ";INTEGRATED SECURITY=SSPI;",
    ","","",1);
    // Create a recordset
    string sSQL = "SELECT DepartmentID, DepartmentName " + "FROM Department";
    // For C#, make sure you set up a client cursor
    ADOrs.CursorLocation = CursorLocationEnum.adUseClient;
    ADOrs.Open(sSQL, ADOcn, CursorTypeEnum.adOpenKeyset,
    LockTypeEnum.adLockOptimistic, -1);
    int i;
    for (i = 1; i <= 10; i++)
    {
        // Create a new buffer
        // C# requires you to supply the AddNew parameters
        ADOrs.AddNew("DepartmentID", i);
        ADOrs.Update("DepartmentName", "Department" + i);
    }
    // Display the updated recordset in a grid
    OleDbDataAdapter da = new OleDbDataAdapter();
    DataSet ds = new DataSet();
    // Load the Recordset into a DataTable object
    // Fill closes the Recordset upon completion
    da.Fill(ds, ADOrs, "ADOrsTable");
    // Bind the DataTable to a Grid
    grdResults.DataSource = ds.Tables["ADOrsTable"]; // Close the ADO connection
    ADOcn.Close();
}
```

At the beginning of the ADOUpdateRecordset subroutine, you can see where the ADO Connection and Recordset objects named ADOcn and ADOrs are created. Next, the ADOcn Connection object’s Open method is used to initiate a connection to the database system identified in the sServer variable that’s passed into the beginning of this subroutine. From the connection string, you can see that this connection uses the SQL Server OLE DB Provider and the database contained in the sDB variable will be used as the initial catalog (aka. default database) and the connection is set to use integrated Windows security.
After a connection has been opened, the sSQL variable is assigned a SQL Select statement that will retrieve all the DepartmentID and DepartmentName columns from the Department table that was built earlier in this chapter.

Next, the ADOrs Recordset object’s Open method is used to populate the Recordset object. As in the earlier examples, the first parameter contains the SQL statement that’s sent to the target database; the second parameter contains an active ADO Connection object. In this case, note that the third parameter of the Recordset’s Open method sets the Recordset’s cursor type to CursorTypeEnum.adOpenKeyset. Unlike the forward-only cursor shown in the previous examples, the keyset cursor supports both resultset navigation and updating. The fourth parameter uses the LockTypeEnum.Optimistic value to set the locking type to optimistic—which essentially means that no lock is held on the data in the Recordset until immediately before any updates are sent to the database. These values in the third and forth parameters allow the ADO Recordset object to update the data in the Recordset, and those updates will be immediately propagated back to the data source.

One key difference between the VB.NET routine and C# routine is that C# requires you to explicitly set the cursor location to a client cursor using the CursorLocation.adUseClient enumerator for the Recordset object’s CursorLocation property.

Once the Recordset has been populated with a resultset from the target database, a For-Next loop is used to add 10 rows to the department table. The first operation within the For-Next loop is the invocation of the ADOrs Recordset object’s AddNew method. The AddNew method essentially allocates a buffer within the Recordset object for a new row. Next, the Value property of the two Field objects contained in the ADOrs.Fields collection are assigned new values. For the sake of simplicity, this example assigns the DepartmentID column the value of the loop counter contained in variable i, and the string containing the name “Department” is concatenated with the value of the loop counter and assigned to the DepartmentName column. At the bottom of the For-Next loop, the ADOrs Recordset object’s Update method is called to add the new row to the Recordset, which also results in the row being added to the data source.

C# doesn’t allow the use of default parameters, so you need to explicitly define each of the parameters. In this example, the AddNew method of the C# ADOUpdateRecordset subroutine supplies the two optional parameters with the values used for the first column. Here, the string “DepartmentID” identifies the column name in the new Recordset buffer, and the value of variable i supplies the value for the column. Subsequent usage of the Update method supplies the remaining column values.

The next section of code displays the updated Recordset object in a grid. First, an OleDbDataAdapter object and a DataSet object named da and ds are instantiated. Then the OleDbDataAdapter’s Fill method is used to create a new DataTable object within the DataSet named ADOrsTable, using the data in the ADOrs Recordset object. The Fill method implicitly closes the ADOrs Recordset object after it completes. Once the DataTable has been created, it’s bound to a DataGrid object named grdResults and the ADOcn Connection object is closed.
Using an ADO Command Object

The previous examples all illustrated using the ADO Recordset object; however, that’s not the only ADO object that’s supported within the .NET Framework. All of the ADO objects can be used. To round out the ADO examples, the following example illustrates using the ADODB assembly to invoke the ADO Command object. In this case, the ADO Command object will be used to execute a parameterized query that updates the data in the department table that was used in the earlier example. In this section of code, you’ll see how to create an ADO Command object and add ADO Parameter objects to it, as well as how to set the Parameter object’s attributes:

```vbnet
Private Sub ADOCommand(ByVal sServer As String, ByVal sDB As String)
    Dim ADOcn As Connection = New ADODB.Connection()
    Dim ADOrs As Recordset = New ADODB.Recordset()
    Dim ADOcmd As Command = New ADODB.Command()
    ADOcn.Open("PROVIDER=SQLOLEDB;DATA SOURCE=" & sServer & ";INITIAL CATALOG=" & sDB & ";INTEGRATED SECURITY=SSPI;")
    With ADOcmd
        .ActiveConnection = ADOcn
        .CommandText = "UPDATE department SET DepartmentName=?" & 
        "WHERE DepartmentID=?"
        .CreateParameter(, DataTypeEnum.adChar, 
        ParameterDirectionEnum.adParamInput, 25)
        .CreateParameter(, DataTypeEnum.adInteger, 
        ParameterDirectionEnum.adParamInput)
    End With
    Dim i As Integer
    For i = 1 To 10
        ADOcmd.Parameters(0).Value = "Updated DepartmentName:" & 
        CStr(i)
        ADOcmd.Parameters(1).Value = CStr(i)
        ADOcmd.Execute(, , -1)
    Next
    ADOrs.Open("SELECT * FROM department", ADOcn, 
    CursorTypeEnum.adOpenForwardOnly, 
    LockTypeEnum.adLockReadOnly, 0)
    ' Display the updated recordset in a grid
    Dim da As New OleDbDataAdapter()
    Dim ds As New DataSet()
    ' Load the Recordset into a DataTable object
    ' Fill closes the Recordset upon completion
```
da.Fill(ds, ADOrs, "ADOrsTable")
' Bind the DataTable to a Grid
grdResults.DataSource = ds.Tables("ADOrsTable")
ADOcn.Close()

The C# version of the ADOCommand subroutine follows:

private void ADOCommand(string sServer, string sDB)
{
    ADODB.Connection ADOcn = new ADODB.Connection();
    ADODB.Recordset ADOrs = new ADODB.Recordset();
    ADODB.Command ADOcmd = new ADODB.Command();
    ADOcn.Open("PROVIDER=SQLOLEDB;DATA SOURCE=" + sServer +
        ";INITIAL CATALOG=" + sDB + ";INTEGRATED SECURITY=SSPI;",
    ","","",-1);
    // Set the Command object properties
    ADOcmd.ActiveConnection = ADOcn;
    ADOcmd.CommandText =
        "UPDATE department SET DepartmentName = ? " +
        "WHERE DepartmentID = ?;";
    ADOcmd.CreateParameter("", DataTypeEnum.adChar,
        ParameterDirectionEnum.adParamInput, 25,"");
    ADOcmd.CreateParameter("", DataTypeEnum.adInteger,
        ParameterDirectionEnum.adParamInput, 4, 0);
    // Update 10 rows
    int i;
    object nRowsAffected;
    object o = Missing.Value;
    for (i = 1; i <= 10; i++)
    {
        ADOcmd.Parameters[0].Value = "Updated DepartmentName:" + i;
        ADOcmd.Parameters[1].Value = i;
        ADOcmd.Execute(out nRowsAffected, ref o, 1);
    }
    // Open a recordset
    ADOrs.Open("SELECT * FROM department", ADOcn,
        CursorTypeEnum.adOpenForwardOnly,
        LockTypeEnum.adLockReadOnly, 0);
    // Display the updated recordset in a grid
    OleDbDataAdapter da = new OleDbDataAdapter();
DataSet ds = new DataSet();
// Load the Recordset into a DataTable object
// Fill closes the Recordset upon completion
da.Fill(ds, ADOrs, "ADOrsTable");
// Bind the DataTable to a Grid
grdResults.DataSource = ds.Tables["ADOrsTable"];
// Close the ADO connection
ADOcn.Close();
}

At the top of the ADOCommand subroutine, you can see where the ADO Connection,
Recordset, and Command objects named ADOcn, ADOrs, and ADOcmd are created.
Next, the ADOcn Connection object’s Open method is used to initiate a connection to
the database system identified in the sServer variable. Like the earlier examples, this
connection uses the SQL Server OLE DB Provider to connect to the database identified
in the sDB variable using integrated security.

After a connection has been opened, the ADOcmd Command object’s properties are
assigned. The VB.NET version uses a With statement for this; the C# version, which doesn’t
support the With structure, uses simple assignment statements. First, the ActiveConnection
property is set to the ADOcn Connection object. Next, the CommandText property is
assigned a SQL Update statement. In this example, note that the Update statement is
using two parameters indicated with the “?” character. The “?” is used as a parameter
marker and will be substituted with the value in an associated Parameter object at runtime.
Next, the ADOcmd.CreateCommand method is used to create two Parameter objects that
are added to the ADOcmd Command object. The first parameter accepts a string that
optionally names the parameter. The second parameter is used to identify the Parameter
object’s data type. The value of DataTypeEnum.adChar indicates a character value. The
third parameter of the CreateParameter method specifies the direction of the parameter. In
this case, the value of ParameterDirectionEnum.adParamInput shows that both Parameter
objects will be input parameters. The fourth parameter shows the size of the parameter;
the fifth parameter provides an optional default value.

Next, a For-Next loop is used to update the 10 rows in the department table that were
added in an earlier example. With the For-Next loop, the first statement sets the first
Parameter object, identified with an index of 0, to a new string value starting with the
characters “Updated DepartmentName” concatenated with the loop counter. The second
statement sets the Parameter object, identified with the index of 1, to the value of the
loop counter. After values have been assigned to the Parameter objects, the ADOcmd
.Execute method is called to actually perform the update. You might notice that the
VB.NET version does some of the work for you by automatically handling the three
optional parameters of the ADOcmd.Execute method, whereas the C# version requires
that you explicitly provide variables for the parameters.

In the previous C# examples, you saw how to use the various ADO functions that required
optional parameters by explicitly supplying the parameter values in your code. Note that
this example is different. In this case, the System.Reflection namespace is added to the
Visual Studio.NET project and the Missing.Value member of that namespace is used to
handle the optional parameter. In this C# code, an object named o is created and assigned
the Missing.Value property. This allows the object to be passed to the ADO Execute method
without explicitly assigning it a value. This technique allows C# code to work much more
like VB.NET, which has the capability to supply default parameters.

The last section of code then opens a Recordset object and uses the OleDbDataAdapter’s
Fill method to load a DataTable with the resultset from the Recordset. Then the DataTable
is bound to a DataGrid named grdResults and displayed to the user, and the ADO
Connection object is closed.

**ADO and ADO.NET Coexistence Issues**

As you’ve seen in the previous examples, the .NET Framework is fully capable of utilizing
COM objects like the ADO object library. And, in fact, doing so is very much like using
ADO from the earlier COM-based languages such as VB 6.0 and VC++. However, as you
probably noted from some of the earlier examples, the usage isn’t identical—especially if
you’re writing in C#. You need to look out for several differences, including the following:

- .NET languages such as VB.NET and C# can’t use default properties.
- .NET languages require the use of parentheses when calling methods.
- C# explicitly requires you to supply values for all of the method parameters.
- Arguments passed to subroutines default to ByVal rather than ByRef.

**Summary**

Although ADO.NET is definitely the preferred method of accessing data using the .NET
Framework, the difference between .NET applications and COM-based applications, and
the difference between ADO.NET and ADO, combine to make migration a long-term
process, thereby ensuring that ADO code will be around for a long time to come. In this
chapter, you saw how you could use ADO with ADO.NET applications as one method
to begin moving your existing ADO code to ADO.NET.

For more information about ADO, you can refer to the following resources: