12

CHAPTER

Connecting to a database

In This Chapter

Using the ADO Connection object

Working with the ADO Error object

Introducing the ADO Errors collection

Connecting to your database server

Analyzing errors

In this chapter, I'm going to discuss the ADO `Connection` object in depth. Also, since the `Errors` collection and the `Error` object are tightly coupled with the `Connection` object, I'm going to cover them also. Access to a data provider is managed using the ADO `Connection` object. Thus, every program that uses a database server must include at least one `Connection` object. Unless you are using multiple data providers or accessing multiple database servers, one `Connection` object is sufficient.

The Connection Object

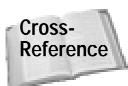
The `Connection` object is used to maintain a connection to a data source. It can be implicitly created through the `Command` and `Recordset` objects, or you can create an instance of the `Connection` object and share it among multiple `Command` and `Recordset` objects.

Connection object properties

Table 12-1 lists the properties associated with the `Connection` object.

Table 12-1
Properties of the Connection Object

Property	*Description*
`Attributes`	A `Long` value containing the transaction attributes for a connection (see Table 12-2). Note that not all data providers support this property.
`CommandTimeout`	A `Long` containing the maximum number of seconds for the command to execute before an error is returned. Default is 30 seconds.
`ConnectionString`	A `String` containing the information necessary to connect to a data source.
`ConnectionTimeout`	A `Long` containing the maximum number of seconds that the program should wait for a connection to be opened before returning an error. Default is 15 seconds.
`CursorLocation`	A `Long` containing the default location of the cursor service (see Table 12-3). This value will automatically be inherited by the `Recordset` object using this `Connection` object.
`DefaultDatabase`	A `String` containing the name of the default database.
`Errors`	An object reference to an `Errors` collection.
`IsolationLevel`	A `Long` containing the level of transaction isolation (see Table 12-4).
`Mode`	A `Long` containing the available permissions for modifying data (see Table 12-5).
`Properties`	An object reference to a `Properties` collection containing provider-specific information.
`Provider`	A `String` containing the name of the data provider. This value may also be set as part of the `ConnectionString`.
`State`	A `Long` describing the current state of the `Command` object. Multiple values can be combined to describe the current state (see Table 12-6).
`Version`	A `String` containing the current ADO version number.



See Chapter 22, “Integrating XML with Internet Explorer 5,” for more information about SQL Server connection strings; Chapter 26, “Overview of Oracle 8i,” for more information about Oracle 8i connection strings; and Chapter 30, “Creating Jet Database Objects,” for more information about Jet connection strings.

Table 12-2
Values for Attributes

Constant	*Value*	*Description*
adXactCommitRetaining	131072	Calling CommitTrans automatically starts a new transaction.
adXactAbortRetaining	262144	Calling RollbackTrans automatically starts a new transaction.

Table 12-3
Values for CursorLocation

Constant	*Value*	*Description*
adUseNone	1	Does not use cursor services (obsolete).
adUseServer	2	Uses the server-side cursor library.
adUseClient	3	Uses the client-side cursor library.

Table 12-4
Values for IsolationLevel

Constant	*Value*	*Description*
adXactUnspecified	-1	The provider is using a different isolation level than specified.
adXactChaos	16	Pending changes from more highly isolated transactions can't be overwritten.
adXactBrowse	256	Can view uncommitted changes in other transactions.
adXactReadUncommitted	256	Same as adXactBrowse.
adXactCursorStability	4096	Can view only committed changes in other transactions.
adXactReadCommitted	4096	Same as adXactCursorStability.
adXactRepeatableRead	65536	Can't view changes in other transactions until you Requery the Recordset object.

Continued

Table 12-4 *(continued)*

Constant	*Value*	*Description*
`adXactIsolated`	1048576	Transactions are conducted in isolation from all other transactions.
`adXactSerializable`	1048576	Same as `adXactIsolated`.

Table 12-5
Values for Mode

Constant	*Value*	*Description*
`adModeUnknown`	0	Permissions are not set or can't be determined.
`adModeRead`	1	Requests read permission.
`adModeWrite`	2	Requests write permission.
`adModeReadWrite`	3	Requests read/write permission.
`adModeShareDenyRead`	4	Prevent other connections from opening with read permissions.
`adModeShareDenyWrite`	8	Prevent other connections from opening with write permissions.
`adModeShareExclusive`	12	Prevent other connections from opening.
`adModeShareDenyNone`	16	Permit other connections with any permissions.
`adModeRecursive`	32	Used with `adModeShareDenyRead`, `adModeShareDenyWrite`, and `adModeShareDenyNone` to propagate sharing restrictions to all sub-records of the current `Record`.

Connection object methods

The `Connection` object has many methods that allow you to manage your connection to a data source.

Table 12-6
Values for State

Constant	*Value*	*Description*
`adStateClosed`	0	The `Command` object is closed.
`adStateOpen`	1	The `Command` object is open.
`adStateConnecting`	2	The `Command` object is connecting to the database.
`adStateExecuting`	4	The `Command` object is executing.
`adStateFetching`	8	Rows are being retrieved.

Function BeginTrans () As Long

The `BeginTrans` method marks the beginning of a transaction. The return value corresponds to the nesting level of the transaction. The first call to `BeginTrans` will return a one. A second call to `BeginTrans`, without a call to `CommitTrans` or `RollbackTrans`, will return two.

Sub Cancel ()

The `Cancel` method is used to terminate an asynchronous task started by the `Execute` or `Open` methods.

Sub Close ()

The `Close` method closes the connection to the data provider. It will also close any open `Recordset` objects and set the `ActiveConnection` property of any `Command` objects to `Nothing`.

Sub CommitTrans ()

The `CommitTrans` method ends a transaction and saves the changes to the database. Depending on the `Attributes` property, a new transaction may automatically be started.

Function Execute (CommandText As String, [RecordsAffected], [Options As Long = -1]) As Recordset

The `Execute` method is used to execute the specified command. A `Recordset` object will be returned as the result of the function, which will contain any rows returned by the command.

`CommandText` is a `String` containing an SQL Statement, stored procedure, table name, or other data provider-specific command to be executed.

`RecordsAffected` optionally returns a `Long` value with the number of records affected by the command.

`Options` optionally passes a combination of the values specified in Table 11-5 found in the section on the `Command` object.

Sub Open ([ConnectionString As String], [UserID As String], [Password As String], [Options As Long = -1])

The `Open` method initializes the `Connection` object by establishing a connection to a data provider.

`ConnectionString` is a `String` value containing the same connection information found in the `ConnectionString` property. The value in this parameter will override the value in the property.

`UserID` contains a `String` value with the UserID needed to access the database. This value will override any UserID information included in the `ConnectionString` parameter or property.

`Password` contains a `String` value with the password associated with the specified UserID. This value will override any password information included in the `ConnectionString` parameter or property.

`Options` optionally passes one of the values specified in Table 12-7. If you specify `adAsyncConnect`, the `ConnectComplete` event will be fired when the connection process has finished.

Table 12-7
Values for Options

Constant	*Value*	*Description*
`adConnectUnspecified`	-1	Opens the connection synchronously. Default.
`adAsyncConnect`	16	Opens the connection asynchronously.

Function OpenSchema (Schema As SchemaEnum, [Restrictions], [SchemaID]) As Recordset

The OpenSchema method returns database information from the data provider. A Recordset object will be returned as the result of the function, which will contain any rows returned by the command.

Schema is an enumerated value specifying the type of information to be returned.

Restrictions contains an array of query constraints.

SchemaID optionally contains a GUID for a provider-schema query not defined in the OLE DB specification. This parameter is only used when the Schema parameter is set to adSchemaProviderSpecific.

Note

So you want to write a database utility: The OpenSchema method can be used to perform nearly forty different queries against a database catalog. Each query returns a different Recordset containing the relevant information. Since this information is extremely complex and not generally used by database programmers, you should reference the OLE DB documentation for detailed information about this method.

Sub RollbackTrans ()

The RollbackTrans method ends a transaction and discards any changes to the database. Depending on the Attributes property, a new transaction may automatically be started.

Connection object events

The Connection object contains events that allow you to intercept status information and determine error conditions while you have a connection to your data source.

Event BeginTransComplete (TransactionLevel As Long, pError As Error, adStatus As EventStatusEnum, pConnection As Connection)

The BeginTransComplete is called after the BeginTrans method has finished running in asynchronous mode.

TransactionLevel is a Long value containing the new transaction level.

pError is an object reference to an Error object if the value of adStatus is set to adStatusErrorsOccured.

adStatus is a Long value that contains one of the status values listed in Table 12-8.

pConnection is an object reference to the Connection object associated with the BeginTrans method.

Table 12-8
Values for adStatus

Constant	*Value*	*Description*
adStatusOK	1	The operation completed successfully.
adStatusErrorsOccured	2	The operation failed. Error information is in pError.
adStatusCantDeny	3	The operation can't request the cancellation of the current operation.
adStatusCancel	4	The operation requested that the operation be canceled.
adStatusUnwantedEvent	5	Setting the value of the adStatus parameter to this value while in the event will prevent subsequent events from being fired.

Event CommitTransComplete (pError As Error, adStatus As EventStatusEnum, pConnection As Connection)

The CommitTransComplete is called when the CommitTrans method has finished running in asynchronous mode.

pError is an object reference to an Error object if the value of adStatus is set to adStatusErrorsOccured.

adStatus is a Long value that contains one of the status values listed in Table 12-8 in the BeginTransComplete event.

pConnection is an object reference to the Connection object associated with the CommitTrans method.

Event ConnectComplete (pError As Error, adStatus As EventStatusEnum, pConnection As Connection)

The ConnectComplete is called when the Connect method has finished running in asynchronous mode.

pError is an object reference to an Error object if the value of adStatus is set to adStatusErrorsOccured.

adStatus is a Long value that contains one of the status values listed in Table 12-8 in the BeginTransComplete event.

pConnection is an object reference to the Connection object associated with the Connect method.

Event Disconnect (adStatus As EventStatusEnum, pConnection As Connection)

The Disconnect event is called when the connection has been dropped from the data source.

adStatus is a Long value that always contains adStatusOK.

pConnection is an object reference to the Connection object associated with the CommitTrans method.

Event ExecuteComplete (RecordsAffected As Long, pError As Error, adStatus As EventStatusEnum, pCommand As Command, pRecordset As Recordset, pConnection as Connection)

The ExecuteComplete event is called when the Execute method has finished running in asynchronous mode.

RecordsAffected is a Long value containing the number of records affected by the command executed by the Execute method.

pError is an object reference to an Error object if the value of adStatus is set to adStatusErrorsOccured.

adStatus is a Long value containing one of the status values listed in Table 12-8 in the BeginTransComplete event.

pCommand is an object reference to a Command object, if a Command object was executed.

pRecordset is an object reference to a Recordset object containing the results of the command's execution.

pConnection is an object reference to the Connection object associated with the ExecuteComplete method.

Event InfoMessage (pError As Error, adStatus As EventStatusEnum, pConnection as Connection)

The `InfoMessage` event is called when a warning message is received by the current connection.

`pError` is an object reference to an `Error` object if the value of `adStatus` is set to `adStatusErrorsOccured`.

`adStatus` is a `Long` value containing one of the status values listed in Table 12-8 in the `BeginTransComplete` event.

`pConnection` is an object reference to the `Connection` object associated with the message.

Event RollbackTransComplete (pError As Error, adStatus As EventStatusEnum, pConnection as Connection)

The `RollbackTransComplete` event is called when the `RollbackTrans` method has finished running in asynchronous mode.

`pError` is an object reference to an `Error` object if the value of `adStatus` is set to `adStatusErrorsOccured`.

`adStatus` is a `Long` value containing one of the status values listed in Table 12-8 in the `BeginTransComplete` event.

`pConnection` is an object reference to the `Connection` object associated with `RollbackTrans` method.

Event WillConnect (ConnectionString As String, UserID As String, Password As String, Options As Long, adStatus As EventStatusEnum, pConnection as Connection)

The `WillConnect` event is called before the process to establish that a connection is started. You can override any of the values in the `ConnectionString`, `UserID`, `Password`, and `Options` properties. By default, the value of `adStatus` is set to `adStatusOK`. If you set `adStatus` to `adStatusCancel`, you will terminate the connection request. This will trigger the `ConnectComplete` event with an `adStatus` of `adStatusErrorsOccurred`.

`ConnectionString` is a `String` containing the same connection information found in the `ConnectionString` property.

`UserID` contains a `String` value with the UserID needed to access the database.

Password contains a String value with the password associated with the specified UserID.

Options optionally passes one of the values specified in Table 12-7 in the Open method above.

adStatus is a Long value containing one of the status values listed in Table 12-8 in the BeginTransComplete event.

pConnection is an object reference to the Connection object associated with the connection that triggered this event.

Event WillExecute (Source As String, CursorType As CursorTypeEnum, LockType As LockTypeEnum, Options As Long, adStatus As EventStatus Enum, pCommand As Command, pRecordset As Recordset, pConnection as Connection)

The WillExecute event is called before a command is executed. You can override any of the values in the Source, CursorType, LockType and Options properties. By default, the value of adStatus is set to adStatusOK. If you set adStatus to adStatusCancel, you will terminate the connection request. This will trigger the ConnectComplete event with an adStatus of adStatusErrorsOccurred.

Source is a String containing the SQL Statement, stored procedure name, or other command to be executed.

CursorType contains a CursorTypeEnum value describing the type of cursor to be used in the Recordset (see Table 12-9).

LockType contains a LockTypeEnum value (see Table 12-10).

Options optionally passes one of the values specified in Table 12-7 in the Open method above.

adStatus is a Long value containing one of the status values listed in Table 12-7 in the BeginTransComplete event.

pCommand is an object reference to a Command object, if a Command object is about to be executed.

pRecordset is an object reference to a Recordset object, if a Recordset object was the source of the function to be executed.

pConnection is an object reference to the Connection object associated with the connection that triggered this event.

Table 12-9
Values for CursorType

Constant	*Value*	*Description*
`adOpenUnspecified`	-1	The type of cursor isn't specified.
`adOpenForwardOnly`	0	A forward-only cursor is used, which permits you only to scroll forward through the records in the `Recordset`.
`adOpenKeyset`	1	A keyset cursor is used, which is similar to a dynamic cursor, but doesn't permit you to see records added by other users.
`adOpenDynamic`	2	A dynamic cursor is used, which allows you to see records added by other users, plus any changes and deletions made by other users.
`adOpenStatic`	3	A static cursor is used, which prevents you from seeing any and all changes from other users.

Table 12-10
Values for LockType

Constant	*Value*	*Description*
`adLockUnspecified`	-1	The type of locking isn't specified.
`adLockReadOnly`	1	Doesn't permit you to change any values.
`adLockPessimistic`	2	Records are locked at the data source record by record once the data in the record has been changed.
`adLockOptimistic`	3	Records are locked only when you call the `UpdateMethod`.
`adLockBatchOptimistic`	4	Records are not locked, and conflicts will be returned for resolution after the `UpdateBatch` method has completed.

The Error Object

The `Error` object contains information about a specific error condition.

Error object properties

Table 12-11 lists the properties associated with the `Error` object.

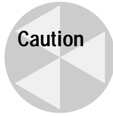
Table 12-11
Properties of the Error Object

Property	*Description*
`Description`	A `String` value containing a short text description of the error.
`HelpContext`	A `Long` value containing the help context ID reference within the help file specified by `HelpFile`. If no additional help can be found, this value will contain a zero.
`HelpFile`	A `String` containing the name of the help file where a more detailed description of the error may be found. If no additional help is available, this value will contain an empty string.
`NativeError`	A `Long` containing a provider-specific error code.
`Number`	A `Long` containing the OLE DB error code number. This value is unique to this specific error condition.
`Source`	A `String` containing the name of the object or application that caused the error. ADO errors will generally have `Source` values of the format ADODB.*objectname*, ADOX.*objectname*, or ADOMD.*objectname*, where *objectname* is the name of the object that caused the error.
`SQLState`	A `String` containing the standard five-character ANSI SQL error code.

The Errors Collection

The `Errors` collection contains the set of errors generated in response to a specific failure. If an operation fails, the `Errors` collection is cleared and all of the individual errors are recorded in the collection.

If you are using the Resync, UpdateBatch, or CancelBatch methods on a Recordset object, you may generate a set of warnings that will not raise the On Error condition in Visual Basic. Thus, it is important to check for warnings when using these methods and take the appropriate action.



I'm certain it didn't error again: Successfully performing a function will not clear the Errors collection. Thus, the information from a previous error will remain in the collection until it is either explicitly cleared or another error occurs. For this reason, it is very important that you clear the Errors collection after you handle the error condition and before you resume normal processing. Otherwise, you may falsely detect an error condition.

Errors collection properties

Table 12-12 lists the properties associated with the Errors collection.

Table 12-12
Properties of the Errors Collection

Property	*Description*
Count	A Long value containing the number of errors in the collection.
Item(index)	An object reference to an Error object containing information about a particular error. To locate an error, specify a value in the range of 0 to Count –1.

Errors collection methods

The Errors collection contains methods to manage the collection of error information.

Sub Clear ()

The Clear method initializes the Errors collection to the empty state.

Sub Refresh ()

The Refresh method gets a fresh copy of the error information from the data provider.

Connecting To Database Server

In the previous chapters, I provided all of the information necessary to connect to either the ADO Data Control or the Data Environment Designer and they took care of connecting to the database server. However, if you plan to use the ADO objects directly, you need to deal with a few issues yourself.

Connection strings

A *connection string* contains the information necessary to connect your application to a data source. This value is stored in the `ConnectionString` property of the `Connection` object. It consists of a series of keyword clauses separated by semicolons (;). You create a keyword clause by specifying a keyword, an equal sign (=), and then the value of the keyword. If the same keyword is specified more than once, only the last occurrence will be used, except in the case of the `provider` keyword, in which the first occurrence will be used.

Note

Spaces are permitted: A keyword always ends with an equal sign, so special characters, such as a space or a period, are legal.

Consider the following connection string:

```
provider=sqloledb;data source=Athena;initial catalog=VB6DB
```

It uses the `sqloledb` provider and then specifies `Athena` as the data source and `VB6DB` as the initial catalog. Note the spaces inside both the data source and the initial catalog keywords are legal.

Tip

Connection strings the easy way: Building a connection string to a new database system can be a real headache, making sure that you have all the needed keywords to make the connection. Try building a dummy application using the ADO Data Control. Then configure the `ConnectionString` property using the Properties dialog box. This creates the connection string and puts it in the `ConnectionString` property. Then all you need to do is copy the connection string to your application.

Provider keyword

The `Provider` keyword specifies the name of the OLE DB provider that will be used to connect to the data source. If this keyword is not included in the connection string, the `OLD DB Provider for ODBC` will be used. Table 12-13 lists some common databases and their OLE DB providers.

Table 12-13
Common OLE DB Providers

Database	*Provider*
OLE DB Provider for ODBC	`MSDASQL.1`
Jet 3.51 (Access 97)	`Microsoft.Jet.OLEDB.3.51`
Jet 4.0 (Access 2000)	`Microsoft.Jet.OLEDB.4.0`
Oracle	`MSDAORA.1`
SQL Server 7	`SQLOLEDB`

Common keywords

Nearly all data providers support the keywords listed in Table 12-14. In many cases, these keywords will be all you need to connect to the data source.

Table 12-14
Common Keywords for SQLOLEDB

Keyword	*Alias*	*Description*
`Data Source`	`Server`	Specifies the location of the database server or the name of the file containing the data, depending on the specific provider.
`Initial Catalog`	`Database`	Specifies the name of the default database on the database server.
`Password`	`PWD`	Specifies the password associated with the User Id keyword.
`User Id`	`UID`	Specifies the user name that will be used to connect to the database server.

Keywords for SQLOLEDB

Accessing an SQL Server database is very straightforward. All you need to do is include the `Data Source` keyword and either specify the user name and password or set the `Trusted_Connection` keyword to `yes`. However, there are a number of other keywords that can provide additional functions. Table 12-15 lists the set of keywords that are specific to SQL Server.

Table 12-15
Common Keywords for SQLOLEDB

Keyword	*Description*
`Application Name`	Contains the name of the application program.
`Connect Timeout`	Specifies the number of seconds in which the database server must respond before the connection will timeout.
`Integrated Security`	When set to `SSPI`, Windows NT Authentication will be used.
`Trusted_Connection`	Contains `yes` if you are using Windows NT Authentication.
`Workstation ID`	Contains name of the client machine.

Keywords for Microsoft.Jet.OLEDB.4.0

When accessing a Jet database, you need to remember that there isn't really a database server involved, like there is with most other database systems. The database is a specially formatted disk file, which you reference in the `Data Source` keyword. The other keywords have the normal meaning, and the list of specific keywords for the Microsoft Jet provider is listed in Table 12-16.

Table 12-16
Common Keywords for SQLOLEDB

Keyword	*Description*
`Jet OLEDB:System Database`	Contains the fully qualified file name for the workgroup information file.
`Jet OLEDB:Registry Path`	Specifies the registry key that contains values for the database engine.
`Jet OLEDB:Database Password`	Contains the database password.

Opening a connection

Opening a database connection is merely a matter of declaring an object, creating a new instance of it, and then opening the connection. The key is using the proper connection string when you open the connection.

Declaring a Connection object

The following line of code declares the variable db as a `Connection` object.

```
Dim db As ADODB.Connection
```

You can use any of the methods or properties associated with the `Connection` object, but not any of the events. This is perfectly fine for most programs. The events only provide status information that can be safely ignored by most programs.

Globally speaking: When creating applications with multiple forms, I often add a module to the program to hold objects that I want to access, including things like the `Connection` object, which can easily share among multiple forms.

Sometimes, however, you might want to track this status information. This is a great place to include extra security checks, since you have the opportunity to review various functions before they are actually performed, and cancel them. To include events with the `Connection` object, you need to use the `WithEvents` keyword in the `Dim` statement as in the statement below:

```
Dim WithEvents db As ADODB.Connection
```

Qualifying for clarity: I usually use the `ADODB` prefix for all ADO objects. This eliminates confusion with other objects (such as the DAO) that have the same name.

The `WithEvents` keyword imposes some restrictions on how you can declare your object. You can only use it in `Form` modules and `Class` modules. You can't use it in a regular `.BAS` module. You also can't use the `New` keyword. You must instantiate the object using a `Set` statement with the `New` keyword.

Faster objects: While you can use the `New` keyword in a `Dim` statement to create an object the first time it's used, it adds code to every statement to see if the object exists and create it if necessary.

Coding the Connection object

One of the things I like best about using the ADO objects directly, rather than using the Data Environment Designer or the ADO Data Command, is that the database isn't opened for me when the program is started. This allows me the opportunity to ask the user for their user name and password before I open the database.

The Connect Demo program shown in Figure 12-1 is a very simple program that demonstrates how to connect to an SQL Server database. It consists of two command buttons that are used to connect and disconnect from the database, plus two text boxes that allow the user to enter their user name and password.

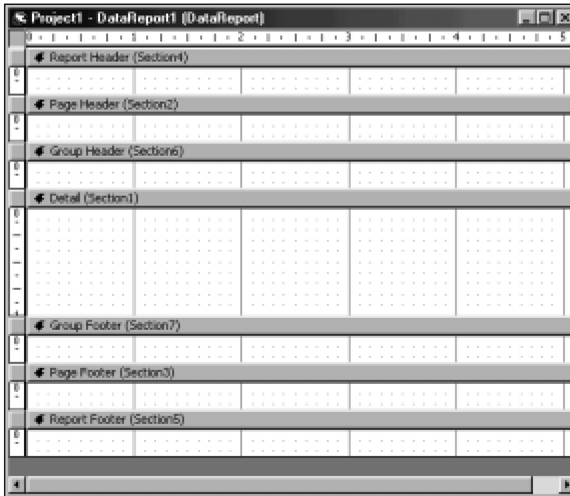


Figure 12-1: The Connect Demo program

Clicking the button labeled Connect will fire the `Command1_Click` event, as shown in Listing 12-1. I begin the routine by using the `On Error Resume Next` statement, which prevents a run-time error from killing the program. However, I need to be careful to explicitly check for error conditions, or an undetected error could cause havoc with my program.

Listing 12-1: The Command1_Click event in Connect Demo

```
Private Sub Command1_Click()

Dim p As ADODB.Property

On Error Resume Next
Set db = New ADODB.Connection
db.ConnectionString = "provider=sqloledb;" & _
   "data source=Athena;initial catalog=VB6DB"

db.Properties("User Id").Value = Text1.Text
db.Properties("Password").Value = Text2.Text

db.Open
```

Continued

Listing 12-1 *(continued)*

```
If db.State = adStateOpen Then
   Command1.Enabled = False
   Command2.Enabled = True

Else
   WriteError

End If

End Sub
```

Next, I create a new instance of the `Connection` object using the `Set` statement. Then I set the various properties in the `Connection` before I open the connection. While I can set the `ConnectionString` property directly, I need to set the values for `User Id` and `Password` through the `Properties` collection. For these values, I simply use the name of the property as the index in the `Properties` collection and assign the values I want to the `Value` property. Of course, these properties are specific to the provider that is used, so you need to see which of these custom properties you really need.

Check "The Parameter Object" and "The Parameters Collection" in Chapter 13 for more details about these objects.

Opening the connection

Once the properties are set, I invoke the `Open` method to connect to the database server. Another way to handle the connection would be to specify all of the information as part of the call to the `Open` method, as shown below:

```
db.Open "provider=sqloledb;data source=Athena; " & _
   "initial catalog=VB6DB", Text1.Text, Text2.Text
```

This has the advantage of fewer lines of text, which means fewer places where something can go wrong.

After I've used the `Open` method, I need to know if it was successful. Had I not used the `On Error` statement, I could safely assume that the `Open` method was successful, because the program would had gotten a run-time error and died. Here I can do one of two things. First, I can check the `Errors` object to see if there was an error and check the error code for the appropriate action. Second, I can check the `State` property to make sure that the object's state is open. If the connection is open, I'll disable the button that connects to the database and enable the button that closes the connection. If it isn't, I'll display an error message to the user with the `WriteError` routine.

Closing a connection

Closing a `Connection` object is merely a matter of using the `Close` method and releasing the resources associated with the object, which you can see in Listing 12-2. If I was able to close the connection, I will disable the Disconnect button and reenable the Connect button so the user can try connecting to the database again. If the `Close` method generated an error, I'll display the error using the `WriteError` routine.

Listing 12-2: The Command2_Click event in Connect Demo

```
Private Sub Command2_Click()

db.Close
If db.State = adStateClosed Then
   Set db = Nothing
   Command2.Enabled = False
   Command1.Enabled = True

Else
   WriteError

End If

End Sub
```

Analyzing Errors

The `Connection` object's `Errors` collection contains the information about the most recent error that occurred. When performing database functions, it is quite possible that a single request may generate multiple errors. Usually, the first error is the most significant error, and the rest of the errors are secondary effects of the main error.

Retrieving error informationThe `WriteError` subroutine in Listing 12-3 is designed to update a `StatusBar` control with the results of the most recent error. I check the `Count` property to see how many errors are in the collection, and if there's only one, I display it in the status bar. If I have multiple errors, I'll display the first error in the status bar just like I displayed the single error, and then use a `For Each` loop to display each of the individual error messages.

Listing 12-3: The WriteError subroutine in Connect Demo

```
Private Sub WriteError()

Dim e As ADODB.Error

If db.Errors.Count = 1 Then
   StatusBar1.SimpleText = "Error: " & db.Errors(0).Description

Elseif db.Errors.Count > 1 Then
   StatusBar1.SimpleText = "Multiple errors:" & _
      db.Errors(0).Description

   For Each e In db.Errors
      MsgBox e.Description

   Next e

End If

db.Errors.Clear

End Sub
```

It is important to clear the `Errors` collection before you issue the next database request. The `Errors` collection is only cleared automatically the next time an error is encountered. If you don't clear the collection before you issue a database request, and then check it afterwards, you can't be certain that the errors in the `Errors` collection were caused by the most recent database request.

There are two places where you should clear the `Errors` collection. The first is immediately after handling an error condition, as I did in the `WriteError` routine. This ensures that `Error` object is clear after processing an error condition. However, since it is possible that you may not check every place there can be an error, you should also clear the `Errors` collection before any call that might result in an error.

Watching connection activity

The events associated with the `Connect` object provide a way to catch an activity before and after it executes. This will allow you to grab information and display it to the user, or to review the request and deny it.

Displaying status information

The `db_ConnectComplete` event shown in Listing 12-4 will be fired after the user connects to the database. I begin by checking the `adStatus` parameter to see if the

connection completed without an error. If it did, I let the user know that they're connected to the database. Otherwise, I get the error message from the `pError` object and display it in the status bar.

Listing 12-4: The db_ConnectComplete event in Connect Demo

```
Private Sub db_ConnectComplete(ByVal pError As ADODB.Error, _
   adStatus As ADODB.EventStatusEnum, _
   ByVal pConnection As ADODB.Connection)

If adStatus = adStatusOK Then
   StatusBar1.SimpleText = "Connected."

Else
   StatusBar1.SimpleText = "Error: " & pError.Description

End If

End Sub
```

Canceling a request

The `Complete` events in the Connection object merely indicate the current status of a request. The `Will` events, on the other hand, are the perfect place to review a request and cancel it if desired. Listing 12-5 takes advantage of the `db_WillConnect` event to see if the user really wants to connect to the database server.

Listing 12-5: The db_WillConnect event in Connect Demo

```
Private Sub db_WillConnect(ConnectionString As String, _
   UserID As String, Password As String, Options As Long, _
   adStatus As ADODB.EventStatusEnum, _
   ByVal pConnection As ADODB.Connection)

If MsgBox("Do you really want to connect?", vbYesNo, _
      "Connect to remote database") = vbYes Then
   StatusBar1.SimpleText = "Will connect."

Else
   adStatus = adStatusCancel

End If

End Sub
```

I begin this routine by displaying a message box that asks the user if they really want to connect to the database server. If the user responds no, I'll cancel the request by setting the `adStatus` parameter to `adStatusCancel`. This will trigger an error condition, which is intercepted by both the `WriteError` routine and the `ConnectComplete` event. If the user responds yes, the status bar is updated, and the `Open` method will be allowed to continue.

So it's contrived: The example here is somewhat contrived; however, in a real application, you may want to restrict connections based on the time of date, day of week, or particular value of the user name.

Thoughts on the Connection Object

The `Connection` object manages the path to your database server. In most cases, you'll open the connection when your program begins and close it when it ends. You won't bother with any of the events and you may not even specify any of the connection properties other than the connection string. After all, the main reason you want to use the `Connection` object is to connect to the database. The real work of your application will be done with the other objects in the ADO library.

Summary

In this chapter you learned the following:

- **You can use the `Connection` object to establish a link between the database client program and the database server.**
- **You can use connection strings to specify the parameters that are passed to the OLE DB provider to establish the connection to the database server.**
- **You can use the `Error` object to determine why the most recent database request failed.**
- **You can use the events in the `Connection` object to gather information about the various database requests sent to the database server and cancel them if desired.**