13

CHAPTER

In This Chapter

Presenting the ADO `Command` Object

Introducing the ADO `Parameter` Object

Working with the ADO `Parameters` Collection

Creating and executing commands

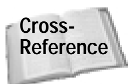
Using stored procedures

Using Commands and Stored Procedures

In this chapter, I will cover how to create and use ADO `Command` objects. The `Command` object allows you to specify a command that will retrieve data from your database. Typically this object is used with stored procedures and SQL queries that include a series of parameters.

Introducing the ADO Command Object

After you have an open connection, the first thing you're going to want to do is to execute some commands. Commands are defined in the `Command` object and can be SQL statements or calls to stored procedures. Some commands, particularly those that use stored procedures, may have parameters that supply additional information to the command or return information after the command was executed. Parameter information is stored in the `Parameters` collection, with each parameter having its own `Parameter` object. Also, depending on the particular command, it may or may not return a `Recordset` object containing rows of information from your database.



Discussing `Command` objects without talking about `Recordset` objects can be difficult, since the primary reason for using `Command` objects is to create `Recordsets`. See Chapter 14 for more information about `Recordset` objects.

The Command Object

The Command object contains information about an SQL statement or stored procedure you wish to execute against a database. You may optionally include a list of parameters that will be passed to the stored procedure. If the command returns a set of rows, the Command object will return a Recordset object containing the results.

Command object properties

Table 13-1 lists the properties associated with the Command object.

Table 13-1
Properties of the Command Object

Property	*Description*
ActiveConnection	A Variant array containing an object reference to the Connection object used to access the database. May also contain a valid connection string that will dynamically create a connection to your database when you use the Execute method.
CommandText	A String value containing an SQL statement, a stored procedure name, or other provider command to be executed.
CommandTimeout	A Long value containing the maximum number of seconds for the command to execute before an error is returned. Default is 30 seconds.
CommandType	An enumerated type (see Table 13-2) describing the type of command to be executed.
Name	A String containing the name of the object.
Parameters	An object reference to a Parameters collection containing the parameters that will be passed to a stored procedure or a parameterized query.
Prepared	A Boolean, when TRUE, means that the command should be prepared before execution.
Properties	An object reference to a Properties collection containing additional information about the Command object.
State	A Long describing the current state of the Command object. Multiple values from Table 13-3 can be combined to describe the current state.

Table 13-2
Values for CommandType

Constant	*Value*	*Description*
`adCmdUnspecified`	-1	`CommandType` should be automatically determined.
`adCmdText`	1	`CommandText` contains either an SQL statement or stored procedure call.
`adCmdTable`	2	`CommandText` contains the name of a table in the database.
`adCmdStoredProcedure`	4	`CommandText` contains the name of a stored procedure.
`adCmdUnknown`	8	The type of command isn't known.
`adExecuteNoRecords`	128	Indicates that the command will not return any rows or will automatically discard any rows that are generated. Must be used with `adCmdText` or `adCmdStoredProcedure`.
`adCmdFile`	256	`CommandText` is the name of a persistently stored `Recordset`.
`adCmdTableDirect`	512	`CommandText` contains the name of a database table.

Table 13-3
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.

Command bject methods

The `Command` object contains only three methods: `Cancel`, `CreateParameter`, and `Execute`.

Sub Cancel()

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

Function CreateParameter ([Name As String], [Type As DataTypeEnum = adEmpty], [Direction As ParameterDirectionEnum = adParamInput], [Size As Long], [Value]) As Parameter

The `CreateParameter` method creates a new `Parameter` object, which may be added to the `Parameters` collection using the `Parameters.Append` method.

`Name` specifies the name of the parameter.

`Type` specifies the data type of the parameter.

`Direction` specifies whether the parameter is input only, output only, or both input and output.

`Size` specifies the length of the parameter. This parameter is only important when you have a variable length data type, such as a string or an array.

`Value` specifies the value of the parameter.

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

The `Execute` method runs the SQL statement or stored procedure specified in the `Command` object. A `Recordset` object will be returned as the result of the function, which will contain any rows returned by the command.

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

`Parameters` optionally passes a `Variant` array containing a list of parameters to be used by the command. Note that output parameters will not return correct values through this parameter. Instead, you should use the `Parameters` collection to get the correct values for output parameters.

`Options` **optionally passes a combination of the values specified in Table 13-4. Note that some of these values are the same as those available for the** `CommandType` **property.**

Table 13-4
Values for Options

Constant	*Value*	*Description*
`adOptionUnspecified`	-1	No options are specified.
`adCmdText`	1	`CommandText` contains either an SQL statement or stored procedure call.
`adCmdTable`	2	`CommandText` contains the name of a table in the database.
`adCmdStoredProcedure`	4	`CommandText` contains the name of a stored procedure.
`adCmdUnknown`	8	The type of command isn't known.
`adAsyncExecute`	16	The command should be executed asynchronously.
`adAsyncFetch`	32	After the number of rows specified in the `CacheSize` property of the `Recordset` object are returned, the remaining rows will be returned asynchronously.
`adAsyncFetchNonBlocking`	64	The main thread isn't blocked while retrieving rows. If the current row hasn't been retrieved, the current row will be moved to the end of the file.
`adExecuteNoRecords`	128	Indicates that the command will not return any rows or will automatically discard any rows that are generated. Must be used with either the `adCmdText` or `adCmdStoredProcedure` values.
`adCmdFile`	256	`CommandText` is the name of a persistently stored `Recordset`.
`adCmdTableDirect`	512	`CommandText` contains the name of a database table.

The Parameter Object

The `Parameter` object contains information about a parameter used in a stored procedure or parameterized query defined in a `Command` object.

Parameter object properties

Table 13-6 lists the properties associated with the `Parameter` object.

Table 13-6
Properties of the Parameter Object

Property	*Description*
`Attributes`	An enumerated type describing the characteristics of the parameter (see Table 13-7).
`Direction`	An enumerated type that describes whether the parameter is an input-only, input/outpu, or output-only parameter (see Table 13-8).
`Name`	A `String` value containing the name of the parameter.
`NumericScale`	A `Byte` value containing the number of digits to the right of the decimal point for a numeric field.
`Precision`	A `Byte` value containing the total number of digits in a numeric field.
`Properties`	An object reference to a `Properties` collection containing provider-specific information about a parameter.
`Type`	An enumerated type (see Table 13-9) containing the OLE DB data type of the field.
`Value`	A `Variant` array containing the current value of the parameter.

Table 13-7
Values for Attributes

Constant	*Value*	*Description*
`adFldUnspecified`	-1	The provider doesn't supply field attributes.
`adFldMayDefer`	2	The field value is not retrieved with the whole record, but only when you explicitly access the field.

Constant	***Value***	***Description***
`adFldUpdateable`	4	The field's value may be changed.
`adFldUnknownUpdateable`	8	The provider can't determine if you can change the field's value.
`adFldFixed`	16	The field contains fixed-length data.
`adFldIsNullable`	32	The field will accept **Null** values.
`adFldMayBeNull`	64	The field may contain a **Null** value.
`adFldLong`	128	The field contains a long binary value and you should use the `AppendChunk` and `GetChunk` methods to access its data.
`adFldRowID`	256	The field contains an identity value that can't be changed.
`adFldRowVersion`	512	The field contains a time stamp value that is used to track updates.
`adFldCacheDeferred`	4096	This field is cached by the provider and subsequent reads and writes are done from cache.
`adFldIsChapter`	8192	The field contains a chapter value, which specifies a specific child `Recordset` related to this parent field.
`adFldNegativeScale`	16384	The field contains a numeric column that supports negative scale values.
`adFldKeyColumn`	32768	The field is (or is at least part of) the primary key for the table.
`adFldIsRowURL`	65536	The field contains the URL that names the resource from the data store represented by the record.
`adFldIsDefaultStream`	131072	The field contains the default stream for the resource represented by the record.
`adFldIsCollection`	262144	The field contains a collection of another resource, such as a folder, rather than a simple resource, such as a file.

Table 13-8
Values for Direction

Constant	*Value*	*Description*
adParamUnknown	0	The direction of the parameter is unknown.
adParamInput	1	The parameter is an input-only parameter.
adParamOutput	2	The parameter is an output-only parameter.
adParamInputOutput	3	The parameter is both an input and an output parameter.
adParamReturnValue	4	The parameter contains a return value.

Table 13-9
Values for Type

Constant	*Value*	*Description*
adEmpty	0	This field has no value (DBTYPE_EMPTY).
adSmallInt	2	This field has an Integer value (DBTYPE_I2).
adInteger	3	This field has a Long value (DBTYPE_I4).
adSingle	4	This field has a Single value (DBTYPE_R4).
adDouble	5	This field has a Double value (DBTYPE_R8).
adCurrency	6	This field has a Currency value (DBTYPE_CY).
adDate	7	This field has a Date value (DBTYPE_DATE).
adBSTR	8	This field has a null-terminated Unicode string (DBTYPE_BSTR).
adIDispatch	9	This field has a pointer to an IDispatch interface in a COM object (DBTYPE_IDISPATCH).
adError	10	This field has a 32-bit error code (DBTYPE_ERROR).
adBoolean	11	This field has a Boolean value (DBTYPE_BOOL).

Constant	*Value*	*Description*
adVariant	12	This field has a Variant value (DBTYPE_VARIANT). Note that this type is not supported by ADO and causes unpredictable results.
adIUknown	13	This field has a pointer to an IUnknown interface in a COM object (DBTYPE_IUNKNOWN).
adDecimal	14	This field has an exact numeric value with a fixed precision and scale (DBTYPE_DECIMAL).
adTinyInt	16	This field has a one byte signed integer (DBTYPE_I1).
adUnsignedTinyInt	17	This field has a one byte unsigned integer (DBTYPE_UI1).
adUnsignedInt	18	This field has a two byte unsigned integer (DBTYPE_UI2).
adUnsignedInt	19	This field has a four byte unsigned integer (DBTYPE_UI4).
adBigInt	20	This field has an 8-byte signed integer (DBTYPE_I8).
adUnsignedBigInt	21	This field has an 8-byte unsigned integer (DBTYPE_UI8).
adFileTime	64	This field has a 64-bit date-time value represented as the number of 100-nanosecond intervals since 1 January 1601 (DBTYPE_FILETIME).
adGUID	72	This field has a globally unique identifier value (DBTYPE_GUID).
adBinary	128	This field has a Binary value (DBTYPE_BYTES).
adChar	129	This field has a String value (DBTYPE_STR).
adWChar	130	This field contains a null-terminated Unicode character string (DBTYPE_WSTR).
adNumeric	131	This field contains an exact numeric value with a fixed precision and scale (DBTYPE_NUMERIC).

Continued

Table 13-9 *(continued)*

Constant	***Value***	***Description***
adUserDefined	132	This field contains a user-defined value (DBTYPE_UDT).
adDBDate	133	This field has a date value using the YYYYMMDD format (DBTYPE_DBDATE).
adDBTime	134	This field has a time value using the HHMMSS format (DBTYPE_DBTIME).
adDBTimeStamp	135	This field has a date-time stamp in the YYYYMMDDHHMMSS format (DBTYPE_DBTIMESTAMP).
adChapter	136	This field has a 4-byte chapter value that identifies the rows in a child rowset (DBTYPE_HCHAPTER).
adPropVariant	138	This field has an Automation PROPVARIANT (DBTYPE_PROP_VARIANT).
adVarNumeric	139	This field contains a numeric value. (Available for Parameter objects only.)
adVarChar	200	This field has a String. (Available for Parameter objects only.)
adLongVarChar	201	This field has a long character value. (Available for Parameter objects only.)
adVarWChar	202	This field has a null-terminated Unicode character string value. (Available for Parameter objects only.)
adLongVarWChar	203	This field has a long null-terminated character string value. (Available for Parameter objects only.)
adVarBinary	204	This field has a binary value. (Available for Parameter objects only.)
adLongVarBinary	205	This field has a long binary value. (Available for Parameter objects only.)

Parameter object methods

There is only one method available for the Parameter object: the AppendChunk method.

Sub AppendChunk (Data As Variant)

The AppendChunk method is used to add data to a large text or binary field. The first time the AppendChunk method is used, the value in Data will overwrite any existing data in the field. For subsequent calls, simply append data to the end of the existing data. Data is a Variant array containing the data to be appended to the end of the field.

The Parameters Collection

The Parameters collection contains the set of parameters associated with a Command object.

Parameters collection properties

Table 13-10 lists the properties associated with the Parameters collection.

Table 13-10
Properties of the Parameters Collection

Property	*Description*
Count	A Long value containing the number of Parameter objects in the collection.
Item(index)	An object reference to a Parameter object containing information about a particular field in the Recordset. To locate a field, specify a value in the range of 0 to Count –1 or the name of the Parameter.

Parameters collection methods

The method available for the Parameters collection allows you to manage the set of Parameter objects.

Sub Append (Object As Object)

The `Append` method adds a `Parameter` object to the collection. The `Type` property of the `Parameter` object must be defined before you can add it to the collection. `Object` is a `Parameter` object containing the new parameter.

Use the `CreateParameter` method of the `Command` object to create a new `Parameter` object, then use the `Append` method to add it to the collection.

Sub Delete (Index As Variant)

The `Delete` method removes an object specified by `Index` from the collection. `Index` is either a `String` value containing the name of the parameter or a `Long` value containing the ordinal position of the `Parameter` object to be deleted.

Sub Refresh()

The `Refresh` method can be used to retrieve information about the parameters in a stored procedure from the data provider. To use `Refresh` in this fashion, you need a valid `Command` object with an active connection to the data source. Then all you need to do is specify values for `CommandText` and `CommandType` and use the `Parameters.Refresh` method. This will retrieve all of the information for the stored procedure from the data provider. Note that this works only if your data provider supports stored procedures or an SQL query with embedded parameters (also known as a *parameterized query*).

Running SQL Statements

A `Command` object contains all of the information necessary to send a command to the database server for execution. Sometimes the command will need parameters, while at other times it will not. Depending on the command you want to execute, it may return nothing or a `Recordset` object containing the set of rows selected from the database. No matter what the command is, the approach you use to define it and run it is basically the same.

Running a simple command

Every `Command` you execute needs at least two pieces of information before you can use it. You must set the `CommandText` to the command you want to execute and you must set the `ActiveConnection` property to an open `Connection` object.

While not as important, you should set the `CommandType` property to the value appropriate for the command you want to execute. If this value is not specified, or set to `adCmdUnknown`, the database server must determine if the command is an

SQL statement, a stored procedure, or a table name before it can execute it. While this extra overhead isn't large by itself, it can be significant when you multiply it by the number of requests the server must execute.

The Command Demo program (see Figure 13-1) is based on the Connection Demo program from Chapter 12. All of the code and form elements are carried over intact, although I added a few new command buttons and `Click` events to demonstrate how to use the `Command` object.

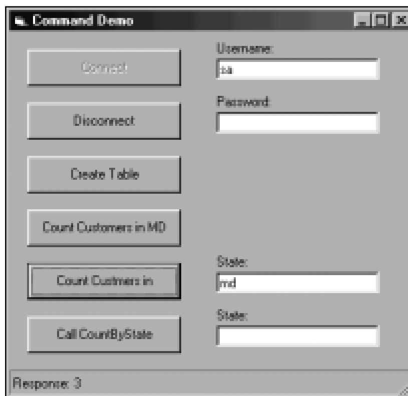


Figure 13-1: Running the Command Demo program

The Command Demo program can be found on the CD-ROM in the \VB6DB\Chapter13\CommandDemo directory.
.

Listing 13-1 shows how you can use a `Command` object to create a database table. In this example, I create a table called `MyTable`. However, this technique can be used to execute any SQL statement except for the **Select** statement. (The **Select** statement will return a `Recordset` object, which requires some additional code.)

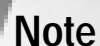
Listing 13-1: The Command5_Click event in Command Demo

```
Private Sub Command5_Click()

Dim c As ADODB.Command

On Error Resume Next
```

Continued

Listing 13-1 *(continued)*

```
Set c = New ADODB.Command
Set c.ActiveConnection = db
c.CommandText = "Create Table MyTable (MyColumn Char (10))"
c.CommandType = adCmdText

db.Errors.Clear
c.Execute
If db.Errors.Count > 0 Then
   WriteError

End If

End Sub
```



Once is okay, twice isn't: Since this routine creates a table in your database, running it a second time will fail unless you remove the table manually. You should use a query tool to delete the table before you try it a second time or try it again to see the error that will result from executing this command twice.

The routine begins by creating a new instance of the `Command` object and setting the `ActiveConnection` property to the same `Connection` object I created in Chapter 12. Then I assign the SQL statement I want to execute to the `CommandText` property. Since I'm executing an SQL statement, I'll set the `CommandType` to `adCmdText` to prevent the server from trying to determine if it is the name of a table or stored procedure.

Once the command is defined, executing is easy. I `Clear` the `Errors` collection to make sure that any errors in the collection are caused by the `Execute` method. Then I use the `Execute` method to run the command on the database server. If there are any errors in the `Errors` collection, I'll call the `WriteError` routine to display the information.

Returning a Recordset

Working with `Recordsets` is basically the same as working with simple commands except that the `Execute` method returns a reference to `Recordset` object. In Listing 13-2, I create a `Connection` object the same way as in Listing 13-1, but instead of the **Create Table** SQL statement, I use a **Select** statement.

Listing 13-2: The Command4_Click event in Command Demo

```
Private Sub Command4_Click()

Dim c As ADODB.Command
Dim rs As ADODB.Recordset

On Error Resume Next

Set c = New ADODB.Command
Set c.ActiveConnection = db
c.CommandText = "Select Count(*) From Customers Where State =
'MD'"
c.CommandType = adCmdText

db.Errors.Clear
Set rs = c.Execute
If db.Errors.Count > 0 Then
   WriteError

Else
   StatusBar1.SimpleText = "Response: " & _
      FormatNumber(rs.Fields(0).Value, 0)

End If

End Sub
```

Since the call to `c.Execute` will return an object reference, I need to use a `Set` statement to assign the object reference to the `Recordset` object. Note that I didn't need to create an instance of the `Recordset` object. The `Execute` method took care of this for me.

Next, I check for errors in the `Errors` collection and call the `WriteError` routine if I find any. Otherwise, I output the value of the first field in the first row of the `Recordset` in the status bar. You can verify that the value is correct by executing the same query using your database query tool.

Tip

Quick and dirty: Building and debugging a database application can be difficult. If you're truly paranoid, like me, you don't trust your application until you can verify that it worked properly using an independent tool. This is where a tool like SQL Server's Query Analyzer comes in handy. You can use it to execute any type of query you like. Thus, you can test your **Select** statement to ensure that the right number of rows was returned or to verify that an update to the database was made properly. You can also use it to test various SQL statements, and when you're satisfied that they're correct, you can copy them to the `CommandText` property of a `Command` object.

Running with parameters

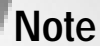
While knowing the number of customers in Maryland is nice, a better approach than creating a `Command` object for each state would be to create one that accepts parameters. This is also an easy process, as you can see in Listing 13-3. This routine is based on the one shown in Listing 13-2. The only differences are the ones needed for parameters.

Listing 13-3: The Command3_Click event in Command Demo

```
Private Sub Command3_Click()

Dim c As ADODB.Command
Dim p As ADODB.Parameter
Dim rs As ADODB.Recordset

On Error Resume Next

Set c = New ADODB.Command
Set c.ActiveConnection = db
c.CommandText = "Select Count(*) From Customers Where State =
?"
c.CommandType = adCmdText

Set p = c.CreateParameter("State", adChar, adParamInput, 2)
c.Parameters.Append p

c.Parameters("State").Value = Text3.Text

db.Errors.Clear
Set rs = c.Execute
If db.Errors.Count > 0 Then
   WriteError

Else
   StatusBar1.SimpleText = "Response: " & _
      FormatNumber(rs.Fields(0).Value, 0)

End If

End Sub
```

The first difference you might have noticed is that I assigned the **Select** statement to the `CommandText` property. In place of the MD, there is now a question mark (?). Question marks are used to identify the place where a parameter will be substituted into the statement. In this case, the parameter is the two-character state abbreviation.

Next, I use the CreateParameter method to create a new Parameter object called State. I'll save the object reference in a temporary variable called p. It has a type of **Char(2)** and is an input-only parameter. The last argument of the method is omitted. Had I wanted to assign a default value for the parameter, I would have specified it as the last parameter. After creating the parameter, I use the Append method of the Parameters collection to add the object to the collection.



Order in the parameters: The order in which you add the parameters to the collection is the same order that will be used to match the parameters to the question marks in the CommandText. The first parameter is mapped to the first question mark, while the second parameter is mapped to the second question mark, and so on.

Once the parameter has been defined, you can use the Parameters collection to identify the parameter by name and assign it a value. Then you can use the Execute method to generate the Recordset object and display the results.

An alternate way to execute a command with parameters is to supply the parameter values as part of the Execute method, as shown in the following line of code:

```
Set rs = c.Execute(, Array(Text3.Text))
```

The second argument of the Execute method allows you to specify a Variant array containing a list of parameters that will be used by the command. The easiest way to construct a Variant array is to use the Array function, which takes a list of values and returns a Variant array containing the values. If you have already specified a parameter directly in the Parameters collection, the value specified in the Execute method will override that value.

Stored Procedures

Stored procedures are useful tools that allow you to execute a set of SQL statements on the database server by issuing a single command with a series of parameters. Information can be returned via the Parameters collection or in a Recordset object. Stored procedures are highly dependent on the database system on which they run. However, the benefits of using stored procedures often far outweigh having SQL statements that are independent of a particular database system.

Advantages of stored procedures

Many people use stored procedures in their database applications for three main reasons: faster performance, application logic, and security.

Faster performance

Using stored procedures is typically faster than issuing the equivalent SQL statements from your application, for several reasons. The first reason is that stored procedures are stored on the database server in a prepared format. This avoids the overhead of preparing a statement on the fly. Also, if you repeatedly execute the same statement, most database servers will prepare the statement each time you execute it, imposing a lot of extra overhead on the database server.

Prepared for speed: Before any SQL statement can be executed, it must be prepared. Preparing a statement involves parsing the words in the statement, compiling them into a package, and then optimizing the package based on the data in the database.

In addition, stored procedures often contain multiple SQL statements, which means that you don't have to wait for a response across the network before you send the next command. The individual statements are executed in sequence until the stored procedure is complete. This means that the intermediate recordsets you would have transmitted to the client computer, and the commands you would have issued in response, aren't necessary. Which in turn reduces the amount of time needed to perform the function.

Application logic

Stored procedures are written using a language that allows you to have local variables, perform computations, call other stored procedures, and process recordsets — as well as execute SQL statements. In short, you can think of a stored procedure much like you would think of a Visual Basic program, but one whose execution is tightly coupled with the database server.

Since many developers find this concept appealing, they code their business logic as stored procedures and make them available for application programmers to use. This ensures that each program can take advantage of the business logic, and as long as the calling sequence isn't changed, you won't have to recompile your Visual Basic program each time a stored procedure is changed.

Security

Using a stored procedures can be more secure than granting someone direct access to a database. Since stored procedures are database objects, they are usually secured using the same tools that you would use for a table or a view for user access, yet you can allow the stored procedure to run using someone else's database privileges. This allows you to put code in a stored procedure to perform a specific function that the user might not otherwise be able to perform. The user will not be able to see or change this code.

Stored procedures and the Data View Window

In Chapter 9, I talked about how to use the Data View Window (see Figure 13-2) to access your database while using the Data Environment Designer. It's possible to use the Data View Window without the Data Environment Designer. Unlike the Data Environment Designer, the Data View Window isn't integrated into your program. It is a design-time only tool that allows you to access your database design, tables, views, and stored procedures tasks using a database vendor independent tool.

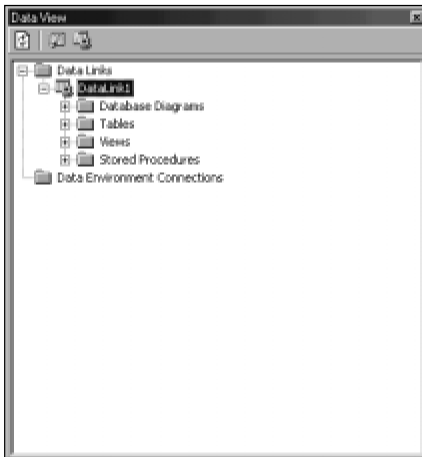


Figure 13-2: Adding the Data View Window to your application

Note

Exclusively Enterprise Edition: The Data View Window features I'm going to cover apply to the Enterprise Edition of Visual Basic. If you have the Professional Edition, you will need to create and debug your stored procedures using tools provided by your database vendor. This isn't entirely bad, as I feel these tools are often better than the ones supplied with Visual Basic.

Creating a Data Link

Before you can use the Data View Window, you need to create a link to the database. After opening the Data View Window, right click on the Data Link icon and select Add a Data Link from the pop-up menu. This will display the same Data Link Properties dialog box that you've seen many times by now (see Figure 13-3). Simply select the proper OLE DB provider for your database, enter the connection information, and press OK to return to the Data View Window.

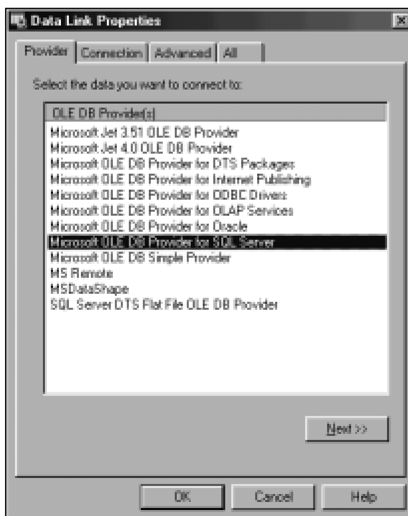


Figure 13-3: Viewing the Data Link Properties window yet again

Creating a stored procedure

If you expand your newly created Data Link icon, you'll see each of the four types of objects available for you to manipulate: Database Diagrams, Tables, Views and Stored Procedures. Double clicking on the Stored Procedures icon will display the list of stored procedures in the database, and right clicking on the same icon and selecting New Stored Procedure from the po-up menu will allow you to create a new stored procedure (see Figure 13-4).

The New Stored Procedure window is a simple editor with a series of buttons across the top of the screen. The same window will be used to edit an existing stored procedure. The only difference is that the name of the existing stored procedure will be displayed in the title bar. The icons that appear in the toolbar are explained below:

- **New Stored Procedure** – prompts you to save your changes and then displays a skeleton stored procedure for you to edit.
- **Open Text File** – asks if you want to overwrite the existing text and then displays a File Open dialog box to load a text file into the edit window.
- **Save to Database** – saves your stored procedure to the database.

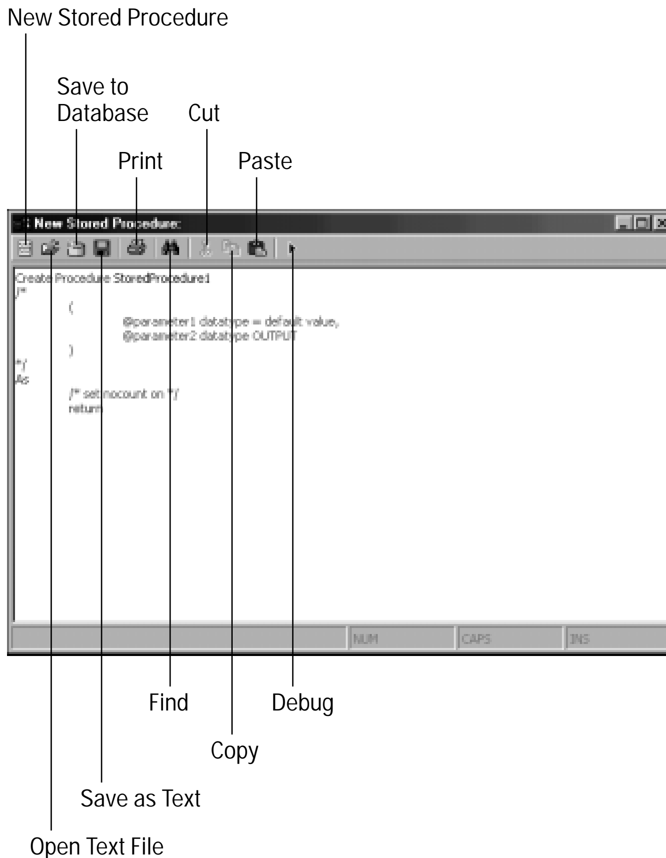


Figure 13-4: Creating a stored procedure

- ✦ **Save as Text** – saves your stored procedure to a text file.
- ✦ **Print** – sends your stored procedure to the default Windows printer.
- ✦ **Find** – displays a Find dialog box.
- ✦ **Cut/Copy/Paste** – performs the standard editing function using the clipboard.
- ✦ **Debug** – starts the stored procedure debugger.

Coding the stored procedure

Writing a stored procedure is highly dependent on the database server you use. The example I'm going to use here is for the SQL Server 7. However, no matter whose database system you're using, all stored procedures have some things in common. First, they are all created using the **Create Procedure** SQL statement, and can be

deleted with the **Drop Procedure** statement. The identifier that follows the **Create Procedure** statement is the name of the stored procedure. Second, all stored procedures accept arguments that allow you to pass parameters to them and return values from them. After the arguments are defined, you enter the statements that comprise the stored procedure.

The Stored Procedure window allows you to build the **Create Procedure** statement that will create your stored procedure. Listing 13-4 shows a stored procedure that performs the same function as the **Select** statement I've been using throughout this chapter in the sample program. It takes a single parameter, `@State` that has a type of **Char(2)**, which is the same type as the State column in the database. The `@State` parameter is used in place of the question mark in the parameterized **Select** statement in the previous example.

Listing 13-4: The CountByState stored procedure

```
Create Procedure CountByState (@State Char(2)) As

Select Count(*)
From Customers
Where State = @State
Return
```

Saving the stored procedure

Pressing the Save to Database button will execute the **Create Procedure** statement on the database server and will create a stored procedure using the name `CountByState`. If you want to edit the stored procedure, simply expand the Stored Procedures icon on the Data View window and double click on `CountByName`. This will display the same SQL statement, with one minor difference. The words **Create Procedure** will be replaced with **Alter Procedure**.

Debugging stored procedures

The T-SQL Debugger allows you to debug stored procedures directly from Visual Basic. This tool only works with SQL Server 6.5 and later databases. The debugger can be called directly from Visual Basic. Before you can use the debugger, you must install the appropriate code on your database server.

Installing the SQL Server debugging support

The setup program is contained in the `\SQDBG_SS` directory on disk 2 of the Visual Basic installation CD-ROMs. You can even run the setup program while your SQL Server database is running.

When you start the setup program, you'll see a dialog box similar to the one shown in Figure 13-5 reminding you that this utility is not part of SQL Server, but part of Visual Basic. If you agree with the license information, press the Continue button.

Figure 13-5: Reviewing license information for the SQL Server Debugging facility

The next few dialog boxes will ask you to verify your name and organization information and to enter the CD Key from the back of your Visual Basic CD-ROM case. Enter the information as requested and press OK or Continue at each step of the wizard until you reach the Installation dialog box shown in Figure 13-6. You can change the directory where the software will be installed by pressing the Change Folder button. When you're ready to begin, press the square Server button. The setup program will then install the debugging support feature.

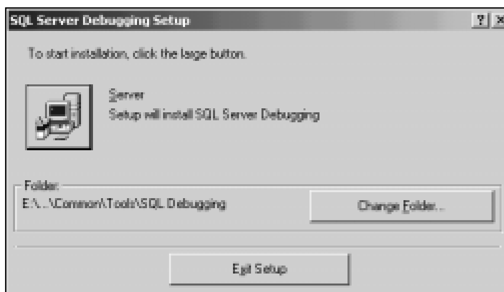


Figure 13-6: Starting the installation program

I got this weird error: If you get a strange error while trying to start the T-SQL Debugger and it instructs you to look in the client log on the database server, most likely the SQL Server Debugging support has not been installed.

Setting T-SQL Debugging Options

Before you use the T-SQL Debugger, you should review the options. To display the options, choose Tools ➪ T-SQL Debugging Options from the Visual Basic main menu. This will bring up the T-SQL Debugging Options dialog box, as shown in Figure 13-7.

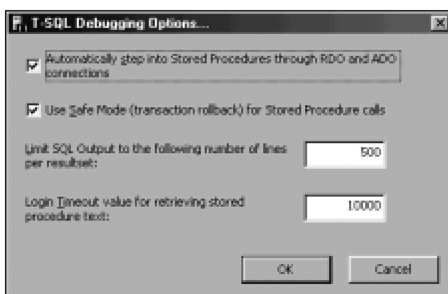


Figure 13-7: Setting debugging options

There are four options you can set. Checking the *Automatically step into Stored Procedures through RDO and ADO connections* will automatically start the debugger anytime you execute a stored procedure while running your Visual Basic program. If this box is not checked, the debugger will not be used at runtime. Checking *Use Safe Mode (transaction rollback) for Stored Procedure calls* means that any changes made by the stored procedure while in debug mode are discarded.

Sometimes it works and sometimes it doesn't: I've noticed that changing these properties don't always take effect the next time you run your program in the IDE. I suggest running your program and ending it before you attempt to do anything. Then run the program again, and the debugger should behave properly.

The *Limit SQL Output to the following number of lines per resultset* determines the upper limit in the number of rows that will be retrieved while debugging the procedure. This value helps to ensure that your stored procedure doesn't run out of control. The *Login Timeout value for retrieving stored procedure text* sets the maximum amount of time the debugger will wait to connect to the database server.

Starting the T-SQL Debugger

You can run the T-SQL Debugger directly from Visual Basic by choosing Add-Ins ÿ T-SQL Debugger from the main menu. This will display the Visual Basic T-SQL Batch Debugger dialog box (see Figure 13-8). You need to provide the information in this window to connect to the database to select and run your stored procedure.

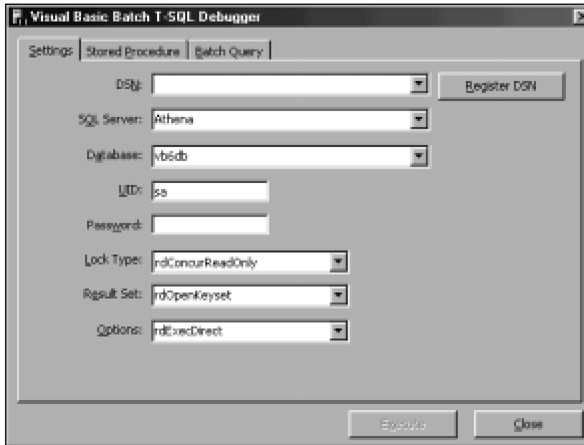


Figure 13-8: Setting options in the T-SQL Debugger window

To connect to an SQL Server database using OLE DB, simply enter the name of your database server in the SQL Server drop-down box, specify the name of the database you want to use in the Database drop down box, and supply your user name and password in the *UID* and *Password* fields. You can also define an ODBC connection by pressing the Define DSN button.

Once these values are set, the Stored Procedure and Batch Query tabs will be enabled. To debug a stored procedure, select the Stored Procedure tab (see Figure 13-9) and choose the stored procedure you want to debug in the Procedure Name drop-down box. Then select each of the parameters listed in the *Parameters* area and assign them the values you want to use during the execution.

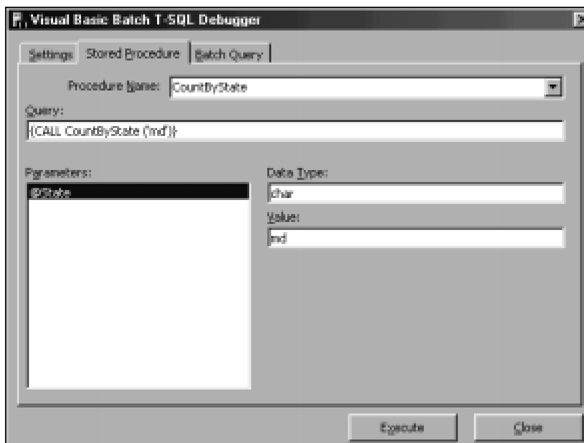


Figure 13-9: Selecting the stored procedure and entering its parameters

Run simple queries: You can use the Batch Query tab in the Visual Basic T-SQL Debugger to run any block of SQL statements you may choose using the T-SQL Debugger.

Running the Debugger

After you have finished entering values for all of the parameters, press the Execute button on the Stored Procedure tab of the Visual Basic Batch T-SQL Debugger to start a debugging session. The T-SQL Debugger window will be displayed (see Figure 13-10).

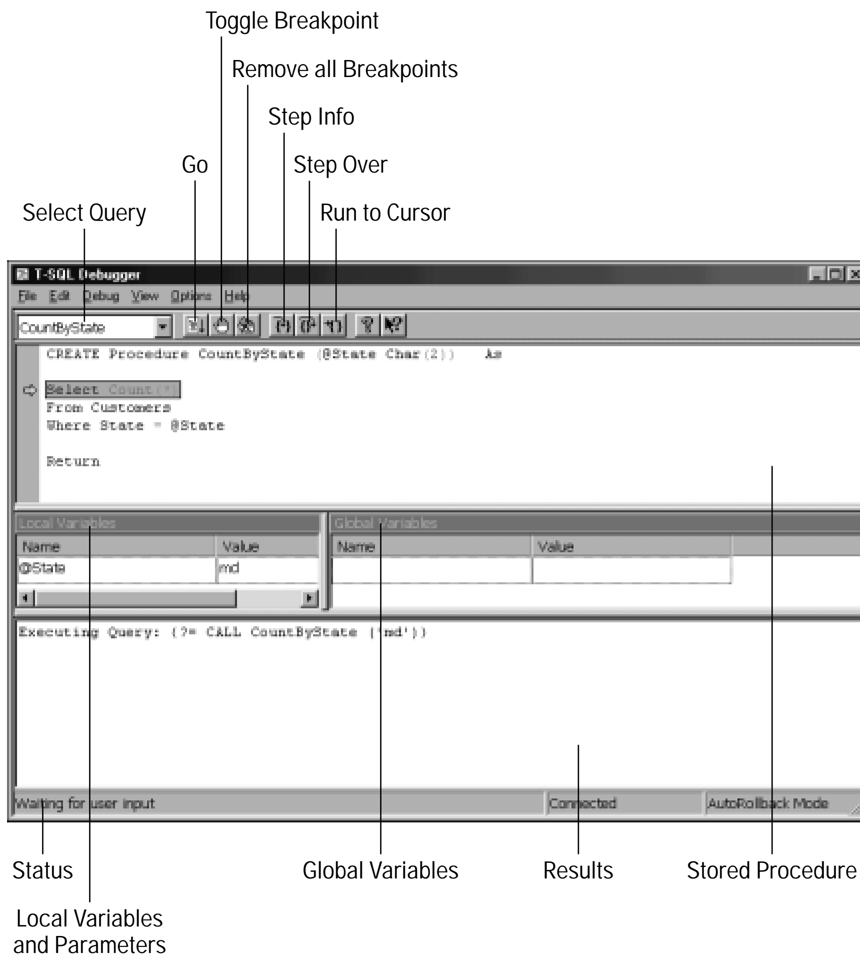


Figure 13-10: Running the T-SQL Debugger

- **Select Query** – allows you to choose from multiple queries you may be debugging at the same time.
- **Go** – runs the query to completion from the current statement or to the next breakpoint.
- **Toggle Breakpoint** – enables or disables a breakpoint at the specified line of code.
- **Removes All Breakpoints** – removes all of the breakpoints in the stored procedure.
- **Step Into** – runs the stored procedure until the specified subexpression is reached
- **Step Over** – runs the selected subexpression.
- **Run To Cursor** – runs the stored procedure up to where the cursor is pointing.
- **Stored Procedure** – contains the text of the stored procedure. This area is used to display the currently active statement and any breakpoints that have been set.
- **Local Variables and Parameters** – this section contains variables local to the stored procedure and their current values. You can edit a value by double clicking on the value to select it and entering a new value.
- **Global Variables** – this section contains the global variables for the stored procedure and their current values. These values may also be changed while the stored procedure is waiting for user input.
- **Results** – displays any rows returned by a **Select** statement and also describes the current state of execution.
- **Status** – describes the current state of the debugger. If the message *Waiting for user input* is displayed, the debugger is in break mode and waiting for you to resume execution.

Calling a stored procedure

As you might expect, defining a stored procedure in a `Command` object isn't much different than defining an SQL statement (see Listing 13-5). In place of the SQL statement, you will need to specify the name of the stored procedure in the `CommandText` property. You should specify `adCmdStoredProc` as the value of `CommandType`.

One advantage of using stored procedures is the ability to automatically retrieve the parameter definitions rather than manually defining each `Parameter` object and adding it to the `Parameters` collection. Simply use the `Refresh` method of the `Parameters` collection to retrieve the definitions from the database. Then you may assign values to each parameter by name as you did when you explicitly defined the parameters.

Listing 13-5: The CountByState stored procedure

```
Private Sub Command6_Click()

Dim c As ADODB.Command
Dim p As ADODB.Parameter
Dim rs As ADODB.Recordset

On Error Resume Next

Set c = New ADODB.Command
Set c.ActiveConnection = db
c.CommandText = "CountByState"
c.CommandType = adCmdStoredProc

c.Parameters.Refresh
c.Parameters("@State").Value = Text4.Text

db.Errors.Clear
Set rs = c.Execute
If db.Errors.Count > 0 Then
   WriteError

Else
   StatusBar1.SimpleText = "Response: " & _
      FormatNumber(rs.Fields(0).Value, 0)

End If

End Sub
```

Thoughts on Stored Procedures

Using stored procedures is very important when building applications for Oracle. They can make a significant difference in how your application performs. However, stored procedures are not as important for SQL Server 7 databases. SQL Server 7 prepares an SQL statement the first time it encounters it during your program's execution and retains it so that the next time you use it, it won't have to prepare it again. This doesn't mean that you shouldn't use stored procedures in SQL Server 7, because they don't make as big of a difference as they do with an Oracle database.

`Command` objects are necessary when you want to execute a stored procedure or any SQL statement other than a **Select** statement. Also, `Command` objects are important when you want to use parameter-based queries. The rest of the time, you can perform the same function directly using the `Recordset` object, a topic that will be covered in the next chapter.

Summary

In this chapter you learned the following:

- You can define a `Command` object to hold a frequently executed SQL statement or stored procedure.
- You can define `Parameter` objects which contain information that is passed to a stored procedure or parameterized query for execution.
- You can easily create and edit stored procedures directly in Visual Basic.
- You can install the stored procedure debugger routines into your database server so that you can debug stored procedures directly from Visual Basic.