CHAPTER 5

ADO Command Strategies

When we get to Chapters 6 and 7, which are all about Recordsets, you’ll learn lots of different ways to execute queries—many of which don’t require use of the Command object. The one important case where the Command object is required is when you have to capture parameters returned from stored procedures. ADO is very smart when it comes to handling stored procedures, so if you aren't returning OUTPUT parameters, and if you don't care about the stored procedure return status, you don't have to construct a Command object.

However, there are advantages to using Command objects. In this chapter, you’ll learn that the Command object leverages new SQL Server and MDAC technology to execute all kinds of queries more efficiently. This means that when you execute rowset-returning or action queries, setting up the Command object can make the process of managing the query and its parameters far more sane. After a Command object is created, you no longer have to worry about putting single quotes around strings or figuring out how to deal with embedded apostrophes. This means you won't have to remove all of the Irish surnames from your database—such as O’Malley or O’Brien. This chapter also discusses how you can tell what your query is doing. We’ll spend a significant amount of time pouring over SQL Profiler logs to see exactly what unnatural acts SQL Server is being asked to perform.

When it comes time to execute your SQL query, the best object to use is often the ADO Command. However, as you’ll learn, it’s not always the best choice. Fortunately, due to ADO’s flexibility, there are other alternatives to draw on, as I discuss when we get to the Recordset object in Chapter 6. One thing you might not know—because it’s underdocumented—is that all ADO Command objects appear as methods on their associated Connection objects. This innovative technique (well, it was stolen from RDO) enables you to code the Command by name, followed by its parameters, followed by the Recordset to contain the rowset. Cool. I discuss how to set this up later in this chapter.

Inner Workings of the Command Object

The Command object’s biggest benefit is performance. Not only does it make your queries run efficiently, but it also makes you work more efficiently. That is, using the Command object can reduce the length of time it takes to code, debug, test, and deploy complex parameter-based queries—including queries
executing stored procedures. For example, when accessing SQL Server 7.0, the ODBC and OLE DB providers have been tuned to access the new `sp_executesql` system stored procedure. There’s quite a write-up on this in SQL Server Books Online,¹ and I have summarized it here.

Basically, the Command object is driven from the CommandType argument that instructs ADO how to transmit your query to the data provider. Suppose you have a parameter-based ad hoc query² that you wish to execute.

```sql
Select author, au_id, year_born from authors where year_born = ?
```

You placed a parameter marker (?) where you want ADO to insert the parameter, so you’re ready to have ADO execute this query. For this query, ADO manufactures an SQL statement that looks like this:

```sql
sp_executesql N'Select author, au_id, year_born from authors where year_born = @P1', N'@P1 int', 1947³
```

The `sp_executesql` system stored procedure was introduced in Microsoft SQL Server version 7.0. The MDAC developers want us to use it instead of the EXECUTE statement to execute a query string. The support for parameter substitution makes `sp_executesql` more versatile than EXECUTE; it also makes `sp_executesql` more efficient because it generates execution plans that are more likely to be reused by SQL Server.

ADO and the SQL Server data provider have also implemented another (proprietary) interface to handle server-side cursors. These are implemented as system-level `sp_cursor` stored procedures that open, fetch, close, and perform various other operations on your data. When using the default server-side CursorLocation setting, you’ll find that many queries are executed using these stored procedures.

---

¹ Books Online is the copious set of help topics and examples that ships with SQL Server in lieu of printed documentation. Most (if not all) of it is also available through MSDN via subscription or online.

² An ad hoc query is simply a hard-coded SQL query or action. Using these queries is not a good idea for performance, maintainability and security reasons. If you can, use a stored procedure instead, but many developers depend on them, at least initially.

³ Note the “N” prefix on the generated code (N'Select au…). This uppercase N indicates that the following quoted string is in Unicode format. Unicode data is stored using 2 bytes per character, as opposed to 1 byte per character for character data. For more information, see “Constants” in Books Online.
Substituting Parameter Values

The sp_executesql procedure and the sp_cursor stored procedures support the substitution of parameter values for any parameters specified in the Transact-SQL string—unlike the (obsolete) TSQL EXECUTE statement. Transact-SQL strings generated by sp_executesql are more similar to the original SQL query than those generated by the EXECUTE statement, which gives the SQL Server query optimizer a better chance to match the Transact-SQL statements from sp_executesql with execution plans from the previously executed statements. This dramatically reduces the need to compile a new execution plan with each execution of your parameter query. That’s good.

With the TSQL EXECUTE statement, all parameter values must be converted to character or Unicode and made part of the Transact-SQL string, as shown in this code example:

```sql
DECLARE @IntVariable INT
DECLARE @SQLString NVARCHAR(500)
/* Build and execute a string with one parameter value. */
SET @IntVariable = 35
SET @SQLString = N'SELECT * FROM pubs.dbo.employee WHERE job_lvl = ' +
  CAST(@IntVariable AS NVARCHAR(10))
EXEC(@SQLString)
/* Build and execute a string with a second parameter value. */
SET @IntVariable = 201
SET @SQLString = N'SELECT * FROM pubs.dbo.employee WHERE job_lvl = ' +
  CAST(@IntVariable AS NVARCHAR(10))
EXEC(@SQLString)
```
If the statement is executed repeatedly, a completely new Transact-SQL string must be built for each execution, even when the only differences are in the values supplied for the parameters. This generates extra overhead in several ways:

- The ability of the SQL Server query optimizer to match the new Transact-SQL string with an existing execution plan is hampered by the constantly changing parameter values in the text of the string, especially in complex Transact-SQL statements.
- The entire string must be rebuilt for each execution.
- Parameter values (other than character or Unicode values) must be cast to a character or Unicode format for each execution.

In contrast, sp_executesql supports the setting of parameter values separately from the Transact-SQL string.

```sql
DECLARE @IntVariable INT
DECLARE @SQLString NVARCHAR(500)
DECLARE @ParmDefinition NVARCHAR(500)
/* Build the SQL string once. */
SET @SQLString = N'SELECT * FROM pubs.dbo.employee WHERE job_lvl = @level'
/* Specify the parameter format once. */
SET @ParmDefinition = N'@level tinyint'
/* Execute the string with the first parameter value. */
SET @IntVariable = 35
EXECUTE sp_executesql @SQLString, @ParmDefinition, @level = @IntVariable
/* Execute the same string with the second parameter value. */
SET @IntVariable = 32
EXECUTE sp_executesql @SQLString, @ParmDefinition, @level = @IntVariable
```

This sp_executesql example accomplishes the same task as the TSQL EXECUTE example shown earlier, but with these additional benefits:

- Because the actual text of the Transact-SQL statement does not change between executions, the query optimizer should match the Transact-SQL statement in the second execution with the execution plan generated for the first execution. Therefore, SQL Server does not have to compile the second statement.
- The Transact-SQL string is built only once.
- The integer parameter is specified in its native format. Conversion to Unicode is not required.
NOTE  For SQL Server to reuse the execution plan, object names in the statement string must be fully qualified.

Reusing Execution Plans

In earlier versions of SQL Server, the only way to reuse execution plans was to define the Transact-SQL statements as a stored procedure and have an application execute the stored procedure. The sp_executesql procedure can be used instead of stored procedures when executing a Transact-SQL statement a number of times—especially when the only variation is in the parameter values supplied to the Transact-SQL statement. Because the Transact-SQL statements themselves remain constant and only the parameter values change, the SQL Server query optimizer is likely to reuse the execution plan it generates for the first execution. Existing ODBC applications ported to SQL Server 7.0 automatically acquire the performance gains without having to be rewritten. For more information, see “Using Statement Parameters” in Books Online.

The Microsoft OLE DB Provider for SQL Server also uses sp_executesql to implement the direct execution of statements with bound parameters. Applications using OLE DB or ADO gain the advantages provided by sp_executesql without having to be rewritten.

The ADO Command object does not use sp_executesql to execute ad hoc queries that don’t have parameters or that simply reference parameter-less stored procedures.

Building Command Objects

Building a Command object takes a little time—both CPU and development time. But consider the alternatives. In the olden days, setting up a query was a lot tougher than what ADO provides for us today. Okay, no “I used to code with keypunch machines with no ribbons…” stories here, but the code you write to create the Command object is a lot easier to create, understand, and support than the ODBC API or even Remote Data Objects ever hoped to be. There are even wizards that will do it for you, so how much easier can it get?

4. I worked with the “Developer Days” people to tune up a wizard they handed out at the conference. It generates “correct” source code for building Command objects and the Properties collection without you lifting a finger—well, almost. It’s on the CD.
TIP  When you create a Command object, it should be created once. That is, create as many objects as necessary—don’t create one and change its properties (other than its parameters) to suit the immediate requirement. At one time there was evidence that ADO was making many round trips to the server to “figure out” how to execute the query. My tests show that this no longer happens in ADO 2.5—at least not always. The entire setup phase seems to be done entirely on the client. But this is still overhead that you don’t have to tolerate more than once.

So, how do you build a Command object? It’s easy:

1. Declare your Command objects in a scope where they can be seen by all of the routines that need to access them. This means when creating client/server applications, create your Command objects at the Module or Form level. However, if you must create a Command object in the middle-tier (Microsoft Transaction Server/COM+), do so in the method procedure. That’s because Command objects can’t be shared across apartments, and using global or class-scope variables in Microsoft Transaction Server simply does not work very well—to be kind.

2. Name your Command object so that it can be executed as a method on the Connection object. Although this is an optional step, it really helps later in the process. Be sure to use a string to name your Command object.

3. Set the CommandText property to the SQL statement in the query, or to the name of a stored procedure, table, or view. ADO is terrible at guessing what goes here.

4. Set the CommandType property to reflect the way you want the CommandText string to be treated. By default, ADO sets this to adCmdUnknown, so don’t make ADO guess what’s best to do.

5. Set the ActiveConnection property to point to the appropriate ADO Connection object. You can’t execute the Parameters.Refresh method or run the query until you do. Once set, the named Command becomes a method on the Connection specified.

6. If the query expects one or more parameters, decide whether to use the Refresh method to get ADO to construct the Parameters collection for you, or build it yourself using the Parameters.Append or Command.CreateParameter technique.
Setting ADO Command Properties

Setting ADO properties is really a matter of knowing what ADO expects. If you're supposed to provide a number, make sure it's in the correct range. Avoid using literals—use the typelib-defined constants instead. It makes for code that is more readable and it's easier to maintain later. If you are setting a property from a TextBox or other control, be sure to reference the correct property explicitly. Don't depend on the default property to work—sometimes it does, more often it doesn't. For example, when setting a Command Parameter, use:

\[\text{Cmd(eParm.NameWanted)} = \text{txtNameWanted.Text}\]

This code depends on the definition of an Enum and a TextBox control. Leaving off the .Text property qualifier can have, well, unexpected consequences. Just don't tell me I didn't warn you. For example, in some cases, ADO (and COM) think you're trying to pass the TextBox object instead of its default Text property.

The Name Property

If you want to use the (cool) “Command as Connection method” technique to execute your Command object, you must name it. It's not a bad idea to do so in any case. If you're executing a stored procedure, the name must match the name of the stored procedure. Otherwise, you're free to name the command after jungle plants if you are so inclined. Just remember to set your name early—before you set the ActiveConnection property. And don't forget to use a String constant or variable to name your Command.

**TIP**  If you pass in an unquoted value instead of a String constant or declared variable, Visual Basic assumes it's the name of a Variant variable that'll have some value later at runtime, unless you have Option Explicit turned on (which you should). In this case, you'll get a healthy Variable Not Defined warning at compile time.

The CommandText Property

This property tells ADO and the data provider what to do. It's the question you want to ask, or it can simply be the name of a table, stored procedure, or even a URL, where the data should be found.
Usually the CommandText is a SQL statement, such as a SELECT statement, but it can also be any other type of command statement recognized by the provider, such as a stored procedure call. Remember to code the SQL in the SQL dialect understood by the data provider. So, if you're connecting to Oracle, you can (and should) use Oracle SQL extensions, just as you can use TSQL extensions when querying SQL Server.

Depending on the CommandType property setting, ADO may alter the CommandText property. You can read the CommandText property at any time to see the actual command text that ADO will use during execution.

You can also use the CommandText property to set or return a relative URL that specifies a resource, such as a file or directory. The resource is relative to a location specified explicitly by an absolute URL, or implicitly by an open Connection object.

**CommandText and Scalability**

But wait. Because a lot of the problems associated with scalability are caused by the CommandText property, it is a good idea to spend some additional time here. Remember that the query you specify in the CommandText property is simply a request for services from the data provider. It’s the provider’s responsibility to perform the physical input/output (I/O) operations to execute this request, no matter how wrong they might seem to a person.

Suppose you called down to the Pizza Hut in the university district in Walla Walla, Washington, and asked for a thousand meat-lover’s and two veggie pizzas (you wanted to feed the Whitman Women’s Soccer team and its supporters). The manager would probably call you back and check your credit rating, veracity, and sanity before starting to process the order. An ADO data provider doesn’t call you back and say, “You’re kidding, right?” when you ask for 1,000 or 10,000 or 10 million rows from a database. It just starts fetching the data and sending it back up the pipe to you. As it arrives, ADO dutifully starts caching this data into RAM and then spools to your hard disk until both are full to overflowing—on the floor behind the computer. No, ADO does not have a “bear with me, I’m kinda new at this” property—it assumes that you know what you’re doing.

**Intelligent Query Authoring or Authoring Intelligent Queries**

After you are connected, it’s time to submit your question to the database engine. That is, you need to submit a query—usually a SELECT statement to return rows, or an action query of some kind to change the data. Improperly designed queries have a greater impact on overall performance than does all other performance factors combined. In other words, if you ask the database engine to do something that takes 5, 50, or 50,000 seconds, no amount of brilliance on the client-side of the query can make the rows start appearing a second earlier. In addition,
improperly designed concurrency constraints can indefinitely block your application from fetching even a single row.

There is a wealth of information available about writing efficient queries, and most of that advice boils down to these guidelines:

- Fetch just the columns you need, and no more. Thus, never use SELECT * even when you want all of the (currently defined) columns. SELECT * might needlessly fetch columns that are expensive to fetch or irrelevant to the task at hand. In addition, SELECT * does not guarantee the order in which columns are returned. That is, if some ambitious systems administrator chooses to alphabetize the table’s columns or to simply insert a new column in the table, SELECT * applications can croak (that’s a technical term).

  One aspect of performance is developer performance. That is, how efficient are the coders working on the application, how many mistakes do they make, and how many times do they miscommunicate their intentions to other developers? SELECT * might seem to address this problem by telling the server to simply return all columns of a result set. However, if the application does not add the code to automatically morph to changes in the underlying schema, you aren’t making developers more productive. Quite the contrary—you are adding work to those who have to figure out what’s wrong when the schema changes.

- Fetch just the rows (records) you need and no more. Scalable applications fetch enough rows to solve the immediate requirement and no more. It’s up to your design to determine what “immediate” means, because there is also a cost if you need to return to the server for more data. Your code needs to balance round-trips with the expense of returning rows that are not, or might never be, needed. Fetching too many rows also increases the amount of locking done by the database. This may hurt the scalability of your application, and it increases the chances for deadlocks. Don’t confuse interactive human-based applications with reporting applications that often have to work with far more rows.

- Incorporate cursors in your application only when absolutely necessary. As you build scalable applications using increasingly complex queries and stored procedures, you’ll discover that ADO can’t build sophisticated cursors against the generated rowsets. We have found that cursorless result sets (ADO’s default behavior) are faster to create and retrieve anyway.

5. Okay, I was raised to use the terms rows and columns for relational databases. The ISAM world uses records and fields. The Microsoft people who wrote ADO apparently like the ISAM terms, so here we are with records and fields. Sigh.
Consider that when working with sophisticated table relationships, it is rarely sufficient to simply add a row to a base table. In many cases, you first have to successfully add rows to foreign-key tables. This implies that simple cursor updates just won’t work and you’ll have to rely on stored procedures or more client-intensive transaction-governed operations.

• Consider using Return Status, OUTPUT, or INPUT-OUTPUT parameters instead of Recordsets (cursors) to retrieve data values. These are considerably (dramatically) faster than having ADO construct a Recordset to return your single row of data.

• If you simply must create a cursor, build a scrollable cursor only when absolutely necessary. Scrollability dramatically impacts performance, as ADO has to run additional code to construct cursor keysets or static rowsets in memory. While this overhead forces your application to incur a considerable one-time expense on Recordset creation, using Dynamic cursors exacerbates the situation by forcing ADO to requery the database repeatedly as you scroll from page to page.

• If you choose pessimistic locking, be careful about the size of the fetched rowset, as all rows in that rowset (and perhaps the pages where they reside) will remain locked as long as the cursor is open—not just when you are editing a row. Don’t use pessimistic locking until you have thoroughly investigated the locking side effects—and have a note from your high-school principal.

• When you initially fetch rows from the server, don’t let the user govern when (or if) the rowset is completely fetched. That is, avoid the practice of fetching and displaying the first row and permitting the user to push a button to fetch the next set of rows. Consider strategies that fetch all of the rows at once. For example, consider disconnecting the Recordset or using the GetRows or GetString methods. However, GetRows might not be such a good idea after all. It seems that it generates quite a bit of overhead as it constructs the Recordset to a Variant array output structure. There are cases where transforming the Recordset to a Variant array makes sense—just don’t send the arrays from tier to tier. Delaying population delays release of share locks on the rows fetched. While this won’t affect your application’s performance, you will be holding back other applications competing for the same data pages. As a general rule, to achieve higher scalability, avoid holding locks on data that will be displayed to the user.

• Don’t run a hard-coded query when you can run a stored procedure. By precompiling your query code into a stored procedure, you can eliminate
the need to wait while the server validates, compiles, and builds a query plan before executing your query.

• When running ad-hoc queries (which you do any time you set the Command object's CommandText property to a string that does not contain the name of a stored procedure), don't set the Prepared property to True. That's because, as far as I'm concerned, it's broken.

• Consider what your application does to the server and other users—not just what your query does to make your client application or component faster. Sometimes you can perform operations from your client that can make the local application run very quickly, while at the same time locking out other users or otherwise making scalability impossible. Scalability and performance don't always go hand in hand, especially when you have a handful of users.

• Be sure to monitor the impact your application has on the server. Use the SQL Profiler to view the low-level operations your code asks the server to perform—remember our discussions in Chapter 4 (see Figure 4.11). Try to balance round-trips with the volume of useful data being returned. The Profiler and the other diagnostic tools we discuss later can clearly show what impact your programs are having on the system and each other—if you learn how to interpret the dumps.

Some of these strategies have to do with making your client respond faster, and others make your overall system faster. That is, some suggestions help create applications that use system resources more efficiently—resources that all clients contend for. This makes all of your applications run faster, be more responsive, and be less likely to lock up while waiting for resources.

**Performance: Opening Command Objects**

As I said earlier, if the query is to be executed more than once, it saves execution and coding time if you build a Command object to manage the query—especially if the query requires parameters. However, in the middle tier and on Web pages, it's not unusual to execute singleton queries and exit—thus, negating the need to create Command objects for better performance. As a matter of fact, you might see a performance advantage if you don't create the Command object in code—especially if you are coding in Visual Basic Script on ASPs where each line of Visual Basic is interpreted. If you're executing against SQL Server, queries that can reuse a previously cached query plan are not recompiled.

Then, again, consider the way Command objects help you manage parameters. Sometimes it's necessary to capture return status or OUTPUT parameters and you don't really have any other choice. And because most queries...
are parameter driven, using Command objects to simply manage the parameters can also reduce the total number of lines of executed Visual Basic code.

**CommandType Property**

If you don’t tell ADO how to interpret and process your command, its own logic takes over and it makes an educated guess as to how to proceed. It’s better (far better) to set the CommandType property to give ADO a suggestion as to how the query should be interpreted. This saves (considerable) internal processing time. Another boost in processing performance can be achieved by using the adExecuteNoRecords option with adCmdText or adCmdStoredProc. This tells ADO that the query won’t be sending back a rowset, so don’t bother constructing a cursor.

The documentation says that if the CommandType property value equals adCmdUnknown (the default value), you may experience diminished performance because ADO must make calls to the provider to determine whether the CommandText property is an SQL statement, a stored procedure, or a table name. The SQL Profiler did not show any evidence of these DDL requests, but some providers might require them. If the CommandType property does not match the type of command in the CommandText property, an error occurs when you call the Execute method. Table 5-1 lists the valid CommandType property settings.

<table>
<thead>
<tr>
<th>CommandType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adCmdUnspecified</td>
<td>Command does not specify the type of query.</td>
</tr>
<tr>
<td>adCmdText</td>
<td>Evaluates the CommandText as a SQL command or an SQL “call” statement.</td>
</tr>
<tr>
<td>adCmdTable</td>
<td>Evaluates CommandText as a table name. ADO simply executes “SELECT * FROM &lt;table&gt;” when you use this option. You had better know what you’re doing to scalability when you do this.</td>
</tr>
<tr>
<td>adCmdStoredProc</td>
<td>Evaluates CommandText as a stored procedure name.</td>
</tr>
<tr>
<td>adCmdUnknown</td>
<td>Default. Indicates that the type of command in the CommandText property is not known, which makes ADO guess, or after it guessed, it still didn’t know.</td>
</tr>
<tr>
<td>adCmdFile</td>
<td>Evaluates CommandText as the file name of a file-based Recordset.</td>
</tr>
<tr>
<td>adCmdTableDirect</td>
<td>Evaluates CommandText as a table name whose columns are all returned. This is only available when working with Jet 4.0 databases and providers.</td>
</tr>
</tbody>
</table>

*Table 5-1: Valid CommandType property settings*

---

6. It’s no longer necessary to call stored procedures using the Call syntax that we used in RDO.
**Trying to Preset Execution Options**

When you construct your Command object, it might make sense to preset one or more of the options that affect how ADO executes the query. For example, you might want to request asynchronous execution by using adAsyncExecute, or an option indicating that the Command does not return rows such as adExecuteNoRecords. Unfortunately, ADO does not let you set these options when you set the CommandType. The documentation says you can, but if you try, you'll get a 3001 runtime error. You have to pass these options to the Execute method until this is fixed.

**The ActiveConnection Property**

If you expect to execute your Command object against a specific data provider (and most of you do), you have to set the ActiveConnection property to a Connection object pointing at that data source. Attempting to execute a Command or even use the Parameters.Refresh method without a valid ActiveConnection property setting results in—you guessed it—a trappable error.

**TIP**  Once you set the ActiveConnection property, the Command object's Name property is frozen. If you want to reference the Command by name, set the Name before setting the ActiveConnection.

Before the Connection is opened, the ActiveConnection property contains a “definition” of the connection, and this definition is simply a connect string. This means that you can set the ActiveConnection property using the same string that you use for the Connection object's ConnectionString property. After the Connection is open, ActiveConnection contains a live Connection object reference. If you wish to execute a Command on a series of Connections, set the ActiveConnection property to Nothing followed by setting it to alternative Connection objects.

You need to set the ActiveConnection property before you try to use the Parameters.Refresh method. If you do use the Parameters.Refresh method to construct the Parameters collection, and you set the ActiveConnection property to Nothing, ADO clears the Parameters collection (for some reason). However, changing the ActiveConnection property has no effect on the Parameters collection if you construct in code. Closing the Connection object sets the ActiveConnection property to Nothing for each Command object associated with the Connection (how rude). Setting the ActiveConnection property to a closed Connection object generates an error (duhh).
The (So-called) Prepared Property

In theory, the Prepared property was designed to reduce work on the server by precompiling ad hoc queries so that subsequent executions would use a temporary stored procedure instead of repeating the compile phase each time the query is executed. However, this is not the case with ADO's implementation—keep reading.

Since ODBC was invented some years ago, SQL Server has gotten a lot smarter—it now knows how to leverage existing (in cache) compiled query plans. That is, once you execute a query from ADO (or by any means), SQL Server constructs a query plan, saves it in the procedure cache, and executes it. When the query is done, SQL Server marks the query plan as “ discardable” but leaves it in memory as long as it can. When another identical (or close-enough) query comes in, which is very likely in systems running multiple clients, SQL Server simply reuses the cached plan. This saves a significant amount of time and greatly improves scalability. It makes SQL Server actually faster as more users are added—assuming they are doing about the same things with the same set of queries.

ADO and its ODBC and OLE DB data providers know about this strategy, and in most cases they leverage it by executing sp_executesql, which takes advantage of this feature. However, doing so puts the Prepared property in a quandary. It insists on creating temporary stored procedures, but the data providers insist on using sp_executesql. The result? Chaos. I describe what happens a little later in the chapter in my discussion of executing Command objects.

My recommendation for the Prepared property is this: forget it—at least for SQL Server. For other providers, set up a trace that shows exactly what’s going on—what the server is being asked to do.

The CommandTimeout Property

The CommandTimeout property indicates how long to wait (in seconds) while executing a command before terminating the attempt and generating an error. Remember that CommandTimeout starts when the database server has accepted the command and ends when the database server returns the first record. If the server or network is busy (or not answering at all), this setting won’t help you regain control.

It’s important to set this value based on reality. That is, if, based on repeated full-load testing, you know that the query takes a long time to run, then set the CommandTimeout accordingly—and add a fudge factor. Remember that the server or network can delay things as load changes or when the database has to allocate more disk space or other resources. The default of 30 seconds might not
be enough. However, don't be tempted to set this to 0, which disables the timeout. You don't want your application to freeze, waiting for a query that'll never end.

Twice this week I've helped users with timeout errors. Both discovered that increasing the timeout did not solve their problem. Timeouts are caused by the inability of the data provider to complete the requested operation in the specified number of seconds. While the causes that prevent completion are numerous, one of the most common is locking. That is, when one application is holding a lock on a page, row, or table, other applications (or even different parts of the same application) are unable to access that data. Any attempt to access the data has to wait until the locks are released. In a well-tuned system, this usually takes a couple of seconds or so. If the offending application does not release the locks (perhaps it's waiting for other locks to be freed) the other applications attempting to access the data are blocked indefinitely. Before looking far and wide seeking out the guilty party, I think you'll find that all too often your own application is holding the lock. For example, if you open an updatable Recordset and try to update the database using an UPDATE action query, you'll discover exactly what I'm talking about.

In RDO (and in the ODBC API), there was an option that was lost on conversion to ADO—retry on timeout. That is, if you wanted to keep waiting after a Command timeout, you could simply pass a flag back to the event handler and keep waiting. This is not implemented in ADO. Why? BHOM (another “technical” term I learned in the Army that means “beats the hell out of me”).

NOTE  ADO timeout settings are independent of network timeout. Because the low-level network driver makes a synchronous network API call, and because this call does not return until the network timeout expires, the ADO timeout code is blocked.

Handling Parameter-based Queries

Most queries that you execute require one or more parameters to govern the rowset created. These parameters are usually applied to the WHERE clause of a query, but they can be used in a variety of other ways. When you construct any parameter-based query, you have to describe the query parameters one way or another and supply the runtime values, but you don’t have to use the Command object—not unless you expect to have to deal with a stored procedure return
status or with output parameters. You can use other techniques to pass input parameters, and I discuss these other techniques next. Basically, there are several approaches that you can take when constructing parameter queries:

- Construct a Command object in code, which exposes an empty ADO Parameters collection. This approach can generate the Parameters collection using the Refresh method or by constructing the Parameters one-by-one.

- Construct an SQL statement that includes the parameters in the query string. This approach can construct a sp_executesql query instead of having ADO construct one for you.

- The Visual Database Tools, including the Data Environment Designer, can also construct parameter-based SQL statements and expose these as Data Environment Designer-based Command objects. These are discussed more completely in the *Hitchhiker’s Guide to Visual Basic and SQL Server*. In this book, I stay focused on ADO coding.

- Use the Visual Basic Addin supplied with this book that generates the code to construct Command objects.

### How ADO Command Objects Manage Your Parameters

When you use the ADO Parameters collection to manage your parameters, it’s ADO’s responsibility to get these parameters crammed into the query in the right places and in the right format. ADO is also responsible for dealing with “framing” quotes. That is, if the parameter has embedded single-quotes to delineate strings (most do), ADO will automatically double these up (replacing a single quote with two single quotes). This way your query won’t die the first time you work with an Irish surname, such as O’Malley or O’Brien.

When working with Command objects, it’s your responsibility to describe the parameters correctly, unless you use the Refresh method. This means constructing the Parameters collection one parameter at a time, in the order the data provider expects them. No, ADO and its data providers do not support “named” parameters (not until version 2.6), so until then, you *have* to specify them in the right order. Knowing how to describe the Parameter datatype, size, precision, scale, and shoe size is your responsibility. If you get it wrong, you’ll get an error. If you get them out of order, who knows what will happen….

---

7. This Visual Basic Addin is also provided to MSDN subscribers—or should be by the time this book is available.
There are a number of techniques that will make your parameter-based queries more efficient and easier to construct. One of these is the Visual Basic AddIn supplied on the CD that constructs the Visual Basic code required to open a connection, build the Command object, construct the Parameters collection and execute. Eventually, all of your production applications will evolve to depend on parameter queries to both improve performance (both system and developer performance) and to simplify the development of component-based designs.

**TIP** No, you don’t necessarily need to construct a Command object to execute a parameter-based query. However, if you don’t take advantage of `sp_executesql` when it makes sense, I suspect your query performance might be disappointing.

**Constructing the Parameters Collection**

The ADO Command object’s Parameters collection manages all flavors of parameters: `gazintas` (input), `gazouta` (output), and `gazinta-gazouta` (input-output–bidirectional) parameters. Remember that input parameters can be applied to ad hoc queries as well as to stored procedures. The trick is learning how and when to construct the Command object’s Parameters collection in code. As I said before, there are two approaches:

- Use the `Command.Parameters.Refresh` method to get ADO and the associated provider to construct the Parameters collection for you based on the `CommandText` you provide.

- Construct the `Command.Parameters` collection yourself—parameter-by-parameter—based on your understanding of how the parameters are defined.

Each technique has its advantages and disadvantages in terms of developer and application performance.

IMHO (another technical term that means “in my humble opinion”), neither technique should be used from the middle tier if you can help it. Why? Well, consider that the time taken to execute the extra code to build the Command object and the appropriate Parameter objects (one at a time) is wasted. It has to be re-executed each time the ASP is referenced or the MTS component is executed. Yes, the Command objects make the process far simpler to code. If you are looking for a simple solution with somewhat limited scalability, then keep reading.
Using the Refresh Method

The Command.Parameters.Refresh method technique seems to do all of the work for you—it constructs the Command object's Parameters collection for you in a single line of code. That's good and bad. It's good in the sense that you don't have to worry about how ADO creates the Parameter objects (for the most part, that is—it usually gets them right). It's bad in that ADO and the provider take a round-trip to the server to figure out how to do so, which can be costly (as we have already discussed). However, because this can be a one-time performance hit early in the life of the application, it might not make that much difference.

Remember to set the ActiveConnection property before attempting to use the Refresh method—ADO needs a valid connection to hit the database to generate the parameters.

TIP Actually, you don’t even have to use the Refresh method if you don’t want to, and using it might even cause ADO to execute an extra round-trip. When you try to read a property of an uninitialized Command.Parameters collection for the first time, ADO constructs the Parameters collection for you—just as if you had executed the Refresh method.

After executing the Parameters.Refresh method, you still might need to revisit some of the Parameter objects. For example, if your stored procedure expects OUTPUT parameters, the providers have a tendency to tell ADO to set the Parameter.Direction property to adParamInputOutput instead of adParamOutput. This is not disastrous, because you can simply provide empty, null, or default values when calling the stored procedure.

An advantage (sort of) to the Parameters.Refresh method technique is that if someone changes the number, position, or datatype of the parameters, your application can automatically morph to the new parameter requirements. Of course, if these changes are significant, your code might very easily pass in wrong values to these parameters. Your code references Parameter objects by position—not by name. Suppose the datatype of parameter four changes from SmallInteger to Integer—no big deal. But if it changes from VarChar to VarBinary…that’s another matter.

Constructing the Parameters Collection in Code

The second technique also has its good and bad points. Constructing the Parameters collection in code assumes that you understand how ADO and the
called procedure expect the Parameter objects to be created. This also assumes you understand ADO well enough to know how to code the Parameter objects yourself and that the definitions for the parameters are not subject to change.

Just choosing the right datatype for each object can be tricky—and there is often more to constructing Parameter objects that makes the task even more complex. That’s why I often defer to the Data Environment Designer or the Parameters.Refresh method to do this for me. Although the Data Environment Designer makes the same mistakes that ADO makes when building parameter collections, using it can save quite a bit of guessing on my part. To leverage the prefabricated Parameters collection, I use the Data Environment Designer to construct a DataEnvironment object and copy out the generated settings for each Parameter property into my code—even the individual parameter object data types.

The Other Side of the Refresh Method Story

Let’s see what ADO does when you use the Refresh method—especially in conjunction with the so-called Prepared property. There are a couple of important points not covered in the documentation. For example, in the following code, I used the Refresh method to build the Parameters collection.

```vbscript
With cmd
    .Name = "GetTitles"
    .Prepared = False
    .CommandType = adCmdText
    .ActiveConnection = cn
    .CommandText = "Select title from titles " & "where title like ? " & "and year_published between ? and ?"
    .Parameters.Refresh ' let ADO create the Parameters collection

According to the SQL Server 7.0 Profiler, when the Refresh method executes, the provider (SQL Server in this case) is asked to execute these two queries (two round-trips):

```
SET FMTONLY ON select title, year_published, year_published from titles SET FMTONLY OFF
declare @P1 int
set @P1=NULL
sp_prepare @P1 output, N'@P1 varchar(255),@P2 smallint,@P3 smallint', N'Select title from titles where title like @P1 and year_published between @P2 and @P3', 1
select @P1
```

```
However, when the application executes:

```vbnet
cmd(eParms.TitleWanted) = "Hitch%"
cmd(eParms.YearHigh) = 1950
cmd(eParms.YearLow) = 1999
Set rs = New Recordset
rs.Open cmd
```

The profiler tells us that the SQL Server data provider destroys (sp_unprepare) the temporary stored procedure (1). ADO and the SQL Server provider then proceed to use sp_executesql to run the query:

```
sp_unprepare
sp_executesql N'Select title from titles where title like @P1 and year_published between @P2 and @P3', N'@P1 varchar(255),@P2 smallint,@P3 smallint', 'Hitch%', 1999, 1950
```

Each subsequent execution of the Command object, regardless of syntax, simply generates a call to sp_executesql with the new parameters.

```
sp_executesql N'Select title from titles where title like @P1 and year_published between @P2 and @P3', N'@P1 varchar(255),@P2 smallint,@P3 smallint', 'Any%', 1999, 1950
```

Okay, that's not so terrible (only one unnecessary tear down and reconstruction), but what if we change Prepare to True? In this case, ADO and the SQL Server data provider seem to get confused. Just as before (when Prepare was left as False), the DDL queries are used to construct (sp_prepare) the temporary stored procedure when you use the Refresh method. Nothing has really changed. However, if you use the Command as Connection method technique, ADO uses the existing temporary stored procedure, but each subsequent execution tears down the temporary stored procedure and reconstructs it.

**Constructing the Parameter Collection in Code**

But what happens if you construct the parameters yourself—in code—instead of using the Refresh method? Well, if you set Prepared=True, then ADO follows about the same path as before, but instead of creating the temporary stored procedure with sp_prepare when the Refresh method is executed, it's now executed when the Command is executed for the first time, as expected. And ADO also reuses this temporary stored procedure until you use the Command as Connection method technique, when it reverts back to its old bad habits.
However, if you construct the Parameter collection in code and don't set the Prepared property to True (it defaults to False), ADO knows just what to do—it simply constructs sp_executesql statements to run your query. That much it knows how to do. No extra round-trips to set up the temporary stored procedures just to tear them down again. It just runs the queries.

```
sp_executesql N'Select title from titles where title like @P1 and year_published between @P2 and @P3', N'@P1 varchar(20),@P2 int,@P3 int', 'Hitch%', 1999, 1950
sp_executesql N'Select title from titles where title like @P1 and year_published between @P2 and @P3', N'@P1 varchar(20),@P2 int,@P3 int', 'Any%', 1999, 1950
sp_executesql N'Select title from titles where title like @P1 and year_published between @P2 and @P3', N'@P1 varchar(20),@P2 int,@P3 int', 'Hitch%', 1940, 1999
```

The SQL Server team knows about this Prepared property bug, and while SQL Server 7.0 SP2 has not (so far) completely addressed this problem, it has gone a long way toward doing so. After SP2 is applied, this aberrant behavior is only triggered when you use the Command as Connection method technique. While I like the simplicity and flexibility of this technique, I use it cautiously until this bug is fully resolved.

This silliness prevents the use of the Refresh method as a best practice, especially when coupled with the Prepared property setting of True. If you build your Parameters collection in code, ADO does not make these mistakes.

So, in summary, how can you prevent these problems—at least until Microsoft fixes the bug?

- Don't set the Prepared property to True. It only makes matters worse.
- Construct the Parameters collection for ad hoc queries in code.
- Don't use the Refresh method for adCmdText commands. Without it, ADO uses the sp_execute strategy for your queries. Because SQL Server seems to be optimized for the sp_execute strategy, this should help, not hurt, performance.
- Be careful if you use the Command as Connection method.
- Keep an eye on the Profiler to see whether your queries are generating extra sp_unprepare and sp_prepare operations.
- Consider using the stored procedure as Connection method technique discussed in the Recordset chapter or other strategies that don't use the Command object at all.
The Parameter: Direction Property

The Parameter object’s Direction property tells ADO how and when to expect data for this parameter. The default is adParamInput, but as shown in Figure 5-1, there are several intuitive alternatives:

- **adParamReturnValue**: If declared, the return value parameter is the first in the Parameters collection. It’s used to receive the stored procedure’s return status integer.

- **adParamInput** (the default): The parameter is passed to the query at runtime.

- **adParamInputOutput**: The parameter is passed to the query at runtime, and the same parameter is used to receive a value from the stored procedure.

- **adParamOutput**: The parameter is used to receive a value from the stored procedure.

- **adParamUnknown**: If ADO cannot figure out how to deal with a parameter (or you can’t), it gets set to this “unknown” value.

```vba
With cmd
    .Parameters.Append .CreateParameter ["RetStatus", adInteger]
End Sub
```

Figure 5-1: Direction property enumerated constants

The first Parameter object in the Parameters collection is special. It can hold either the return status from a stored procedure or the first parameter. Only the first Parameter object in the Properties collection can be set to adParamReturnValue. This position is reserved for stored procedures that return an integer status value (yes, it’s always an integer). If you are executing an ordinary parameter query, not a stored procedure, you can define the first argument as adParamInput. For stored procedures, you can ignore the return status by setting the first (ordinal 0) Parameter object’s Direction property to any other direction enumeration constant.
WARNING  The Visual Basic Locals Window enumerates the Parameters collection using one-based referencing. That is, cmd.Parameters(0) is listed as Parameters(1). Swell.

The Parameter .Type Property

When working in the Visual Basic integrated design environment (VB IDE), the statement completion feature can prompt you with a list of valid datatypes. The trick is to choose the right datatype. For instance, the other trainers, support staff, and I get quite a few questions on the Parameter object’s Type property, which describes its datatype. Figure 5-2 shows how the VB IDE tries to prompt you, but there are many data types to choose from. By far the easiest way to determine the correct datatype to use is to ask ADO—at least initially. That is, use the Refresh method to populate the Parameters collection and examine the Type property settings for each parameter. While you’re at it, also check the Size, Precision, and NumericScale property settings. Actually, there’s an even easier way and it does not require any code. Use the Data View window to call up a table that has a field whose datatype you want ADO to access. Right-click the field in question, and the correct ADO-enumerated datatype constant will appear like magic.

```vbscript
With cmd
    .Parameters.Append .CreateParameter("RetStatus")
End Sub
```

Figure 5-2: The VB IDE prompting with a list of ADO datatypes
Table 5-2 lists all of the SQL Server datatypes and some to watch out for. Checkout the datetime datatype—yep, you have to pass it as adDBTimeStamp. The code for generating this table is on the CD, so you can run it against your own data provider.

<table>
<thead>
<tr>
<th>SQL SERVER DATATYPE</th>
<th>ADO PARAMETER TYPE</th>
<th>PROPERTY</th>
<th>PRECISION</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RETURN_VALUE)</td>
<td>adInteger</td>
<td></td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Varchar</td>
<td>adVarChar</td>
<td>(You decide)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Char</td>
<td>adVarChar</td>
<td>(You decide)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int</td>
<td>adInteger</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Smallint</td>
<td>adSmallInt</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tinyint</td>
<td>adUnsignedTinyInt</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Datetime</td>
<td>adDBTimeStamp</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Smalldatetime</td>
<td>adDBTimeStamp</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Bit</td>
<td>adBoolean</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>adVarChar</td>
<td></td>
<td>2,147,483,647</td>
<td></td>
</tr>
<tr>
<td>Image</td>
<td>adVarBinary</td>
<td></td>
<td>2,147,483,647</td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>adVarBinary</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Varbinary</td>
<td>adVarBinary</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal</td>
<td>adNumeric</td>
<td>18</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Smallmoney</td>
<td>adCurrency</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Money</td>
<td>adCurrency</td>
<td>19</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Numeric</td>
<td>adNumeric</td>
<td>18</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>adSingle</td>
<td>7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td>adDouble</td>
<td>15</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Nchar</td>
<td>adVarWChar</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Ntext</td>
<td>adVarWChar</td>
<td>1,073,741,823</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nvarchar</td>
<td>adVarWChar</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-2: SQL Server data type and the correct ADO-enumerated datatype constants

**The Parameter.Size Property**

Use the Size property to determine the maximum size for values written to or read from the Value property of a Parameter object. If the size of the parameter exceeds this upper limit, you’ll trip a trappable error.

---

8. The reason (rationale) for the use of the TimeStamp datatype instead of something more reasonable (adDate|Time, or some such) is so thin it’s not worth mentioning.
Yes, some datatypes don’t require that you set the Size property. You do have to set it for Char and VarChar fields—where the size can vary. ADO and the data provider returned the default size properties shown in Table 5-2 (except for the Char and VarChar). Note that the (maximum) size for the NText (Unicode Text) is half the size of its equivalent ANSI Text datatype.

**The Parameter.Precision and Parameter.NumericScale Properties**

The Precision and NumericScale properties are used to describe numbers that have decimal points or varying degrees of accuracy. For numbers capable of storing a fractional value, use the NumericScale property to determine how many digits to the right of the decimal point are used to represent values. The Precision property is used to determine the maximum number of digits to be stored for any number—integer or floating point. These properties have to be set in code because they aren’t supported by the CreateParameter method.

**Constructing Parameter Objects with CreateParameter**

You can create individual Parameter objects and add them to the Parameters collection using the Append method.

```vbscript
Dim Pr as Parameter
Set Pr = New Parameter
With Pr
    .Name = "P1"
    .Type = adVarChar
    .Direction = adParmInput
    .Size = 30
    .Value = "Fred graduates in 2000"
End With
Cmd.Append Pr
```

If you charge by the line, this is the best approach—but it’s a lot easier to do all of this with a single line of code.

```vbscript
.Parameters.Append .CreateParameter("P1", adVarChar, adParamInput, 30, _
    "Fred graduates in 2000")
```

The CreateParameter method constructs the entire Parameter object in a single step. It does not include the NumericScale or Precision properties, but it does include the other essential properties—including Value. Because this method reminds you of all of the essential properties, prompting you for each one as you type in the Visual Basic IDE, its use qualifies as a best practice.
**Rolling Your Own Parameter Queries**

No, you don’t have to use the Parameters collection to execute a parameter query, or queries without parameters for that matter. For instance, you can construct a query in a string and pass it to the Source property of a Recordset or the CommandText property of a Command object. You can also execute any stored procedure as a method of the Connection object. The stored procedure parameters are simply passed as method arguments.\(^9\) If the query has parameters, you can concatenate parameters into the WHERE clause. If the stored procedure returns a Recordset, you pass that as the last argument, as shown below:

```vbscript
Set Rs = New Recordset
Cn.MySP "My parm1", 2, "O'Malley", RS
```

Here’s an example that executes a parameter query without (apparent) use of the Command object at all—it simply sets the Recordset object’s Source property to a constructed SELECT statement. Actually, behind the scenes ADO constructs a Command object to manage the query.

```vbscript
Set rs = New Recordset
With rs
  SQL = "Select title, Year_Published from titles " & 
    "where title like " & txtParm1 & 
    "and year_published between " & txtParm2 & " and " & txtParm3
  .ActiveConnection = cn
  .Source = SQL
  .CursorType = adOpenStatic
  .LockType = adLockOptimistic
  .Open Options:=adCmdText
End With
```

The data provider is managed very differently depending on where the cursors are created. If a client-side cursor is selected prior to opening the Connection, then only these two SQL statements are sent to the server:

```
SET NO_BROWSETABLE ON
Select title, Year_Published from titles where title like 'Hitch%' and
year_published between 1900 and 1999
```

---

9. This "stored procedure as Connection method" technique is discussed in Chapter 6.
When rows are needed from the client-side cursor, ADO simply fetches them from locally cached data loaded from the single SELECT statement. However, if server-side cursors are used, the ADO instructs the data provider to construct a server-managed cursor. When rows are needed, the provider runs another query to fetch rows from the cursor.

```sql
sp_cursoropen @P1 output, N'Select title, Year_Published from titles where title like ''Hitch%'' and year_published between 1900 and 1999', @P3 output, @P4 output, @P5 output select @P1, @P3, @P4, @P5
sp_cursorfetch 531488860, 16, 1, 2048
sp_cursorfetch 531488860, 16, 1, 1
sp_cursorclose 531488860
```

Notice that the provider submits queries that execute stored procedures you might not see documented in Books Online—that’s because they aren’t documented. These `sp_cursor...` stored procedures are installed by the routines that set up the ODBC and OLE DB data access providers. Their source is encrypted so that you can’t see what magic the provider is doing under the covers to implement your cursor. When I get permission, I’ll document these new system stored procedures and publish the results.

**What’s Tough to Parameterize**

While inserting constants into an SQL query’s WHERE clause is easy, parameterizing other parts of the query is not so easy. Unless you use the concatenation technique I just discussed, changing table, field, or join syntax requires construction of a special type of query that’s compiled each time it’s run. That’s a TSQL subject for another book.

**Setting Parameter Values**

At one point or another, you’re going to want to pass a constant or variable to the query in one or more Parameters. The Value property serves to pass data to and from the procedure—depending on the Direction property setting. That is, before the Command object is executed, you must set the Value property for any input parameter that does not have a default value defined in the stored procedure, which means you have to provide a runtime parameter value for all ad hoc query parameters.
Chapter 5

TIP  We’ll discuss how to use and set default values in properties. See the “Using the Execute Method on a Command Object” section later in this chapter.

Note that when you first set up your Command object, you can provide an initial Value property setting. However, this value is not read-only when the Command object is created, so it’s not a true default value. When this value is set, though, ADO applies this value if a specific value is not provided before execution of the Command.

You can also provide default value settings when you write stored procedures. The following example illustrates coding a SQL Server stored procedure whose first argument has a default setting. That is, if a parameter is not passed to the stored procedure, the default value is used.

Create Procedure DefaultValues
   (@InputParm VarChar(20) = 'Fred', @InputParm2 integer)
As
   SELECT @InputParm Stuff
   return @InputParm2

The Command object’s Value property can be referenced by name. ADO will name the Parameter objects for you as Param1, Param2, and so on, but remember that referencing any collection item by name is slower than referencing it by ordinal, as shown next. You can also name the Parameter objects if you use the (suggested) roll-your-own Parameters collection technique.

Set cmd = New Command
With cmd
   .ActiveConnection = cn
   .CommandText = "DefaultValues"
   .CommandType = adCmdStoredProc
   .Parameters.Refresh
   Set rs = .Execute
   Debug.Print .Parameters(0)  'return status
   Debug.Print .Parameters(1)  'Input param (default value?)
End With

What Happens When the Value Is Wrong

ADO has a fairly rude way of handling Field-constraint violations. Okay, so I’m a little sensitive, but when you assign a string containing 22 bytes to a Field or
Parameter you have described as VarChar(20), ADO trips a trappable error. I expected ADO to simply truncate the string, but I guess this is a better approach—you’ll know about any violations of this field constraint. It also means you’ll have to tighten up your parameter validation routines. The same thing happens when you pass an invalid date—February 30, 2000 for example, or an invalid integer (one that is too large, or too small, or that contains a decimal component).

No, you don’t have to provide values for stored procedure parameters that have default values defined. This means that you might not have to supply any parameters. If you don’t care about capturing return status or output parameters, you don’t even have to code a Command object at all—just execute the stored procedure off of the Connection object.

Cn.spAllDefaultValues

The Parameters.Item(n).Value property is the default property for the Command object. So, to reference individual parameter values you can code:

MyCommand(3) = "George has a baby"        ' Notes Parameter

instead of:

MyCommand.Parameters.Item(3).Value = "George has a baby"    ' Notes Parameter

I discuss efficient COM object referencing later, but suffice it to say that while string referenced parameters are easier to read, they are almost twice as slow as ordinal references. This means that neither of the following alternatives is a good idea if runtime performance is the goal:

MyCommand("Notes") = "George has a baby"    ' Notes Parameter
MyCommand!Notes = "George has a baby"    ' Notes Parameter

So how do you add back in the human readability? One approach is to use enumerations for your parameters. So, if you had the following Enum defined:

Enum eParms
    ReturnStatus
    PName
    PGrade
    Notes
End Enum
you could refer to individual Value properties of selected Parameter objects as shown next:

MyCommand(eParms.Notes) = "It's a girl, and her name is Mary Anne!"
' And yes, ADO deals with the extra apostrophe

Referencing Field and Parameter objects by Enum or ordinal (with comments) is another best practice, as this improves human readability as well as performance.

If you don't provide the required Parameter values, ADO informs you with a trappable error. For example, I provided one of the three required parameters for an ad hoc query and got the following error (Err.Description):

[Microsoft][ODBC SQL Server Driver][SQL Server]Prepared statement '(@P1 varchar(255),@P2 smallint,@P3 smallint)Select title from ti' expects parameter @P2, which was not supplied.

Considering that my query looks like this:

"Select title from titles where title like ? and year_published between ? and ?"

you might see where some developers would be as confused as ADO appeared to be. Well, because I asked that the query be "prepared" (Prepared = True), this Prepared statement syntax was not entirely unexpected. To add to the confusion, the error handler (in the data provider) truncated the query slightly (cutting it off shortly after the From clause). To prepare the query, ADO constructed a temporary stored procedure to execute the query—you can see the SQL for that query echoed in the error message. Make sense? I hope so.

**Fetching Parameter Values**

The return status from a stored procedure and the parameters that you marked as output (adParamOutput) or input/output (adParamInputOutput) are available, but only after the data provider sends them. This means that you won't see them until your Recordset reaches (EOF), unless the low-level protocol returns them earlier (not usually the case). Remember, the return status is always in the first (0th) Parameter in the Parameters collection—if you asked for it.

MyRetStatus = cmd(0)          ' Return Status parameter
**Handling OUTPUT Parameters**

ADO does not seem to know how to construct OUTPUT parameter queries very well. Because it can't really guess how you are going to use the parameters in code, ADO assumes that simple OUTPUT parameters are really INPUT-OUTPUT parameters. Because of this, you usually have to change the Direction property to adParamOutput for each of your OUTPUT parameters. Because you're constructing your parameter collection using code, this should not be a problem. Of course, you could rewrite the stored procedure to pass SELECT rowsets instead of OUTPUT parameters....

It's really easy to make a mistake because the documentation for the process of setting up the Parameters collection leaves something to be desired. Also consider what happens when your database administrator changes the procedure in question to accommodate some other developer. If you have hard-coded the datatype for a particular procedure argument and it changes in the database, you have to recode, recompile, and redeploy—perhaps just before you resign. Just remember to let the air out of the SA's tires before you leave the parking lot.

**Parameter-based Ad Hoc Queries**

You don't have to construct stored procedures or even ADO Command objects to execute parameter-based queries. Instead, you can construct an SQL query containing question marks (?) as parameter markers that ADO can handle for you—if you describe the parameters correctly in code. This saves you coding time.

Even though ADO can construct a prototype Parameters collection for you, you still have to add code to complete the process. For example, if you manage the parameters yourself in code, you still have to make sure that any string-based parameters do not contain single quotes (the SQL "framing" quote). You also need to set the ActiveConnection property before ADO will be able to construct the Parameters collection for you; yes, you guessed it, ADO has to query the database to construct the prototype Parameters collection. After you set the ActiveConnection property, you must visit each Parameter object in the Parameters collection and set any needed Type (the datatype), Direction (the default is adParamUnknown), Precision, NumericScale, and Attributes.

---

10. Unless you have the new "Read programmer's mind" add-in—available at a slight extra charge.

11. If your database is redefined regularly, then using the Parameters.Refresh method has more appeal.
Generating Parameters Collection Code Automatically

You can use a new Visual Basic Addin to make the process of generating correct code for parameter-based stored procedures easier. This code started circulating in the developer community after Developer Days 1999. While this addin will not automatically morph your code to changing procedure parameters, it does virtually eliminate the need to guess how to construct the Parameters collection in code. By simply providing enough information to get connected to your database and identifying which stored procedure to reference, you can have the addin write the needed code to construct the Command object's Parameters collection. Yes, you still need to go into the code and make sure that OUTPUT parameters are correctly coded, but the rest is automatic. This addin is provided on the CD.

TIP When working with active server pages (and Visual Basic Script), you might weigh the extra cost of compiling the source code required to construct the Parameters collection against the cost of using the Refresh method. It might turn out that the round-trip costs less than compiling and executing 40 lines of code used to construct a large Parameters collection.

Coding Command Objects for Oracle Stored Procedures

I don't think there's anything I missed in regard to using Command objects to run SQL Server stored procedures. We've discussed input, output, and input/output, as well as return-status parameters. When we get to the Recordset section of the book (in Chapters 6 and 7), I discuss the result sets generated by stored procedures. In other words, there are no other words to add—at least not here! However, there are a number of issues that arise when calling Oracle stored procedures. There are a number of Knowledge Base articles on connecting to Oracle databases, and the following paragraphs are an update of one of the most helpful of those articles.

To begin, you really need to get the latest Oracle ODBC or OLE DB drivers. A lot of work was done in this area, so be sure to leverage these improvements and bug fixes by loading the most recent versions. Without these drivers, you'll find it impossible to retrieve Recordsets from Oracle stored procedures.

Oracle developers know that you can't just use Visual Basic or Visual Studio right out of the box with Oracle systems because you still need their proprietary drivers. These are installed on the client separately, and not being an Oracle expert, I won't try to explain that installation here.
With the release of the Microsoft ODBC Driver for Oracle version 2.0 and higher, you can now retrieve result sets from Oracle stored procedures. By creating Oracle stored procedures that return parameters of type TABLE, you can return row and column data that can then be manipulated and displayed as a result set. Knowledge Base article Q174679 uses the example in the Help file for the Microsoft ODBC Driver for Oracle v2.0 and shows how to use that example in Visual Basic.

**NOTE**  
The result sets created by the Microsoft ODBC Driver for Oracle version 2.0 and 2.5 using Oracle stored procedures are READ ONLY and STATIC. An Oracle Package must be created to retrieve a result set.

### Managing Command Objects

Remember that there are four phases to the creation and execution of Command objects:

1. Create the Command object and construct the Parameters collection.
2. Set the Parameter Value property for each input parameter.
3. Execute the query.
4. Process the results.

Phase 1 should be executed *once*. The remaining phases can be done on an “as required” basis, later in your application. If you review a client/server application that reruns the first phase repeatedly, revisit the source, and code the initialization process to a one-time-only event procedure. This is another best practice. Unfortunately, in the middle tier (and ASP) code, you have to construct the Command objects each time. But remember that Command objects are *not* required unless you're executing queries that return OUTPUT or Return Status values from the query.

### Executing and Reexecuting Command Objects

After the Command object is constructed, the expensive part of constructing the Command object on the client is completed. We discussed some of the ways to
execute queries in Chapter 4, so what follows won’t be all that new. The Command’s query can be executed in several ways:

- Using the Execute method on the constructed Command object.
- Using the Command object as a method on the Connection object.
- Referencing the Command with the Execute method on the Connection object.
- Referencing the Command with the Open method on the Recordset object. This technique is discussed in Chapter 6 where we focus on creating Recordsets.

Let’s take a look at these techniques one at a time. Some of the primary differences you want to watch out for are how the Recordset is constructed (do you get to build it ahead of time or do you have to take the standard default) and how the parameters are passed to the Command object (do you have to provide them parameter-by-parameter or can you pass them as method arguments one way or another).

**Using the Execute Method on a Command Object**

One of the simplest and least flexible techniques is to simply “execute” the Command. This approach tells ADO to take the Command as it’s currently populated and run it, and (optionally) construct a new Recordset object to receive the rowset. The Command object’s Execute method is basically the same as when used against a Connection object (as we discussed in Chapter 4).

The following code example illustrates the use of the Execute method. (I use this Command object for the rest of the Command execution examples.) The principal difference here is that when you use the Execute method, ADO constructs a virgin Recordset set to ReadOnly/ForwardOnly to handle the rowset. If you use the Command as Connection method technique, you get to construct your own Recordset beforehand. Although the default Recordset constructed by the Execute method retrieves rows efficiently, if you expect updatability, you won’t get it this way.

```vbnet
Set cmd = New Command
With cmd
    .Name = "GetTitles"
    .ActiveConnection = cn
```

*Chapter 5*
ADO Command Strategies

Handling Parameter Default Values

Note that in the preceding example, the second parameter is set up in the Parameters collection with a value (1940). This tells ADO to reset the parameter to this value after each use of the Command object. After a Command object is executed, ADO resets each Parameter.Value back to the initial setting. In the case of the first and third Parameters, the Value is set back to empty.

Now that the Command is set up, we’re ready to execute it. The following line might do the trick:

```vbscript
cmd.Execute
```

This code simply ignores the “rows affected” value—it won’t return anything for this query anyway—and accepts the existing parameter value settings. Oops, because they weren’t set, we get a trappable error. We should have set the parameter values first.

```vbscript
cmd(0) = "Hitch%"
cmd(1) = 1940
cmd(2) = 1990
cmd.Execute
```

Now the Execute method works fine. After the Command is executed, the Parameters are reset to their initial state—empty, 1940, and empty.

Passing Arguments with Variant Arrays

Another way to pass in input parameters is to use a Variant array. This approach is kinda cool. And better yet, if you don’t provide one of the elements, ADO does not submit the parameter—it assumes the provider will insert the default value. If there is no default value set, either in the called stored procedure or in the initial
Parameter collection, ADO returns a trappable error: "-2147217900 Incorrect syntax near the keyword 'DEFAULT'."

vParms = Array("Hitch", 1940, 1990)
cmd.Execute lRA, vParms

Let's try another variation:


This technique bypasses the creation of a separate Variant array and simply uses the new Visual Basic Array function. In this case, we let the existing setting of the second parameter (cmd(1)) be used, because it has a default value. This variation, however, prevents you from reading the OUTPUT parameters off the array.

If you're executing an action query, it's nice to know how many rows are affected. If the provider can tell you, the value is returned in the Long variable passed as the first argument, RecordsAffected. It does not return the number of records in your rowset. It's passed to our routines above in the lRA variable.

**Setting Additional Execution-time Options**

When we defined the Command object, we specified the CommandType property, so we shouldn't have to provide it again when it comes time to execute a Command, but you can. There are also other more interesting options that do make sense when using the Execute method, including options that tell ADO you aren't expecting rows, that it should execute asynchronously, and several others, as shown in Table 5-3:

<table>
<thead>
<tr>
<th>CONSTANT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>adAsyncExecute</td>
<td>Indicates that the command should execute asynchronously. This returns control to your application or component immediately (just after the first CacheSize rows are fetched), freeing up your thread for other tasks. It does not make a lot of sense to use this option for the middle tier, but it can really help &quot;perceived&quot; performance with client/server rigs.</td>
</tr>
<tr>
<td>adAsyncFetch</td>
<td>Tells ADO to asynchronously fetch the remaining rows after the initial quantity specified in the CacheSize are fetched. If this option is specified, ADO keeps fetching rows so that rowset population occurs more quickly. This is important for the overall</td>
</tr>
</tbody>
</table>
ADO Command Strategies

system, as well as for your application, as rows past the initial cache are available more quickly. I discuss asynchronous fetching in Chapter 6.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adAsyncFetchNonBlocking</td>
<td>Prevents the main thread from blocking while retrieving. If the requested row has not been retrieved, the current row automatically moves to the end of the result set (EOF).</td>
</tr>
<tr>
<td>adExecuteNoRecords</td>
<td>Indicates that the command text is a command or stored procedure that does not return rows (for example, a command that only inserts data). If any rows are retrieved, they are discarded and not returned. This option is always combined with CommandType options of adCmdText or adCmdStoredProc. This option prevents ADO from constructing a Recordset object in cases where it’s not needed.</td>
</tr>
</tbody>
</table>

By using the right set of options you help ADO do less work to run your query and fetch rows more efficiently. You might have to “add” several of these together, as shown in the following code:

cn.Execute("MyUpdateSP", , adCmdStoredProcedure + adExecuteNoRecords)

When the Command object completes (successfully or not) an ExecuteComplete event fires.

Executing a Command Object as a Connection Method

Because we are also discussing developer performance, it makes sense to mention the Command as Connection method technique. This technique uses a named Command object as a method against the ADO Connection associated with it. That is, once you name a Command object (by setting its Name property) and set the ActiveConnection to a Connection object, you can use the following syntax to execute it:

Private Sub cmdRunQuery_Click()
    cnMyConnection.GetAuthors txtYearWanted.Text, rs
    ' parameter, Recordset
    ProcessResults(rs)
End Sub
In this case, I’m executing a Command object called GetAuthors. The input parameters are passed as arguments to the method. The last argument passed is a reference to an instantiated Recordset object, which will contain the rowset when ADO completes executing the query.

**TIP** For reasons that have to do with how COM works behind the scenes, you must fully qualify all object-sourced parameters when using this technique. This means that you can’t just pass in txtYearWanted, which should refer to the default Text property of the TextBox control referenced. Instead, you must pass in txtYearWanted.Text. This prevents a number of strange behaviors, including confused parameters being passed to the data provider.

This Command as Connection method technique is also very easy to code and yields excellent performance. In this case, you work with a preconstructed Recordset object configured with the correct CursorType, LockType, and other properties that make sense. Unlike the Command object’s Execute method (discussed next), you get to describe how the Recordset object is to be constructed.

**WARNING** There is an outstanding bug posted against this technique. For more information, see “The Other Side of the Refresh Method Story” section, earlier in this chapter.

### Using the Command Object’s Execute Method

The Command object’s Execute method is far less flexible than other techniques. It also has a number of, well, interesting side effects that you might not be aware of. There are two flavors of the Execute method:

- Against a Connection object: In this case, you provide a Source argument, but you can’t specify a Command object here as you can with the Recordset object’s Open method (which I discuss in Chapter 6). You have to pass the SQL query as an SQL statement, a table name, a URL, or the name of a stored procedure. You also need to pass an appropriate CommandType in the Options argument—or let ADO guess the type of command to

---

12. All parameters are stored in a DISPARAMS structure of pure variants, but control references are valid variants, so Visual Basic just passes those—and ADO choke.
execute—but you should always provide the CommandType to prevent ADO from "guessing" what's in the query you're submitting.

- Against a Command object: In this case, the Source argument is contained in the Command object, as are all of the other arguments needed to run the query, including the CommandType. But, again, ADO constructs the Recordset for you, so you have to settle for the "default" Recordset properties—read-only, forward-only.

In either case, if you want to return a rowset, you must capture the Recordset passed back from the Execute method. The Execute method always constructs the Recordset to the default specifications (read-only, forward-only) for you. For example:

```vbscript
Set rs = cmd.Execute
```

If you don't need a Recordset, use the alternative syntax:

```vbscript
cmd.Execute
```

However, there is an interesting side effect. In some cases, and especially when you create a Recordset using the Execute method against a Connection object, ADO opens an additional connection to run the query, and closes it afterwards. This is especially true if you use the Execute method again before fully populating the original Recordset.

Actually, the sequence of operations triggers this behavior. For example, if you set cmd.ActiveConnection to cn, and use the cmd.Execute method followed by cn.Execute, ADO does not open a new connection. However, if you use the cn.Execute method followed by the cmd.Execute method, ADO does create another connection to run the Command.

**Tracing the Server and Data Provider**

When you use ADO to execute your query, the server dutifully executes what ADO and the data provider sends it. However, how efficiently these statements are executed and how they affect scalability is an important consideration. The CursorLocation (client-side or server-side cursors), CursorType, LockType, and even the syntax you use can affect what type of query is sent to the backend. Because only SQL Server exposes enough trace information for you to be able to see what's actually getting transmitted to the server, I'm going to walk you through a number of configurations so that you can choose the query style that makes the most sense for your application or component. At least you'll understand which factors affect the resulting commands sent to SQL Server.
I’m going to step through these configurations one group at a time. First, server-side cursors with the default Recordset settings of read-only, forward-only are listed in Table 5-4. Table 5-5 lists server-side cursors with keyset/optimistic cursors. Notice the considerable difference in the number of round-trips and the complexity of the queries that the server has to process to execute basically the same Command but using different syntax to support the selected cursor types.

<table>
<thead>
<tr>
<th>COMMAND SYNTAX</th>
<th>GENERATED QUERY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cn.GetTitlesByYear</td>
<td>GetTitlesByYear 'Hitch%', 1900, 1999</td>
</tr>
<tr>
<td>rs.Open cmd</td>
<td>sp_executesql N'Select Top 50 title, ISBN, Year_Published from titles where title like @P1 and year_published between @P2 and @P3', @P1 varchar(20), @P2 int, @P3 int, 'Hitch%', 1900, 1999</td>
</tr>
<tr>
<td>rs.Open strSQL</td>
<td>Select Top 50 title, ISBN, Year_Published from titles where title like 'Hitch%' and year_published between 1900 and 1999</td>
</tr>
<tr>
<td>Cn.GetTitles</td>
<td>sp_executesql N'Select Top 50 title, ISBN, ...</td>
</tr>
<tr>
<td>Set rs = cmd.Execute...</td>
<td>sp_executesql N'Select Top 50 title, ISBN, ...</td>
</tr>
</tbody>
</table>

Table 5-4: Server-side Read-only/Forward-only Cursors

<table>
<thead>
<tr>
<th>COMMAND SYNTAX</th>
<th>GENERATED QUERY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cn.GetTitlesByYear</td>
<td>declare @P1 int declare @P3 int declare @P4 int declare @P5 int set @P1=NULL set @P3=102401 set @P4=311300 set @P5=NULL exec sp_cursoropen @P1 output, N'Select Top 50 title, ISBN, Year_Published from titles where title like @P1 and year_published between @P2 and @P3', @P1 varchar(20), @P2 int, @P3 int, 'Hitch%', 1900, 1999</td>
</tr>
<tr>
<td></td>
<td>select @P1, @P3, @P4, @P5</td>
</tr>
</tbody>
</table>

13. Using the Command.Execute method only makes sense with forward-only/read-only, because ADO manufactures the Recordset on each invocation.
Client-side cursors are far less flexible than server-side cursors. In this case, only the Static CursorType is supported, and updatability is only provided if you choose adLockBatchOptimistic as the LockType. No, I don’t expect you to study these tables that closely. I do expect you to try these tests on your own. The source code to set up all of these tests is on the book’s CD. Dig into the ..\sample application\command objects\ directory for the code. Table 5-6 lists client-side static/optimistic cursors, and Table 5-7 lists client-side static/batch optimistic cursors.

### Table 5-6: Client-side Static/Optimistic Cursors

<table>
<thead>
<tr>
<th>COMMAND SYNTAX</th>
<th>GENERATED QUERY</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs.Open cmd</td>
<td>sp_executesql...</td>
</tr>
<tr>
<td>cmd.Execute</td>
<td>sp_executesql...</td>
</tr>
<tr>
<td>Cn.GetTitlesByYear</td>
<td>GetTitlesByYear 'Hitch%', 1900, 1999</td>
</tr>
<tr>
<td>Cn.GetTitles</td>
<td>sp_executesql...</td>
</tr>
<tr>
<td>rs.Open strSQL</td>
<td>Select Top 50 title, ISBN, Year_Published from ...</td>
</tr>
</tbody>
</table>

### Table 5-7: Client-side Static/Batch Optimistic Cursors

<table>
<thead>
<tr>
<th>COMMAND SYNTAX</th>
<th>GENERATED QUERY</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs.Open cmd</td>
<td>sp_executesql...</td>
</tr>
<tr>
<td>cmd.Execute</td>
<td>sp_executesql...</td>
</tr>
<tr>
<td>Cn.GetTitlesByYear</td>
<td>GetTitlesByYear 'Hitch%', 1900, 1999</td>
</tr>
<tr>
<td>Cn.GetTitles</td>
<td>sp_executesql...</td>
</tr>
<tr>
<td>rs.Open strSQL</td>
<td>Select Top 50 title, ISBN, Year_Published from ...</td>
</tr>
</tbody>
</table>

As you can see, there are a variety of approaches that the SQL Server data provider takes when executing queries on ADO’s behalf. However, ADO seems to be fairly predictable, regardless of the cursor location setting. If you use client-side cursors and execute an ad hoc query using the Recordset Open method, or execute a

---

14. There seems to be some controversy about this technique. The ADO people are still working on it.
stored procedure, ADO simply takes the SQL and “passes it through,” not attempting to leverage any of the system-level sp_cursor functions or even use the sp_executesql function. This is not necessarily wrong. SQL Server knows how to persist query plans for ad hoc queries, and the query plans for SQL Server are already compiled.

Calling stored procedures seems to be far more efficient than using ad hoc queries or Command objects. When you stick with forward-only/read-only concurrency, ADO just passes the stored procedure invocation on through to the server, even managing the parameters. If you ask for an updatable cursor, ADO prefixes the stored procedure call with a sp_cursoropen to handle scrolling and updating.

Detecting Command State

The Command object’s State property is only really interesting when you execute a query asynchronously. That is, if you start executing a Command using any one of the techniques discussed in this chapter, the State property can indicate whether or not the query has completed. As long as ADO is busy managing the execution of your Command object, the Executing (adStateExecuting) bit of the State property (4) is on and the Open (adStateOpen) bit is off. Yes, you can poll the adStateExecuting bit as a way of waiting until an asynchronous operation is done, but I would rather use the ExecuteComplete event to do the job. Using the event consumes far less CPU resources when compared to polling—especially when compared to loop polling. It’s like having your daughter sitting in the back seat on the way to grandma’s house asking (every two minutes), “Are we there yet, daddy?”

The Cancel Method

So, you started the execution of your Command object with the adAsyncExecute option, and this returned control of your thread so that you could entertain the user with a progress bar or launch “Age of Empires II” while the user waits. If your user decides to quit waiting, you can use the Cancel method to stop processing the query—or at least you can try. Sometimes it’s like trying to stop a train 40 feet from the rail crossing.

NOTE The Cancel method can also be applied to the Connection object to cancel an asynchronous open operation, but that’s not as likely to be necessary—most connections complete before the user can blink, and if they are delayed longer than that, it’s usually a network delay that can’t be interrupted anyway.
You get a trappable error if the operation is not stoppable, such as if you try to cancel an operation that wasn’t started asynchronously. Consider that it might not be safe to cancel an operation midstream—especially operations making changes to the database. SQL Server and other providers won’t (necessarily) back out changes already made unless there was a transaction started beforehand.

**ADO Command Object’s “Fallback” Plan**

The ADO named command fallback plan seems to be stored procedures. That is, if you don’t create a Command object and you reference a named object off of the Connection object (like this):

```plaintext
cnOLEDB.TestQuery txtParm, rs
```

ADO will assume that TestQuery is a stored procedure. If it’s not, you’ll trip an error complaining that the stored procedure TestQuery could not be located. The same will happen if you use adCmdText as the Command.CommandType property and submit a single object name. This means you can execute any stored procedure (parameters and all) using the following syntax (assuming GetTitles is a stored procedure):

```plaintext
Cn.GetTitles "1980", rsMyRecordset
```

The problem with the stored procedure as Connection method technique is that it cannot handle OUTPUT parameters. This means that if the stored procedure has OUTPUT parameters, this technique will not return them to you—no Command object is created to handle them. While you can execute the stored procedure, you have to pass placeholders for each of the OUTPUT parameters because ADO defaults OUTPUT Parameter.Direction to INPUT/OUTPUT (adParamInputOutput). But when the query is executed, there is no Command object created to fetch the returning parameters.

**Command Object Tips and Warnings**

As I was working with the Command object to develop the example and test code, I came across several miscellaneous tips. A summary of those tips and techniques
appears below. I've also included a few suggestions that I picked up from my students and the folks I communicate with via e-mail:

- If you try to execute a stored procedure, and the parameters you describe don't match what SQL Server expects, almost anything can happen, including the error: "–2147418113 Unexpected failure."

- If you set the CommandType property to adCmdStoredProc and you reference a table instead of a stored procedure, you'll trip this trappable error: "–2147217900 — The request for procedure 'Publishers' failed because 'Publishers' is a table object." It's clear that ADO is watching out for you. The Profiler showed no signs of running round-trips to the server to figure this out, even though one actually takes place. However, this information might have been cached.

- If the argument passed to the Command object's parameter does not meet the datatype requirements (string, number, integer, date, currency, and so forth) your code will trigger a 3421 error: "The application is using a value of the wrong type for the current operation." Who writes these error messages anyway?

- If the CommandType is left at adCmdUnknown and the query contains a SELECT statement, ADO figures it out on its own and resets the CommandType to adCmdText.

- If the CommandType is adCmdTable, anything other than a table name in the CommandText trips the following syntax error when you attempt to execute the query: "–2147217900 Incorrect syntax near the keyword 'select'." However, the ODBC provider does not trap incorrect table names. I set up a Command object using adCmdTable with Fred as the CommandText, and when ADO submitted SELECT * FROM FRED, ADO did not complain. On the other hand, the OLE DB provider returned: "–2147217865 Invalid object name 'fred'.” Interesting.

NOTE  Error messages returned from the OLE DB provider arrive without any prefix indicating which layer caused the error. The ODBC messages come back looking like this:

[Microsoft][ODBC SQL Server Driver][SQL Server]Invalid object name 'fred'.

Chapter 5

114
Handling Connection Object Events

This is easy because there aren’t any events exposed on the Command object. You have to depend on the Connection or Recordset object to handle events associated with the Command. If you run a Command object asynchronously, you can trap the WillExecute or ExecuteComplete event on the associated Connection to determine when it’s done, or poll the Command object’s State property.

Performance Considerations: Using Command Objects Wisely

A common performance problem with Command objects is instantiating (creating), executing, and destroying them in a single Command_Click event, a middle-tier component, or on a Web page. This is a problem, because in these cases, the object setup expense is paid each time the Command is executed instead of once. Because you can’t expect to scale an application or component that leaves Command objects lying around waiting for potential clients, we have to construct them “just in time.” While there are certainly cases in which you must construct a Command object, you don’t want to expend the resources to instantiate and prepare the object unless it is necessary. Of course, in the middle tier (Microsoft Transaction Server or ASP), you may have no choice. In this case, you really want to think out your query strategy to determine whether it’s really necessary to create/use/destroy a Command object each time the method or ASP page is referenced—especially when other cheaper strategies are available. While Command objects make it easier to manage procedure parameters, the extra overhead of creating them might not pay off in better performance.

I used to think that the fundamental purpose of the Command object was to help ADO create a temporary stored procedure to run your query more quickly. However, I have seen ADO change its approach to the creation and execution of temporary stored procedures to leverage SQL Server and other providers’ capabilities to deal more efficiently with repeated ad hoc query invocations.

The Command object’s Execute method also constructs the Recordset object for you. That is, you can’t specify the LockType, CursorType, CacheSize, MaxRecords, or other interesting (but performance robbing) Recordset properties ahead of time—you have to take the defaults. The default Recordset is a read-only, forward-only, CacheSize=1 “firehose” (or cursorless) result set that is very efficient, so this behavior works for a high-performance application. However, if you code the Options argument to the Execute, you can tell ADO that the query does not return rows at all (as shown next), which helps improve performance.

Set rs = cmd.Execute(Options:=adExecuteNoRecords)
Chapter 5

Any information that you can give ADO to help it know how to process the “stuff” in the CommandText property helps performance. As with the Recordset object’s Open method options, which I discuss in Chapter 6, you really must tell ADO whether or not this is the name of a stored procedure, a table, the name of a file, or just random SQL. No, adCmdTableDirect is not a valid choice for most providers. The CommandType default is adCmdUnknown, which means ADO will have to guess, which takes time.

In many cases, you can improve performance by avoiding client-to-server round-trips. For example, if you combine a number of related operations together in a single query, you can save quite a bit of system and backend overhead. However, not all providers support multiple operations—SQL Server does, but Jet/Access does not. I use this technique to perform bulk inserts or to execute a series of commands bound in a transaction. You have to understand how to deal with the more complex result sets that ADO generates using this technique. But that’s not hard.