20

Data Access with ADO

ActiveX Data Objects, or ADO for short, is Microsoft's technology of choice for performing client-server data access between any data consumer (the client) and any data source (the server). There are other data-access technologies you may have heard of in relation to Excel, including DAO and ODBC. However, these are not covered in this chapter because Microsoft intends for ADO to supercede these older technologies, and for the most part this has occurred.

ADO is a vast topic, easily the subject of its own book. This chapter necessarily presents only a small subset of ADO, covering the topics and situations that I've run across most frequently in my career as an Excel programmer. This chapter focuses on ADO 2.5. This version of ADO ships natively with Windows 2000 or Office 2000 and higher, so you can assume it will be present on any computer you distribute your application to.

An Introduction to Structured Query Language (SQL)

It's impossible to get very far into a discussion of data access without running into SQL, the querying language used to communicate with all databases commonly in use today. SQL is a standards-based language that has as many variations as there are databases. This chapter uses constructs compliant with the latest SQL standard, SQL-92, wherever possible.

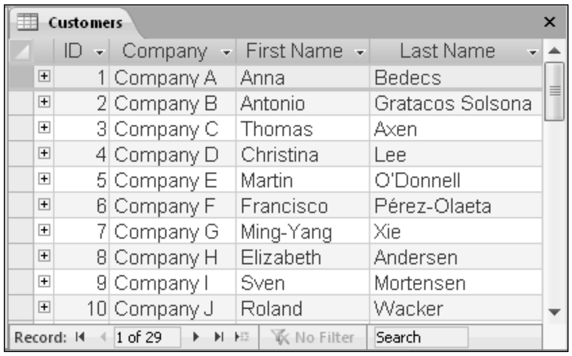
There are four fundamental operations supported by SQL:

- `SELECT` — Used to retrieve data from a data source
- `INSERT` — Used to add new records to a data source
- `UPDATE` — Used to modify existing records in a data source
- `DELETE` — Used to remove records from a data source

The terms *record* and *field* are commonly used when describing data. The data sources you'll be concerned with in this chapter can all be thought of as being stored in a two-dimensional grid. A record represents a single row in that grid. A field represents a column in the grid. The intersection of a record and a field is a specific value. A *resultset* is the term used to describe the set of data returned by a SQL SELECT statement.

You will notice that SQL keywords such as SELECT and UPDATE are shown in uppercase. This is considered good SQL programming practice. When viewing complex SQL statements, having SQL keywords in uppercase makes it significantly easier to distinguish between those keywords and their operands. The subsections of a SQL statement are called clauses. In all SQL statements, some clauses are required and others are optional. When describing the syntax of SQL statements, optional clauses and keywords will be surrounded by square brackets.

Use the Customers table from Microsoft's Northwind sample database, as shown in Figure 20-1, to illustrate the SQL syntax examples. Northwind must be installed with Access 2007 in order to follow many of the examples in this chapter.



ID	Company	First Name	Last Name
1	Company A	Anna	Bedecs
2	Company B	Antonio	Gratacos Solsona
3	Company C	Thomas	Axen
4	Company D	Christina	Lee
5	Company E	Martin	O'Donnell
6	Company F	Francisco	Pérez-Olaeta
7	Company G	Ming-Yang	Xie
8	Company H	Elizabeth	Andersen
9	Company I	Sven	Mortensen
10	Company J	Roland	Wacker

Figure 20-1

The SELECT Statement

The SELECT statement is by far the most commonly used statement in SQL. This is the statement that allows you to retrieve data from a data source. The following clauses of the SELECT statement are used in this chapter. Only the SELECT and FROM clauses are required to constitute a valid SQL statement:

```
SELECT [DISTINCT] column1, column2, ...
FROM table_name
[WHERE restriction_condition]
[ORDER BY column_name [ASC|DESC]]
```

The SELECT clause tells the data source what fields you wish to return. The field names in the SELECT clause are called the SELECT list. The FROM clause tells the data source which table the records should be retrieved from. For instance, a simple example statement could look like this:

```
SELECT Company, [First Name], [Last Name]
FROM Customers
```

This statement will notify the data source that you want to retrieve all of the values for the Company, First Name, and Last Name fields from the Customers table.

Note that the First Name and Last Name fields in the SQL statement are surrounded by square brackets. This is required for any field or table name that contains spaces or non-alphanumeric characters.

The SELECT statement also provides a shorthand method for indicating that you want to retrieve all fields from the specified table. This involves using a single asterisk as the SELECT list:

```
SELECT *
FROM Customers
```

This SQL statement will return all fields and all records from the Customers table. It's generally not considered a good practice to use * in the SELECT list, because it leaves your code vulnerable to changes in field names or the order of fields in the table. It can also be very resource intensive with large tables, because all columns and rows will be returned whether or not they are actually needed by the client. However, there are times when it is a useful and time-saving shortcut.

Say that you want to see a list of countries where you have at least one customer located. Simply performing the following query would return one record for every customer in your table:

```
SELECT [Country/Region]
FROM Customers
```

This resultset would contain many duplicate country names. The optional DISTINCT keyword allows you to return only unique values in your query:

```
SELECT DISTINCT [Country/Region]
FROM Customers
```

If you only want to see the list of customers located in the U.S., you can use the WHERE clause to restrict the results to only those customers:

```
SELECT Company, [First Name], [Last Name]
FROM Customers
WHERE [Country/Region] = 'USA'
```

Note that the string literal USA must be surrounded by single quotes. This is also true of dates. Numeric expressions do not require any surrounding characters.

Finally, suppose you would like to have your USA customer list sorted by Company. This can be accomplished using the ORDER BY clause:

```
SELECT Company, [First Name], [Last Name]
FROM Customers
WHERE Country = 'USA'
ORDER BY Company
```

The ORDER BY clause will order fields in ascending order by default. If instead you wanted to sort a field in descending order, you could use the optional DESC specifier immediately after the name of the column whose sort order you wanted to modify.

The INSERT Statement

The INSERT statement allows you to add new records to a table. The basic syntax of the INSERT statement is the following:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...)
```

Use of the INSERT statement is very simple. You provide the name of the table and its columns that you'll be inserting data into, and then provide a list of values to be inserted. You must provide a value in the VALUES clause for each column named in the INSERT clause, and the values must appear in the same order as the column names they correspond to. Here's an example showing how to insert a new record into the Customers table:

```
INSERT INTO Customers (Company, [First Name], [Last Name], [Country/Region])
VALUES ('New Company', 'Rob', 'Bovey', 'USA')
```

Note that as with the WHERE clause of the SELECT statement, all of the string literals in the VALUES clause are surrounded by single quotes. This is the rule throughout SQL.

If you have provided values for every field in the table in your VALUES clause, the field list in the INSERT clause can be omitted. For example, if the four preceding fields were the only fields in the Customers table, you could simply use:

```
INSERT INTO Customers
VALUES ('New Company', 'Rob', 'Bovey', 'USA')
```

The UPDATE Statement

The UPDATE statement allows you to modify the values in one or more fields of an existing record or records in a table. The basic syntax of the UPDATE statement is the following:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
[WHERE restriction_condition]
```

Even though the WHERE clause of the UPDATE statement is optional, you must take care to specify it unless you are sure that you don't need it. Executing an UPDATE statement *without* a WHERE clause will modify the specified field(s) of every record in the specified table. Say, for example, you executed the following statement:

```
UPDATE Customers
SET [Country/Region] = 'USA'
```

Every record in the `Customers` table would have its `Country` field modified to contain the value `USA`. There are some cases where this mass update capability is useful, but it can also be very dangerous, because there is no way to undo the update if you execute it by mistake.

The more common use of the `UPDATE` statement is to modify the value of a specific record, identified by the use of the `WHERE` clause. Before examining an example of this usage, you need to understand a very important aspect of database design called the *primary key*. The primary key is a field or group of fields in a database table whose values can be used to uniquely identify each record in that table. There is no way to identify a specific record in a table that does not have a primary key. Without that capability, you cannot perform an update on a specific record.

The primary key in this sample `Customers` table is the `ID` field. Each customer record in the `Customers` table has a unique value for `ID`. In other words, a specific `ID` value occurs in one, and only one, customer record in the table.

Say that the `First Name` and `Last Name` fields have changed for the customer "Company A", whose `ID` is 1. You could perform an `UPDATE` to record those changes in the following manner:

```
UPDATE Customers
SET [First Name] = 'First', [Last Name] = 'Last'
WHERE ID = 1
```

Because you used the primary key field to specify a single record in the `Customers` table, only this record will be updated.

The DELETE Statement

The `DELETE` statement allows you to remove one or more records from a table. The basic syntax of the `DELETE` statement is the following:

```
DELETE FROM table_name
[WHERE restriction_condition]
```

As with the `UPDATE` statement, notice that the `WHERE` clause is optional. This is probably more dangerous in the case of the `DELETE` statement, however, because executing a `DELETE` statement without a `WHERE` clause *will delete every single record in the specified table.* Once again, there is no way to undo this, so be very careful. You should always include a `WHERE` clause in your `DELETE` statements unless you have some very specific reason for wanting to remove all records from a table.

Assume that, for some reason, an entry was made into the `Customers` table with the `ID` value of 30 by mistake (maybe they were a supplier rather than a customer). To remove this record from the `Customers` table, you would use the following `DELETE` statement:

```
DELETE FROM Customers
WHERE ID = 30
```

Once again, because you used the record's primary key in the `WHERE` clause, only that specific record will be affected by the `DELETE` statement.

An Overview of ADO

ADO is Microsoft's universal data-access technology. *Universal* means that ADO is designed to allow access to any kind of data source imaginable, from a SQL Server database to the Windows Active Directory to a text file saved on your local hard disk, and even to non-Microsoft products such as Oracle. All these things and many more can be accessed by ADO.

ADO doesn't actually access a data source directly. Instead, ADO is a data consumer that receives its data from a lower-level technology called *OLE DB*. OLE DB cannot be accessed directly using VBA, so ADO was designed to provide an interface that allows you to do so. ADO receives data from OLE DB *providers*. Most OLE DB providers are specific to a single type of data source. Each is designed to provide a common interface to whatever data its source may contain. One of the greatest strengths of ADO is that, regardless of the data source you are accessing, you use essentially the same set of commands. There's no need to learn different technologies or methods to access different data sources.

Microsoft also provides an OLE DB provider for ODBC. This general-purpose provider allows ADO to access any data source that understands ODBC, even if a specific OLE DB data provider is not available for that data source. Figure 20-2 shows the communication path between ADO and a data source.

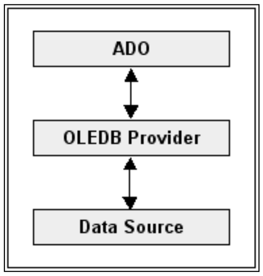


Figure 20-2

Unlike the deep, complex object models of the data access technologies that preceded it, the ADO object model is very flat and simple to understand. It achieves this simplicity without losing any of its power to access and manipulate data.

ADO consists of five top-level objects, all of which can be created independently. This chapter covers the `Connection` object, the `Command` object, and the `Recordset` object. ADO also exposes a `Record` object (not to be confused with the `Recordset` object), as well as a `Stream` object. These objects are not commonly used in Excel applications, so they are not covered in this chapter.

In addition to the five top-level objects, ADO contains four collections. That's it. Five objects and four collections are all you need to master to gain the power of ADO at your fingertips. Figure 20-3 shows the ADO object model.

The next three sections provide an introduction to each of the top-level ADO objects that you'll use in this chapter. These sections provide general information that will be applicable whenever you are using ADO. Specific examples of how to use ADO to accomplish a number of the most common data access tasks you'll encounter in Excel VBA are covered in the sections that follow.

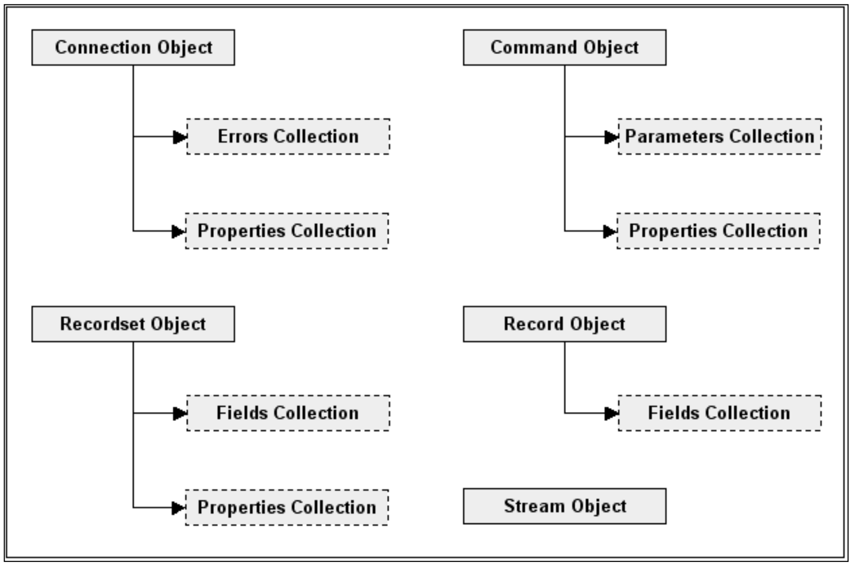


Figure 20-3

This is not intended to be an exhaustive reference to ADO. I will only be covering those items whose use will be demonstrated in this chapter, or those I consider particularly important to point out. ADO frequently gives you the flexibility to make the same setting in multiple ways, as both an object property and an argument to a method, for instance. In these cases, I will usually only cover the method I intend to demonstrate in the example sections.

The Connection Object

The `Connection` object is what provides the pipeline between your application and the data source you want to access. Like the other top-level ADO objects, the `Connection` object is extremely flexible. In some cases, this may be the only object you need to use. Simple commands can easily be executed directly through a `Connection` object. In other cases, you may not need to create a `Connection` object at all. The `Command` and `Recordset` objects can create a `Connection` object automatically if they need one.

Constructing and tearing down a data source connection can be a time-consuming process. If you will be executing multiple SQL statements over the course of your application, you should create a publicly scoped `Connection` object variable and use it for each query. This allows you to take advantage of *connection pooling*.

Connection pooling is a feature provided by ADO that will preserve and reuse connections to the data source rather than creating new connections for each query, which would be a waste of resources. Connections can be reused for different queries as long as their connection strings are identical. This is typically the case in Excel applications, so I recommend taking advantage of it.

Connection Object Properties

This section examines the important `Connection` object properties.

The ConnectionString Property

This property is used to provide ADO, and the OLE DB provider you are using, with the information required to connect to the data source. The connection string consists of a semicolon-delimited series of arguments in the form of `"name=value;"` pairs.

For the purposes of this chapter, the only ADO argument used is the `Provider` argument. The `Provider` argument tells ADO which OLE DB provider to use. All other arguments in connection strings presented in this chapter will be specific to the OLE DB provider being used. ADO will pass these arguments directly through to the provider. The following sample code demonstrates how to create a connection string to the Northwind database using the Access 2007 OLE DB provider:

```
objConn.ConnectionString = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
                           "Data Source=C:\Files\Northwind 2007.accdb"
```

The only argument specific to ADO in the connection string is the `Provider` argument. All other arguments are passed directly through to the specified OLE DB provider. If a different provider were being used, the arguments would be different as well. You will see this when you begin to connect to various data sources in the example sections. The `Provider` argument to the connection string is optional. If no provider is specified, ADO uses the OLE DB provider for ODBC by default.

The ConnectionTimeout Property

This property specifies how many seconds ADO will wait for a connection to complete before canceling the attempt and raising an error. The default value is 15 seconds. If you have a situation where connections normally take a long time to complete, you can increase this number so ADO doesn't terminate the connection attempt prematurely. The following code sample changes the timeout value on the connection to 30 seconds:

```
objConn.ConnectionTimeout = 30
```

The State Property

The `State` property allows you to determine whether a connection is open, closed, connecting, or executing a command. The value will be a bit mask containing one or more of the following `ObjectStateEnum` constants:

- `adStateClosed` — The connection is closed
- `adStateOpen` — The connection is open
- `adStateConnecting` — The object is in the process of making a connection
- `adStateExecuting` — The connection is executing a command

If you attempt to close a `Connection` object that is already closed, you will cause an error. You can prevent this from occurring by testing the state of the `Connection` object before closing it:

```
If CBool(objConn.State And adStateOpen) Then objConn.Close
```

Connection Object Methods

This section examines the `Connection` object's more important methods, all of which have self-explanatory names.

The Open Method

This method opens a connection to the data source, and has the following syntax:

```
connection.Open ConnectionString, UserID, Password, Options
```

The `ConnectionString` argument serves the same purpose as the `ConnectionString` property discussed in the previous section. ADO allows you to set this property in advance or pass it in at the time you open the connection. The `UserID` and `Password` arguments can be passed separately from the connection string if you wish.

The `Options` argument is particularly interesting. This argument allows you to make your connection *asynchronously*. That is, you can tell your `Connection` object to go off and open the connection in the background while your code continues to run. You do this by setting the `Options` argument to the `ConnectOptionEnum` value `adAsyncConnect`. The following code sample demonstrates making an asynchronous connection:

```
objConn.Open Options:=adAsyncConnect
```

This is especially useful in situations where you have lengthy connection times, because it allows you to connect without freezing your application during the connection process.

The Execute Method

This method executes the command text provided to its `CommandText` argument. The `Execute` method has the following syntax for an action query (one that does not return a resultset):

```
connection.Execute CommandText, [RecordsAffected], [Options]
```

And for a select query:

```
Set Recordset = connection.Execute(CommandText, _
                                   [RecordsAffected], [Options])
```

The `CommandText` argument can contain any executable string recognized by the OLE DB provider. However, it will most commonly contain a SQL statement. The optional `RecordsAffected` argument is a return value that tells you how many records the `CommandText` operation has affected. It's a good idea to check this value against the number of records that you expected to be affected, so you can detect potential errors in your command text.

The `Options` argument is crucial to optimizing the execution efficiency of your command. Therefore, you should always use it even though it's nominally optional. The `Options` argument allows you to relay two different types of information to your OLE DB provider: what type of command is contained in the `CommandText` argument, and how the provider should execute the contents of the `CommandText` argument.

To execute the `CommandText`, the OLE DB provider must know what type of command it contains. If you don't specify the type, the provider will have to determine that information for itself. This will slow down the execution of your query. You can avoid this by specifying the `CommandText` type using one of the following `CommandTypeEnum` values:

- `adCmdText` — The `CommandText` is a raw SQL string.
- `adCmdTable` — The `CommandText` is the name of a table. This sends an internally generated SQL statement to the provider that looks something like `"SELECT * FROM table_name"`.
- `adCmdStoredProc` — The `CommandText` is the name of a stored procedure (stored procedures are covered in the section "Using ADO with Microsoft SQL Server").
- `adCmdTableDirect` — The `CommandText` is the name of a table. However, unlike `adCmdTable`, this option does not generate a SQL statement and therefore returns the contents of the table more efficiently. Use this option if your provider supports it.

You can provide specific execution instructions to the provider by including one or more of the `ExecuteOptionEnum` constants:

- `adAsyncExecute` — Tells the provider to execute the command asynchronously, which returns execution to your code immediately.
- `adExecuteNoRecords` — Tells the provider not to construct a `Recordset` object. ADO will always construct a recordset in response to a command, even if your `CommandText` argument is not a row-returning query. To avoid the overhead required to create an unnecessary recordset, use this value in the `Options` argument whenever you execute a non-row-returning query.

The `CommandTypeEnum` and `ExecuteOptionEnum` values are bit masks that can be combined together in the `Options` argument using the logical `Or` operator. For example, to execute a plain text SQL command and tell ADO not to construct a `Recordset` object, you would use the following syntax:

```
szSQL = "DELETE FROM Customers WHERE CustomerID = 'XXXX'"
objConn.Execute szSQL, lNumAffected, adCmdText Or adExecuteNoRecords
If lNumAffected <> 1 Then MsgBox "Error executing SQL statement."
```

The Close Method

This method closes the connection to the data source. Simply closing the connection does not destroy the `Connection` object. To destroy the `Connection` object and free its memory, you need to set the `Connection` object variable to `Nothing`. For example, to ensure that a `Connection` object variable is closed and removed from memory, you would execute the following code:

```
If CBool(objConn.State And adStateOpen) Then objConn.Close
Set objConn = Nothing
```

Connection Object Events

`Connection` object events must be trapped by creating a `WithEvents Connection` object variable in a class module. Trapping these events is necessary whenever you are using a `Connection` object asynchronously, because these events are what notify your application that the `Connection` object has completed its task.

Covering asynchronous connections is beyond the scope of this chapter. However, they are important enough to deserve mention so you can pursue them further if you like. The two most commonly used `Connection` events are:

- `ConnectComplete` — Triggered when an asynchronous connection has been completed. You can examine the arguments passed to this event to determine if the connection was successful or not.
- `ExecuteComplete` — Triggered when an asynchronous command has finished executing.

Connection Object Collections

The `Connection` object has two collections, `Errors` and `Properties`.

Errors Collection

This collection contains a set of `Error` objects, each of which represents an OLE DB provider-specific error (ADO itself generates run-time errors). The `Errors` collection can contain not only errors, but also warnings and even messages (generated by the T-SQL `PRINT` statement, for instance). The `Errors` collection is very helpful in providing extra detail when something in your ADO code has malfunctioned. When debugging ADO problems, you can dump the contents of the `Errors` collection to the Immediate window with the following code:

```
For Each objError In objConn.Errors
    Debug.Print objError.Description
Next objError
```

The Properties Collection

This collection contains provider-specific, or *extended properties*, for the `Connection` object. Some providers add important settings that you will want to be aware of. Extended properties are beyond the scope of this chapter.

The Recordset Object

Just as the most commonly used SQL statement is the `SELECT` statement, the most commonly used ADO object is the `Recordset` object. The `Recordset` object serves as a container for the records and fields returned from a `SELECT` statement executed against a data source.

Recordset Object Properties

The examination of the `Recordset` object begins with a look at its important properties.

The ActiveConnection Property

Prior to opening the `Recordset` object, you can use the `ActiveConnection` property to assign an existing `Connection` object to the `Recordset` or a connection string for the recordset to use to connect to the database. If you assign a connection string, the recordset will create a `Connection` object for itself. Once the recordset has been opened, this property returns an object reference to the `Connection` object being used by the recordset.

The following code assigns a `Connection` object to the `ActiveConnection` property:

```
Set rsData.ActiveConnection = objConn
```

The following code assigns a connection string to the `ActiveConnection` property:

```
rsData.ActiveConnection = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
                          "Data Source=C:\Files\Northwind 2007.accdb"
```

The BOF and EOF Properties

These properties indicate whether the record pointer of the `Recordset` object is positioned before the first record in the recordset (`BOF`, or beginning of file) or after the last record in the recordset (`EOF`, or end of file). If the recordset is empty, both `BOF` and `EOF` will be `True`. The following code example demonstrates using these properties to determine if there is data in a recordset:

```
If Not rsData.EOF Then
    ' The Recordset contains data.
Else
    ' The Recordset is empty.
End If
```

Note that there is a difference between an empty recordset and a closed recordset. If you execute a query that returns no data, ADO will present you with a perfectly valid open recordset, but one that contains no data. Therefore, you should always verify that a recordset contains data using the previous method prior to any attempt to access that data. Attempting to access data from an empty recordset will cause a run-time error.

The CursorLocation Property

This property allows you to specify whether the server-side cursor engine or the client-side cursor engine manages the records in the recordset. A cursor is the underlying object that manages the data in the recordset. Certain operations require the cursor engine to be on one side or the other. This is explored in more detail in the examples section.

This property must be set before the recordset is opened. If you do not specify a cursor location, the default is server-side. You can set this property to one of the two `CursorLocationEnum` values, either `adUseClient` or `adUseServer`. The following code sample demonstrates setting the cursor location to client-side:

```
rsData.CursorLocation = adUseClient
```

The Filter Property

This property allows you to filter an open recordset so that only records that meet the specified condition are visible. The records that cannot be seen are not deleted, removed, or changed in any way, but are simply hidden from normal recordset operations. This property can be set to a string that specifies the filter you want to place on the records, or to one of the `FilterGroupEnum` constants.

You can set multiple filters, and the records exposed by the filtered recordset will be only those records that meet all of the conditions. To remove the filter from the recordset, set the `Filter` property to an empty string or the adFilterNone constant. The following code sample demonstrates filtering a recordset so that it displays only company names that begin with the letter B. Note that the use of wildcards is supported in a filter string:

```
    rsData.Filter = "Company LIKE 'B*'"
```

You can use the logical AND, OR, and NOT operators to set additional Filter property values:

```
    rsData.Filter = "Company LIKE 'B*' AND [Country/Region] = 'USA'"
```

The State Property

This is the same as the State property discussed in the section on the Connection object.

Recordset Object Methods

This section examines only five of the Recordset object's methods, because they are the ones that you are most likely to use.

The Open Method

This method opens the Recordset object and retrieves the data specified by the Source argument. The Open method has the following syntax:

```
    recordset.Open Source, ActiveConnection, CursorType, LockType, Options
```

The Source argument tells the recordset what data it should retrieve. This is most commonly a SQL string or the name of a stored procedure, but can also be the name of a table or a Command object.

The ActiveConnection argument can be a connection string or a Connection object that identifies the connection to be used. If you assign a connection string to the ActiveConnection argument, the recordset will create a Connection object for itself.

The CursorType argument specifies the type of cursor to use when opening the recordset. This is set using one of the CursorTypeEnum values. This chapter uses only the adOpenForwardOnly and adOpenStatic cursor types. The first type will be used for normal queries, and using it means that the recordset can be navigated in only one direction, from beginning to end, which is the fastest method for accessing data. Note that the data in a forward-only recordset cannot be modified. The second type will be used for disconnected recordsets and allows complete navigation. If you do not specify a cursor type, adOpenForwardOnly is the default.

The LockType argument specifies what type of locks the provider should place on the underlying data source when opening the recordset. This is set using one of the LockTypeEnum values. This chapter uses only the following two lock types, corresponding to normal and disconnected recordsets, respectively: adLockReadOnly and adLockBatchOptimistic.

The Options argument here is the same as the Options argument covered in the Connection object's Execute method earlier in the chapter. It is used to tell the provider how to interpret and execute the contents of the Source argument.

The Close Method

This method closes the `Recordset` object. This does not free any memory used by the recordset. To free up the memory used by the `Recordset` object, you must set the `Recordset` object variable to `Nothing`.

The Move Methods

When a recordset is first opened, the *current record pointer* is positioned on the first record in the recordset. The `Move` methods are used to navigate through the records in an open recordset. They do this by repositioning the `Recordset` object's current record pointer. The following `Move` methods are used in this chapter:

- ❑ `MoveFirst` — Positions the current record pointer on the first record of the recordset.
- ❑ `MoveNext` — Positions the current record pointer to the next record in the recordset.

The following code sample demonstrates common recordset navigation handling:

```
' Verify that the Recordset contains data.
If Not rsData.EOF Then
    ' Loop until we reach the end of the Recordset.
    Do While Not rsData.EOF
        ' Perform some action on the current record's data.
        Debug.Print rsData.Fields(0).Value
        ' Move to the next record.
        rsData.MoveNext
    Loop
Else
    MsgBox "Error, no records returned.", vbCritical
End If
```

Pay particular attention to the use of the `MoveNext` method within the `Do While` loop. Omitting this is a very common error and will lead to an endless loop condition in your code. The very first line of code that you should place in the `Do While` loop is the call to `MoveNext`.

The NextRecordset Method

Some providers allow you to execute commands that return multiple recordsets. The `NextRecordset` method is used to move through these recordsets. The `NextRecordset` method clears the current recordset from the `Recordset` object, loads the next recordset into the `Recordset` object, and sets the current record pointer to the first record in that recordset. If the `NextRecordset` method is called and there are no more recordsets to retrieve, the `Recordset` object is set to `Nothing`. The following code sample demonstrates the use of the `NextRecordset` method:

```
' Verify that the Recordset contains more data.
Do While Not rsData Is Nothing
    ' Loop the records in the current recordset.
    Do While Not rsData.EOF
        ' Perform some action on the current record's data.
        Debug.Print rsData.Fields(0).Value
        ' Move to the next record.
        rsData.MoveNext
    Loop
```

```
        ' Return the next recordset
        Set rsData = rsData.NextRecordset
    Loop
```

Recordset Object Events

Recordset object events must be trapped by creating a WithEvents Recordset object variable in a class module. Trapping these events is necessary whenever you are using a Recordset object asynchronously, because these events are what notify your application that the Recordset object has completed its task.

Covering asynchronous recordset usage is beyond the scope of this chapter; however, the topic is important enough to deserve mention so that you can pursue it further if you like. The two most commonly used Recordset object events are:

- FetchComplete — This event is fired after all of the records have been retrieved when opening an asynchronous recordset.
- FetchProgress — The provider fires this event periodically to report the number of records retrieved so far during an asynchronous open operation. It is typically used to provide a visual progress indicator to the user.

Recordset Object Collections

Finish your look at the Recordset object by examining its collections.

The Fields Collection

The Fields collection contains the values, and information about those values, from the current record in a Recordset object. In Excel, the Fields collection is most commonly used to return the column names of each field in the recordset, prior to accessing the contents of the recordset using the CopyFromRecordset method of the Range object. The following example demonstrates how to read the field names from the Fields collection of a Recordset object:

```
        With Sheet1.Range("A1")
            For Each objField In rsData.Fields
                .Offset(0, lOffset).Value = objField.Name
                lOffset = lOffset + 1
            Next objField
        End With
```

The Properties Collection

This collection contains provider-specific or extended properties for the Recordset object. Some providers add important settings — called *extended properties* — that you will want to be aware of. The most important extended properties of each provider are covered in that provider's section.

The Command Object

The Command object is most commonly used for executing action queries. Action queries are queries that perform some action on the data source and do not return a resultset. Action queries include INSERT, UPDATE, and DELETE statements.

Command Object Properties

Begin by looking at the three most important Command object properties.

The ActiveConnection Property

This property is identical to the ActiveConnection property discussed in the section on the Recordset object.

The CommandText Property

This property is used to set the command that will be executed by the data provider. This property will normally be a SQL string or the name of a stored procedure. As you will see in the section on SQL Server, you must use the CommandText property along with the Parameters collection to take advantage of return values and output parameters in SQL Server stored procedures.

The CommandType Property

The CommandType property is identical to the Options argument to the Connection object's Execute method, covered earlier in the chapter. It is used to tell the provider how to interpret and execute the Command object's CommandText.

Command Object Methods

Only two of the Command object's methods are examined, because they are the most commonly used.

The CreateParameter Method

This method is used to manually create Parameter objects that can then be added to the Command object's Parameters collection. The CreateParameter object has the following syntax:

```
Set Parameter = command.CreateParameter([Name], [Type], [Direction], _
                                        [Size], [Value])
```

Name is the name of the parameter object. You can use this name to reference the Parameter object through the Command object's Parameters collection. When working with SQL Server, the name of a Parameter should be the same as the name of the stored procedure argument that it corresponds to.

Type indicates the data type of the parameter. It is specified as one of the DataTypeEnum constants. There are several dozen possible data types, so I will not go into them in any detail here. You will see a few of them in the examples section. The rest can be located in the ADO help file.

Direction is a ParameterDirectionEnum value that indicates whether the parameter will be used to pass data to an input argument, receive data from an output argument, or accept a return value from a stored procedure. Direction can be one of the following values:

- adParamInput — The parameter represents an input argument
- adParamInputOutput — The parameter represents an input/output argument
- adParamOutput — The parameter represents an output argument
- adParamReturnValue — The parameter represents a return value

`Size` is used to specify the size of `Parameter` in bytes, and is dependent on `Parameter`'s data type.

`Value` is used to provide an initial value for the `Parameter`.

The following code sample demonstrates how you can use the `CreateParameter` method in conjunction with the `Parameters` collection `Append` method to create a `Parameter` and append it to the `Parameters` collection with one line of code:

```
objCmd.Parameters.Append _
    objCmd.CreateParameter("MyParam", adInteger, adParamInput, 0)
```

The Execute Method

This method executes the text contained in the `Command` object's `CommandText` property. The `Execute` method has the following syntax for an action query (one that does not return a resultset):

```
command.Execute [RecordsAffected], [Parameters], [Options]
```

And for a select query:

```
Set Recordset = command.Execute([RecordsAffected], [Parameters], [Options])
```

The `RecordsAffected` and `Options` arguments are identical to the corresponding arguments for the `Connection` object's `Execute` method, described in the "Connection Object Methods" section. If you are executing a SQL statement that requires one or more parameters to be passed, you can supply an array of values to the `Parameters` argument, one for each parameter required.

Command Object Collections

The final section before delving into ADO in Excel covers the `Command` object's two collections.

The Parameters Collection

This collection contains all of the `Parameter` objects associated with the `Command` object. Parameters are used to pass arguments to SQL statements and stored procedures, as well as to receive output and return values from stored procedures.

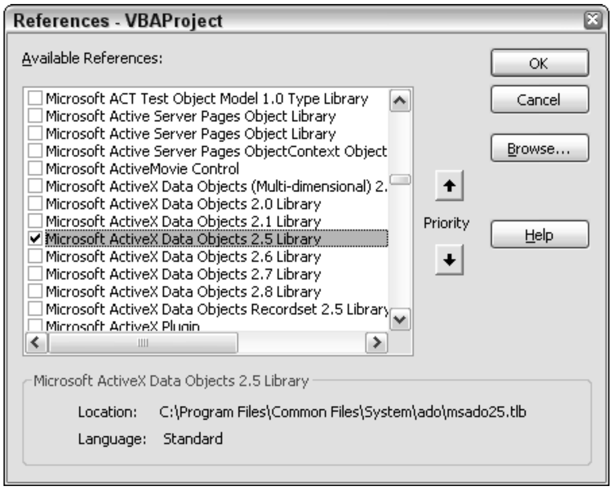
The Properties Collection

This collection contains provider-specific or extended properties for the `Command` object. Some providers add important settings that you will want to be aware of. The most important extended properties of each provider are covered in the provider-specific discussions later in the chapter.

Using ADO in Microsoft Excel Applications

Here's where it all comes together. This section combines the understanding of Excel programming that you've gained from previous chapters with the SQL and ADO techniques discussed so far in this chapter. Excel applications frequently require data from outside sources. The most common of these sources are Access and SQL Server databases. However, I've created applications that required source data from mainframe text file dumps and even Excel workbooks. As you'll see, ADO makes acquiring data from these various data sources easy.

To run the code examples shown in the sections that follow, you must set a reference from your Excel project to the ADO 2.5 Object Library. To do this, bring up the References dialog by selecting the Tools ➪ References menu item from within the VBE. Scroll down until you locate the entry labeled Microsoft ActiveX Data Objects 2.5 Library. Place a check mark beside this entry and click OK (see Figure 20-4).



- ☐ Microsoft ACT Test Object Model 1.0 Type Library
- ☐ Microsoft Active Server Pages Object Library
- ☐ Microsoft Active Server Pages Object Library
- ☐ Microsoft Active Server Pages ObjectContext Object
- ☐ Microsoft ActiveMovie Control
- ☐ Microsoft ActiveX Data Objects (Multi-dimensional) 2.
- ☐ Microsoft ActiveX Data Objects 2.0 Library
- ☐ Microsoft ActiveX Data Objects 2.1 Library
- ✓ Microsoft ActiveX Data Objects 2.5 Library
- ☐ Microsoft ActiveX Data Objects 2.6 Library
- ☐ Microsoft ActiveX Data Objects 2.7 Library
- ☐ Microsoft ActiveX Data Objects 2.8 Library
- ☐ Microsoft ActiveX Data Objects Recordset 2.5 Library
- ☐ Microsoft ActiveX Plugin

Microsoft ActiveX Data Objects 2.5 Library

Location: C:\Program Files\Common Files\System\ado\msado25.tlb

Language: Standard

Figure 20-4

Note that it's perfectly normal to have multiple versions of the ADO object library available.

Using ADO with Microsoft Access

Because Excel applications that utilize data from Microsoft Access tend to have less complicated data access requirements than those using SQL Server, Access is used to introduce the basics of ADO.

In this section you'll utilize the Northwind database. This is a sample database provided with Microsoft Access 2007. If you don't have this database available, you will need to install it to run the example code.

Connecting to Microsoft Access

ADO connects to Microsoft Access databases through the use of the Microsoft Office 12 Access Database Engine OLE DB Provider. To connect to a Microsoft Access database, you simply specify this provider in the ADO connection string and then include any additional provider-specific arguments required. The following is a summary of the connection string arguments you will most frequently use when connecting to an Access database:

- ❑ `Provider=Microsoft.ACE.OLEDB.12.0` (required)
- ❑ `Data Source=[full path and filename to the Access database]` (required)

- `Mode=mode` (optional) — The three most commonly used settings for this property are:
    - `adModeShareDenyNone` — Opens the database and allows complete shared access to other users. This is the default setting if the `Mode` argument is not specified.
    - `adModeShareDenyWrite` — Opens the database and allows other users read access but prevents write access.
    - `adModeShareExclusive` — Opens the database in exclusive mode, which prevents any other users from connecting to the database.
- `User ID=username` (optional) — The username to use when connecting to the database. If the database requires a username and it is not supplied, the connection will fail.
- `Password=password` (optional) — If a password is required to connect to the database, this argument is used to supply it. Again, the connection will fail if it is required and not supplied, or is incorrect.

The following example shows a connection string that uses all of these arguments:

```
Public Const gsCONNECTION As String = _
    "Provider= Microsoft.ACE.OLEDB.12.0;" & _
    "Data Source=C:\Files\Northwind 2007.accdb;" & _
    "Mode=Share Exclusive;" & _
    "User ID=Admin;" & _
    "Password=password"
```

Retrieving Data from Microsoft Access Using a Plain Text Query

The following procedure demonstrates how to retrieve data from a Microsoft Access database using a plain text (sometimes referred to as ad hoc) query and place it on an Excel worksheet:

```
Public Sub PlainTextQuery()

    Dim rsData As ADODB.Recordset
    Dim sConnect As String
    Dim sSQL As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Northwind 2007.accdb"

    ' Create the SQL Statement.
    sSQL = "SELECT Company, [First Name] + ' ' + [Last Name] " & _
            "FROM Customers " & _
            "WHERE [Country/Region] = 'USA' " & _
            "ORDER BY Company;"

    ' Create the Recordset object and run the query.
    Set rsData = New ADODB.Recordset
    rsData.Open sSQL, sConnect, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    ' Make sure we got records back
```

```
        If Not rsData.EOF Then
            ' Dump the contents of the recordset onto the worksheet.
            Sheet1.Range("A2").CopyFromRecordset rsData
            ' Close the Recordset object.
            rsData.Close
            ' Add headers to the worksheet.
            With Sheet1.Range("A1:B1")
                .Value = Array("Company", "Contact Name")
                .Font.Bold = True
            End With
            ' Fit the column widths to the data.
            Sheet1.UsedRange.EntireColumn.AutoFit
        Else
            ' Close the Recordset object.
            rsData.Close
            MsgBox "Error: No records returned.", vbCritical
        End If

        ' Destroy the Recordset object.
        Set rsData = Nothing

    End Sub
```

There are a number of things to note about this procedure:

- The only ADO object used was the `Recordset` object. As mentioned at the beginning of the ADO section, all of the top-level ADO objects can be created and used independently. If you were going to perform multiple queries over the course of your application, you would have created a separate, publicly scoped `Connection` object to take advantage of ADO's connection-pooling feature.
- The syntax of the `Recordset.Open` method has been optimized for maximum performance. You've told the provider what type of command is in the `Source` argument (`adCmdText`, a plain text query), and you've opened a forward-only, read-only, server-side cursor (server-side is the default if the `Recordset.CursorLocation` property is not specified). This type of cursor is often referred to as a *firehose cursor*, because it's the fastest way to retrieve data from a database.
- You do not make any modifications to the destination worksheet until you are sure you have successfully retrieved data from the database. This avoids having to undo anything if the query fails.
- You dump the data onto the destination worksheet and close the recordset as quickly as possible. In most data-access situations, you will be dealing with a multi-user environment, where multiple users can access the database simultaneously. Getting in and out of the database as quickly as possible is critical to preventing contention problems, or situations in which two users attempt to perform mutually incompatible actions on the same piece of data at the same time.
- Note the use of the `CopyFromRecordset` method of the Excel `Range` object. This is by far the fastest method for moving data out of a recordset and onto a worksheet. As you'll see, it doesn't fit every data-access situation, but it's the method of choice in any situation where its use is possible. Note that the `CopyFromRecordset` method does not copy the field names, only data.

Retrieving Data from Microsoft Access Using a Stored Query

Microsoft Access allows you to create and store SQL queries in the database. You can retrieve data from these stored queries just as easily as you can use a plain text SQL statement. The following procedure demonstrates this:

```
Public Sub SavedQuery()

    Dim objField As ADODB.Field
    Dim rsData As ADODB.Recordset
    Dim lOffset As Long
    Dim sConnect As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Northwind 2007.accdb"

    ' Create the Recordset object and run the query.
    Set rsData = New ADODB.Recordset
    rsData.Open "[Product Sales By Category]", sConnect, _
        adOpenForwardOnly, adLockReadOnly, adCmdTable

    ' Make sure we got records back
    If Not rsData.EOF Then
        ' Add headers to the worksheet.
        With Sheet1.Range("A1")
            For Each objField In rsData.Fields
                .Offset(0, lOffset).Value = objField.Name
                lOffset = lOffset + 1
            Next objField
            .Resize(1, rsData.Fields.Count).Font.Bold = True
        End With
        ' Dump the contents of the recordset onto the worksheet.
        Sheet1.Range("A2").CopyFromRecordset rsData
        ' Close the Recordset object.
        rsData.Close
        ' Fit the column widths to the data.
        Sheet1.UsedRange.EntireColumn.AutoFit
    Else
        ' Close the Recordset object.
        rsData.Close
        MsgBox "Error: No records returned.", vbCritical
    End If

    ' Destroy the Recordset object.
    Set rsData = Nothing

End Sub
```

There are two important points to note about this procedure:

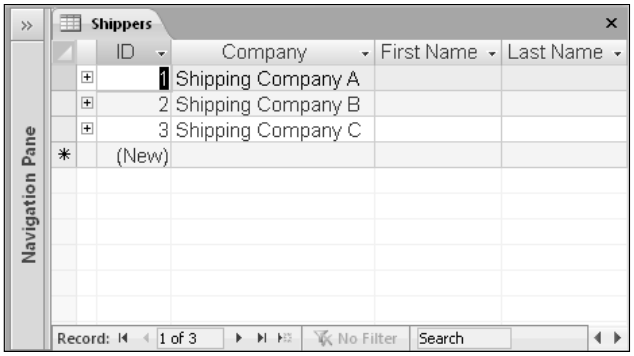
- Examine the differences between the `Recordset.Open` method used in this procedure and the one used in the plain text query. In this case, rather than providing a SQL string, you specified the name of the stored query you wanted to execute. You also told the provider that the type of query being executed was a table query. The Access OLE DB provider treats stored queries and queries of entire database tables in the same manner.

- Because you did not create the SQL statement yourself, you did not know the names of the fields you were retrieving, or even how many fields there were. Therefore, to create the correct set of headers for each column in the destination worksheet, you needed to loop the `Fields` collection of the `Recordset` object and determine this information dynamically. To accomplish this, the recordset had to be open, so you added the fields to the worksheet prior to closing the recordset.

Inserting, Updating, and Deleting Records with Plain Text SQL in Microsoft Access

Executing plain text `INSERT`, `UPDATE`, and `DELETE` statements uses virtually identical methodology. Therefore, you'll examine these action queries by inserting a new record, updating that record, and then deleting it, all within the same procedure. This, of course, is not normally something you would do. You can take this generic procedure, however, and create a single-purpose insert, update, or delete procedure by simply removing the sections that you don't need.

Use the `Shippers` table from the Northwind database in the next procedure. The first few columns of this table are shown in Figure 20-5.



ID	Company	First Name	Last Name
1	Shipping Company A		
2	Shipping Company B		
3	Shipping Company C		
(New)			

Figure 20-5

Notice that the last row in the ID column contains the value `(New)`. This isn't really a value; rather, it's a prompt that alerts you to the fact that values for the ID column are automatically generated by the Access database. This column is the primary key for the `Shippers` table, and AutoNumber fields are a common method used to generate the unique value required for the primary key. You don't (and can't) set or change the value of an AutoNumber field. If you need to maintain a reference to a new record that you've inserted into the table, you'll need to retrieve the value that was assigned to that record by the AutoNumber field. You'll see how this is done in the example that follows:

```
Public Sub InsertUpdateDelete()

    Dim objCommand As ADODB.Command
    Dim rsData As ADODB.Recordset
    Dim lRecordsAffected As Long
    Dim lKey As Long
    Dim sConnect As String

    On Error GoTo ErrorHandler

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Northwind 2007.accdb;" & _
        "Mode=Share Exclusive"

    ' Create the Command object we'll use for all three queries.
    Set objCommand = New ADODB.Command
    objCommand.ActiveConnection = sConnect

    ''''' INSERT a new record into the database '''''''''''''''''
    ' Load the SQL string into the Command object.
    objCommand.CommandText = "INSERT INTO Shippers" & _
            "(Company, [Business Phone]) " & _
            "VALUES('Air Carriers', '(205) 555-1212');"
    ' Execute the SQL statement.
    objCommand.Execute RecordsAffected:=lRecordsAffected, _
        Options:=adCmdText Or adExecuteNoRecords
    ' Check for errors. Only one record should have been affected.
    If lRecordsAffected <> 1 Then Err.Raise _
        Number:=vbObjectError + 1024, _
        Description:="Error executing INSERT statement."

    ' Retrieve the primary key generated for our new record.
    objCommand.CommandText = "SELECT @@IDENTITY;"
    Set rsData = objCommand.Execute(Options:=adCmdText)
    ' Check for errors. The recordset should contain data.
    If rsData.EOF Then Err.Raise _
        Number:=vbObjectError + 1024, _
        Description:="Error retrieving primary key value."
    ' Store the primary key value for later use.
    lKey = rsData.Fields(0).Value
    rsData.Close

    ''''' UPDATE the record we just created '''''''''''''''''''
    ' Load the SQL string into the Command object.
    objCommand.CommandText = "UPDATE Shippers " & _
        "SET [Business Phone]='(206) 546-0086' " & _
        "WHERE ID=" & CStr(lKey) & ";"
    ' Execute the SQL statement.
    objCommand.Execute RecordsAffected:=lRecordsAffected, _
        Options:=adCmdText Or adExecuteNoRecords
    ' Check for errors. Only one record should have been affected.
    If lRecordsAffected <> 1 Then Err.Raise _
        Number:=vbObjectError + 1024, _
        Description:="Error executing UPDATE statement."

    ''''' DELETE our record from the database '''''''''''''''''
    ' Load the SQL string into the Command object.
    objCommand.CommandText = "DELETE FROM Shippers " & _
        "WHERE ID = " & CStr(lKey) & ";"
    ' Execute the SQL statement.
    objCommand.Execute RecordsAffected:=lRecordsAffected, _
```

```
            Options:=adCmdText Or adExecuteNoRecords
        ' Check for errors. Only one record should have been affected.
        If lRecordsAffected <> 1 Then Err.Raise _
            Number:=vbObjectError + 1024, _
            Description:="Error executing DELETE statement."

    ErrorExit:

        ' Destroy our ADO objects.
        Set objCommand = Nothing
        Set rsData = Nothing

        Exit Sub

    ErrorHandler:
        MsgBox Err.Description, vbCritical
        Resume ErrorExit
    End Sub
```

Quickly review what you've done in this procedure. First you inserted a new record into the `Shippers` table. Then you retrieved the primary key that had been assigned to that new record by the database (more on this in a moment). Next you used the primary key of your new record to locate it and modify the telephone number in its `Business Phone` field. Finally, you used the primary key of the record to locate it and delete it from the database.

Important things to note about this procedure:

- You used the same ADO `Command` object throughout the procedure. All that was required to execute different commands was to load new SQL statements into the `Command.CommandText` property.
- The process of preparing and executing the command and then checking for errors upon completion was identical for all three types of action query.
- After inserting a new record in the first part of the procedure, you needed to retrieve the primary key value that had been assigned to that record by the ID AutoNumber field. You did this by querying the value of the `@@IDENTITY` system variable. This is a variable maintained by the Access database that holds the value of the most recently assigned AutoNumber field. In most cases, you must be sure to query this value immediately after performing the insert. The `@@IDENTITY` variable is a database-wide variable, so your primary key value will be overwritten if any other user performs a similar insert before you query it. You prevented the possibility of this occurring in this example by opening the database in exclusive mode (note the `Mode` argument in the connection string).

Using ADO with Microsoft SQL Server

The previous section on Microsoft Access covered the basics of performing the various types of queries in ADO. Because ADO is designed to present a common gateway to different data sources, there isn't a lot of difference in these basic operations whether your database is in Access or in SQL Server. Therefore, after a brief introduction to the few important differences that arise when using ADO with SQL Server, this section covers more advanced topics, including stored procedures, multiple recordsets, and disconnected recordsets.

The examples in this section use the SQL Server version of the Northwind sample database. This database is similar to the Access 2007 version of Northwind used previously, but there will be some differences in the names and data types of various fields.

Connecting to Microsoft SQL Server

To connect to a Microsoft SQL Server database, you simply specify the OLE DB provider for SQL Server in the ADO connection string, and then include any additional provider-specific arguments required. The following is a summary of the connection string arguments you will most frequently use when connecting to a SQL Server database:

- `Provider=SQLOLEDB;`
- `Data Source=server name;` — This will typically be the NetBIOS name of the computer that SQL Server is installed on. If SQL Server is installed as a named instance, the server name will have the following syntax: `NetBIOS Name\SQL Server Instance Name`.
- `Initial Catalog=database name;` — Unlike Access, one instance of SQL Server can contain many databases. This argument will contain the name of the database you want to connect to.
- `User ID=username;` — The username for SQL Server authentication.
- `Password=password;` — The password for SQL Server authentication.
- `Network Library=netlib;` — By default, the SQL Server OLE DB provider will attempt to use *named pipes network protocol* to connect to SQL Server. This is required for using Windows integrated security (explained later). There are many instances, however, where it is not possible to use named pipes. These include accessing SQL Server from a Windows 9x operating system and accessing SQL Server over the Internet. In these cases, the preferred protocol for connecting to SQL Server is TCP/IP. This can be specified on each machine by using the SQL Server Client Network Utility, or you can simply use the Network Library connection string argument to specify the name of the TCP/IP network library, which is `dbmssocn`.
- `Integrated Security=SSPI;` — This connection string argument specifies that you want to use Windows integrated security rather than SQL Server authentication. The `User ID` and `Password` arguments will be ignored if this argument is present.

A Note About SQL Server Security

SQL Server can be set to use three types of security: SQL Server authentication, Windows integrated security, and mixed mode. SQL Server authentication means that separate user accounts must be added to SQL Server, and each user must supply a SQL Server username and password to connect.

This type of security is most commonly used when SQL Server must be accessed from outside the network. With Windows integrated security, SQL Server recognizes the same usernames and passwords that are used to log in to the Windows network. Mixed mode simply means you can use either one of the two.

Following are examples of two different SQL Server connection strings. The first example shows a connection string that uses SQL Server authentication and the TCP/IP connection protocol. The second example shows a connection string that uses Windows integrated security:

```
Public Const gsCONNECTION As String = _
    "Provider=SQLOLEDB;" & _
    "Data Source=ComputerName\SQLServerName;" & _
    "Initial Catalog=Northwind;" & _
    "User ID=User;Password=password;" & _
    "Network Library=dbmssocn"

Public Const gsCONNECTION As String = _
    "Provider=SQLOLEDB;" & _
    "Data Source=ComputerName\SQLServerName;" & _
    "Initial Catalog=Northwind;" & _
    "Integrated Security=SSPI"
```

Microsoft SQL Server Stored Procedures

The syntax for executing plain text (or ad hoc) queries against SQL Server is identical to that which you used in the example for Access. The only difference is the contents of the connection string. When programming with SQL Server, however, it is more common to call SQL Server *stored procedures*.

Stored procedures are simply precompiled SQL statements that can be accessed by name from the database. They are much like VBA procedures in that they can accept arguments and return values. An example of a simple stored procedure that queries the `Customers` table is shown here:

```
CREATE PROC spGetCustomerNames
    @Country  nvarchar(24)
AS
    SELECT    CustomerID,
              CompanyName,
              ContactName
    FROM      Customers
    WHERE     Country = @Country
    ORDER BY  CompanyName
```

This stored procedure takes one argument, `@Country`, and returns a recordset containing the values for the fields specified in the `SELECT` list for customers whose country matches the value passed to the `@Country` argument.

ADO provides a very quick and simple way to execute stored procedures using the `Connection` object. ADO treats all stored procedures in the currently connected database as dynamic methods of the `Connection` object. You can call a stored procedure exactly like any other `Connection` object method, passing any arguments to the stored procedure as method arguments, and optionally passing a `Recordset` object as the last argument if the stored procedure returns a recordset.

This method is best used for "one-off" procedures rather than those you will execute multiple times, because it isn't the most efficient method. However, it is significantly easier to code. The following example demonstrates executing the previous stored procedure as a method of the `Connection` object:

```
Public Sub ExecuteStoredProcAsMethod()

    Dim objConn As ADODB.Connection
    Dim rsData As ADODB.Recordset
    Dim sConnect As String

    ' Create the connection string.
    sConnect = "Provider=SQLOLEDB;Data Source=P2800\P2800;" & _
        "Initial Catalog=Northwind;Integrated Security=SSPI"

    ' Create the Connection and Recordset objects.
    Set objConn = New ADODB.Connection
    Set rsData = New ADODB.Recordset

    ' Open the connection and execute the stored procedure.
    objConn.Open sConnect
    objConn.spGetCustomerNames "UK", rsData

    ' Make sure we got records back
    If Not rsData.EOF Then
        ' Dump the contents of the recordset onto the worksheet.
        Sheet1.Range("A1").CopyFromRecordset rsData
        ' Close the recordset
        rsData.Close
        ' Fit the column widths to the data.
        Sheet1.UsedRange.EntireColumn.AutoFit
    Else
        MsgBox "Error: No records returned.", vbCritical
    End If

    ' Clean up our ADO objects.
    If CBool(objConn.State And adStateOpen) Then objConn.Close
    Set objConn = Nothing
    Set rsData = Nothing

End Sub
```

In this procedure, you executed the `spGetCustomerNames` stored procedure and passed it the value `"UK"` for its `@Country` argument. This populated the `rsData` recordset with all of the customers located in the UK. Note that the `Connection` object must be opened before the dynamic methods are populated, and that you must instantiate the `Recordset` object prior to passing it as an argument.

The most efficient way to handle stored procedures that will be executed multiple times is to prepare a publicly scoped `Command` object to represent them. The `Connection` will be stored in the `Command` object's `ActiveConnection` property, the stored procedure's name will be stored in the `Command` object's `CommandText` property, and any arguments to the stored procedure will be used to populate the `Command` object's `Parameters` collection.

Once this `Command` object has been created, it can be executed as many times as you like over the course of your application, without incurring the overhead required to perform the tasks just described with each execution.

For this example, create a simple stored procedure that you can use to insert new records into the Shippers table:

```
CREATE PROC spInsertShippers
    @CompanyName  nvarchar(40),
    @Phone        nvarchar(24)
AS
    INSERT INTO Shippers(CompanyName, Phone)
    VALUES(@CompanyName, @Phone)
    RETURN @@IDENTITY
```

As you can see, the stored procedure has two arguments, @CompanyName and @Phone, which are used to collect the values to insert into those respective fields in the Shippers table. Similar to the ID field in the Access version of the Northwind database, the ShipperID field in the SQL Server version of Northwind is populated automatically by the database any time a new record is inserted. You retrieve this automatically assigned value in a similar fashion, through the use of SQL Server's @@IDENTITY system function. In this case, however, you won't have to make a separate query to retrieve the Shipper ID value because it will be returned to you by the stored procedure.

To present a more realistic application scenario, the following example uses publicly scoped Connection and Command objects, procedures to create and destroy the connection, a procedure to prepare the Command object for use, and a procedure that demonstrates how to use the Command object:

```
Public Const gszCONNECTION As String = _
    "Provider=SQLOLEDB;Data Source=P2800\P2800;" & _
    "Initial Catalog=Northwind;Integrated Security=SSPI"

Public gobjCmd As ADODB.Command
Public gobjConn As ADODB.Connection

Private Sub CreateConnection()
    ' Create the Connection object.
    Set gobjConn = New ADODB.Connection
    gobjConn.Open gszCONNECTION
End Sub

Private Sub DestroyConnection()
    ' Check to see if connection is still open before attempting to close it.
    If CBool(gobjConn.State And adStateOpen) Then gobjConn.Close
    Set gobjConn = Nothing
End Sub

Private Sub PrepareCommandObject()

    ' Create the Command object.
    Set gobjCmd = New ADODB.Command
    Set gobjCmd.ActiveConnection = gobjConn
    gobjCmd.CommandText = "spInsertShippers"
    gobjCmd.CommandType = adCmdStoredProc

    ' Load the parameters collection. The first parameter
    ' is always the stored procedure return value.
    gobjCmd.Parameters.Append _
```

```
        gobjCmd.CreateParameter("@RETURN_VALUE", adInteger, _
                                adParamReturnValue, 0)
    gobjCmd.Parameters.Append _
        gobjCmd.CreateParameter("@CompanyName", adVarWChar, _
                                adParamInput, 40)
    gobjCmd.Parameters.Append _
        gobjCmd.CreateParameter("@Phone", adVarWChar, _
                                adParamInput, 24)

End Sub

Public Sub UseCommandObject()

    Dim lKeyValue As Long
    Dim lNumAffected As Long

    On Error GoTo ErrorHandler

    ' Create the Connection and the reusable Command object.
    CreateConnection
    PrepareCommandObject

    ' Set the values of the input parameters.
    gobjCmd.Parameters("@CompanyName").Value = "Air Carriers"
    gobjCmd.Parameters("@Phone").Value = "(206) 555-1212"

    ' Execute the Command object and check for errors.
    gobjCmd.Execute Recordsaffected:=lNumAffected, _
                    Options:=adExecuteNoRecords
    If lNumAffected <> 1 Then Err.Raise Number:=vbObjectError + 1024, _
        Description:="Error executing Command object."

    ' Retrieve the primary key value for the new record.
    lKeyValue = gobjCmd.Parameters("@RETURN_VALUE").Value
    Debug.Print "The key value of the new record is: " & CStr(lKeyValue)

ErrorExit:

    ' Destroy the Command and Connection objects.
    Set gobjCmd = Nothing
    DestroyConnection

    Exit Sub

ErrorHandler:
    MsgBox Err.Description, vbCritical
    Resume ErrorExit
End Sub
```

A few things to note about the mini application:

- In a normal application, you would not create and destroy the `Connection` and `Command` objects in the `UseCommandObject` procedure. These objects are intended for reuse and therefore typically would be created when your application first started and destroyed just before it ended.

- When constructing and using the `Command` object's `Parameters` collection, keep in mind that the first parameter is always reserved for the stored procedure return value, even if the stored procedure doesn't have a return value.
- Even though you didn't make any particular use of the Shipper ID value returned from the stored procedure for the new record, in a normal application this value would be very important. The `CompanyName` and `Phone` fields are for human consumption; the primary key value is how the database identifies the record. For example, in the Northwind database, the Shipper ID is a required field for entering new records into the `Orders` table. Therefore, if you planned on adding an order that was going to use the new shipper, you would have to know the Shipper ID.

Multiple Recordsets

The SQL Server OLE DB provider is an example of a provider that allows you to execute a SQL statement that returns multiple recordsets. This feature comes in very handy when you need to populate multiple controls on a form with lookup-table information from the database. You can combine all of the lookup-table `SELECT` queries into a single stored procedure and then loop through the individual recordsets, assigning their contents to the corresponding controls.

For example, if you needed to create a user interface for entering information into the `Orders` table, you would need information from several related tables, including `Customers` and `Shippers`, as shown in Figure 20-6.

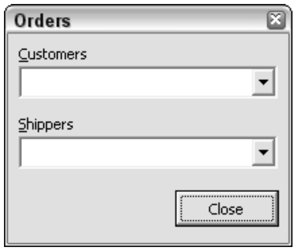


Figure 20-6

Create an abbreviated example of a stored procedure that returns the lookup information from these two tables, and then use the result to populate drop-downs on a UserForm:

```
CREATE PROC spGetLookupValues
AS
    -- Customers lookup table info.
    SELECT    CustomerID,
              CompanyName
    FROM      Customers

    -- Shippers lookup table info.
    SELECT    ShipperID,
              CompanyName
    FROM      Shippers
```

Note that this stored procedure contains two separate `SELECT` statements. These will populate two independent recordsets when the stored procedure is executed using ADO. The double dashes that you see at the beginning of the lines above each `SELECT` statement are T-SQL comment prefixes.

The following procedure is an example of a `UserForm_Initialize` event that populates drop-downs on the UserForm with the results of the `spGetLookupValues` stored procedure. For the purpose of this example, assume that the public `Connection` object `gobjConn` used in the previous example is still open and available for use:

```
Private Sub UserForm_Initialize()

    Dim rsData As ADODB.Recordset

    ' Create and open the Recordset object.
    Set rsData = New ADODB.Recordset
    rsData.Open "spGetLookupValues", gobjConn, _
                adOpenForwardOnly, adLockReadOnly, adCmdStoredProc

    ' The first recordset contains the customer list.
    Do While Not rsData.EOF
        ' Load the dropdown with the recordset values.
        ddCustomers.AddItem rsData.Fields(1).Value
        rsData.MoveNext
    Loop
    Set rsData = rsData.NextRecordset

    ' The second recordset contains the shippers list.
    Do While Not rsData.EOF
        ' Load the dropdown with the recordset values.
        ddShippers.AddItem rsData.Fields(1).Value
        rsData.MoveNext
    Loop
    Set rsData = rsData.NextRecordset

    ' No need to clean up the Recordset object at this point,
    ' it will be closed and set to nothing after the last
    ' call to the NextRecordset method.

End Sub
```

One thing to note about the method demonstrated here is that it requires prior knowledge of the number and order of recordsets returned by the call to the stored procedure. Also left out is any handling of the primary key values associated with the lookup table descriptions. In a real-world application, you would need to maintain these keys (I prefer using a hidden column in a multi-column drop-down for this purpose) so you could retrieve the primary key value that corresponded to the user's selection in each drop-down.

Disconnected Recordsets

The "Retrieving Data from Microsoft Access Using a Plain Text Query" section mentioned that getting in and out of the database as quickly as possible was an important goal. However, the `Recordset` object is a powerful tool that you would often like to hold onto and use without locking other users out of the database. The solution to this problem is the ADO disconnected recordset feature.

A disconnected recordset is a `Recordset` object whose connection to its data source has been severed, but that can still remain open. The result is a fully functional `Recordset` object that does not hold any locks in the database from which it was queried. Disconnected recordsets can remain open as long as you need them, they can be reconnected to and resynchronized with the data source, and they can even be persisted to disk for later retrieval. A few of these capabilities are examined in the following example.

Imagine you wanted to implement a feature that would allow users to view any group of customers they chose. Running a query against the database each time the user specified a different criterion would be an inefficient way to accomplish this. A much better alternative would be to query the complete set of customers from the database and hold them in a disconnected recordset. You could then use the `Filter` property of the `Recordset` object to quickly extract the set of customers that your user requested.

The following example shows all of the elements required to create a disconnected recordset. Again, assume the availability of the public `gobjConn` `Connection` object:

```
Public grsData As ADODB.Recordset

Public Sub CreateDisconnectedRecordset ()

    Dim szSQL As String

    ' Create the SQL Statement.
    szSQL = "SELECT CustomerID, CompanyName, ContactName, Country " & _
            "FROM Customers"

    ' Steps to creating a disconnected recordset:
    ' 1) Create the Recordset object.
    Set grsData = New ADODB.Recordset
    ' 2) Set the cursor location to client side.
    grsData.CursorLocation = adUseClient
    ' 3) Set the cursor type to static.
    grsData.CursorType = adOpenStatic
    ' 4) Set the lock type to batch optimistic.
    grsData.LockType = adLockBatchOptimistic
    ' 5) Open the recordset.
    grsData.Open szSQL, gobjConn, , , adCmdText
    ' 6) Set the Recordset's Connection object to Nothing.
    Set grsData.ActiveConnection = Nothing

    ' grsData is now a disconnected recordset.
    Sheet1.Range("A1").CopyFromRecordset grsData

End Sub
```

Note that the `Recordset` object variable in the preceding example is declared with public scope. If you were to declare the `Recordset` object variable at the procedure level, VBA would automatically destroy it when the procedure ended and it would no longer be available for use.

Six crucial steps are required to successfully create a disconnected recordset. It's possible to combine several of them into one step during the `Recordset.Open` method, and it's more efficient to do so, but they are separated here for the sake of clarity:

❑ You must create a new, empty `Recordset` object to start with.

❑ You must set the cursor location to client-side. Because the recordset will be disconnected from the server, the cursor cannot be managed there. Note that this setting must be made *before* you open the recordset. It is not possible to change the cursor location once the recordset is open.

- The ADO client-side cursor engine supports only one type of cursor, the static cursor, so this is what the `CursorType` property must be set to.
- ADO has a lock type specifically designed for disconnected recordsets called Batch Optimistic. The Batch Optimistic lock type makes it possible to reconnect the disconnected recordset to the database and update the database with records that have been modified while the recordset was disconnected. This operation is beyond the scope of this chapter, so note that the Batch Optimistic lock type is required in order to create a disconnected recordset.
- Opening the recordset is the next step. This example used a plain text SQL query. This is not a requirement. You can create a disconnected recordset from almost any source that can be used to create a standard recordset. The client-side cursor engine lacks a few capabilities, however; multiple recordsets are one example.
- The final step is disconnecting the recordset from the data source. This is accomplished by setting the recordset's `Connection` object to `Nothing`. If you recall from the "Recordset Object Properties" section, the `Connection` object associated with a `Recordset` object is accessed through the `Recordset.ActiveConnection` property. Setting this property to `Nothing` severs the connection between the recordset and the data source.

Now that you have a disconnected recordset to work with, what kinds of things can you do with it? Just about any operation the `Recordset` object allows. Say that the user wanted to see a list of customers located in Germany, sorted by alphabetical order. This is how you'd accomplish that task:

```
' Set the Recordset filter to display only records
' whose Country field is Germany.
grsData.Filter = "Country = 'Germany'"
' Sort the records by CompanyName.
grsData.Sort = "CompanyName"
' Load the processed data onto Sheet1
Sheet1.Range("A1").CopyFromRecordset grsData
```

If you are working in a busy multi-user environment, the data in your disconnected recordset may become out-of-date during the course of your application due to other users inserting, updating, and deleting records. You can solve this problem by requerying the recordset. As demonstrated by the following example, this is a simple matter of reconnecting to the data source, executing the `Recordset.Requery` method, then disconnecting from the data source:

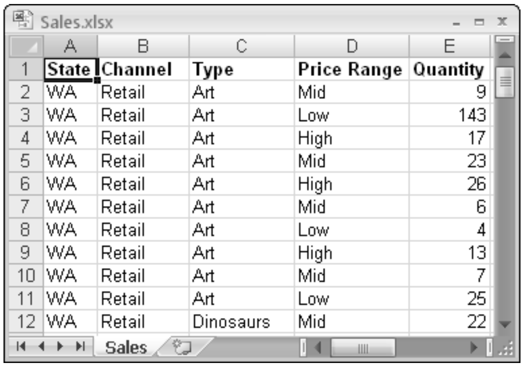
```
' Reconnect to the data source.
Set grsData.ActiveConnection = gobjConn
' Rerun the Recordset object's underlying query,
grsData.Requery Options:=adCmdText
' Disconnect from the data source.
Set grsData.ActiveConnection = Nothing
```

Using ADO with Non-Standard Data Sources

This section describes how you can use ADO to access data from two common non-standard data sources (data sources that are not strictly considered databases), Excel workbooks, and text files. Although the idea may seem somewhat counterintuitive, ADO is often the best choice for retrieving data from workbooks and text files because it eliminates the often lengthy process of opening them in Excel. Using ADO also allows you to take advantage of the power of SQL to do exactly what you want in the process.

Querying Microsoft Excel Workbooks

When using ADO to access data from Excel 2007 workbooks, you use the same OLE DB provider that you used earlier in this chapter to access data from Microsoft Access 2007. In addition to Access, this provider also supports most *ISAM data sources* (data sources that are laid out in a tabular, row and column format). You will use the `Sales.xlsx` workbook, shown in Figure 20-7, as the data source for the Excel examples.



	A	B	C	D	E
1	State	Channel	Type	Price Range	Quantity
2	WA	Retail	Art	Mid	9
3	WA	Retail	Art	Low	143
4	WA	Retail	Art	High	17
5	WA	Retail	Art	Mid	23
6	WA	Retail	Art	High	26
7	WA	Retail	Art	Mid	6
8	WA	Retail	Art	Low	4
9	WA	Retail	Art	High	13
10	WA	Retail	Art	Mid	7
11	WA	Retail	Art	Low	25
12	WA	Retail	Dinosaurs	Mid	22

Figure 20-7

When using ADO to work with Excel, the workbook file takes the place of the database, while worksheets within the workbook, as well as named ranges, serve as tables. Compare a connection string used to connect to an Access database with a connection string used to connect to an Excel workbook.

Connection string to an Access database:

```
sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
           "Data Source=C:\Files\Northwind 2007.accdb;"
```

Connection string to an Excel workbook:

```
sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
           "Data Source=C:\Files\Sales.xlsx;" & _
           "Extended Properties=Excel 12.0;"
```

Note that the same provider is used, and that the full path and filename of the Excel workbook takes the place of the full path and filename of the Access database. The only difference is that you must specify the type name of the data source you want to connect to in the `Extended Properties` argument. When connecting to Excel 2007, you set the `Extended Properties` argument to `Excel 12.0`. For versions of Excel earlier than 2007, you set the `Extended Properties` argument to `Excel 8.0`.

You query data from an Excel worksheet using a plain text SQL statement exactly like you would query a database table. However, the format of the table name is different for Excel queries. You can specify the table that you want to query from an Excel workbook in one of four different ways:

- ❑ Worksheet Name Alone — When using the name of a specific worksheet as the table name in your SQL statement, the worksheet name must be suffixed with a `$` character and surrounded with square brackets. For example, `[Sheet1$]` is a valid worksheet table name. If the worksheet name contains spaces or non-alphanumeric characters, you must surround it with single quotes. An example of this is `['My Sheet$']`.
- ❑ Worksheet-level Range Name — You can use a worksheet-level range name as a table name in your SQL statement. Simply prefix the range name with the worksheet name it belongs to, using the formatting conventions just described. An example of this would be `[Sheet1$SheetLevelName]`.
- ❑ Specific Range Address — You can specify the table in your SQL statement as a specific range address on the target worksheet. The syntax for this method is identical to that for a worksheet-level range name: `[Sheet1$A1:E20]`.
- ❑ Workbook-level Range Name — You can also use a workbook-level range name as the table in your SQL statement. In this case there is no special formatting required. You simply use the name directly, without brackets.

Although your sample workbook contains only one worksheet, this is not a requirement. The target workbook can contain as many worksheets and named ranges as you wish. You simply need to know which one to use in your query. The following procedure demonstrates all four table-specifying methods just discussed:

```
Public Sub QueryWorksheet()

    Dim rsData As ADODB.Recordset
    Dim sConnect As String
    Dim sSQL As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Sales.xlsx;" & _
        "Extended Properties=Excel 12.0;"

    ' Query based on the worksheet name.
    'sSQL = "SELECT * FROM [Sales$]"
    ' Query based on a sheet-level range name.
    'sSQL = "SELECT * FROM [Sales$SheetLevelName];"
    ' Query based on a specific range address.
    'sSQL = "SELECT * FROM [Sales$A1:E89];"
    ' Query based on a book-level range name.
    sSQL = "SELECT * FROM BookLevelName;"

    Set rsData = New ADODB.Recordset
    rsData.Open sSQL, sConnect, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    ' Check to make sure we received data.
    If Not rsData.EOF Then
        Sheet1.Range("A1").CopyFromRecordset rsData
    Else
        MsgBox "No records returned.", vbCritical
    End If

    ' Clean up our Recordset object.
    rsData.Close
    Set rsData = Nothing

End Sub
```

By default, the OLE DB provider for Microsoft Jet assumes that the first row in the table you specify with your SQL statement contains the field names for the data. If this is the case, you can perform more complex SQL queries, making use of the `WHERE` and `ORDER BY` clauses. If the first row of your data table does not contain field names, however, you must inform the provider of this fact or you will lose the first row of data. The way to accomplish this is by providing an additional setting, `HDR=No`, to the `Extended Properties` argument of the connection string:

```
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
                "Data Source=C:\Files\Sales.xlsx;" & _
                "Extended Properties=""Excel 12.0;HDR=No"";"
```

Note that when you pass multiple settings to the `Extended Properties` argument, the entire setting string must be surrounded with double quotes and the individual settings must be delimited with semicolons. If your data table does not include column headers, you will be limited to `SELECT` queries.

Inserting and Updating Records in Microsoft Excel Workbooks

ADO can do more than just query data from an Excel workbook. You can also insert and update records in the workbook, just as you would with any other data source. Deleting records, however, is not supported. Updating records, although possible, is somewhat problematic when an Excel workbook is the data source, because Excel-based data tables rarely have anything that can be used as a primary key to uniquely identify a specific record. Therefore, you must specify the values of enough fields to uniquely identify the record concerned in the `WHERE` clause of your SQL statement when performing an update. If more than one record meets `WHERE` clause criteria, all such records will be updated.

Inserting is significantly less troublesome. All you do is construct a SQL statement that specifies values for each of the fields, and then execute it. Note once again that your data table must have column headers in order for it to be possible to execute action queries against it. The following example demonstrates how to insert a new record into the sales worksheet data table:

```
    Public Sub WorksheetInsert()

        Dim objConn As ADODB.Connection
        Dim sConnect As String
        Dim sSQL As String

        ' Create the connection string.
        sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
            "Data Source=C:\Files\Sales.xlsx;" & _
            "Extended Properties=Excel 12.0;"

        ' Create the SQL statement.
        sSQL = "INSERT INTO [Sales$] " & _
```

```
                "VALUES('VA', 'On-Line', 'Computers', 'Mid', 30);"

        ' Create and open the Connection object.
        Set objConn = New ADODB.Connection
        objConn.Open sConnect

        ' Execute the insert statement.
        objConn.Execute sSQL, , adCmdText Or adExecuteNoRecords

        ' Close and destroy the Connection object.
        objConn.Close
        Set objConn = Nothing

    End Sub
```

Note that if you use ADO to insert a new record into an Excel worksheet, and you use a range name as the table in the `INSERT` statement, that range name will not be extended to include the new record. Therefore, you should only use ADO to insert records into worksheets in situations where the worksheet name can be used as the table in the `INSERT` statement.

Querying Text Files

The last data access technique to discuss in this chapter is querying text files using ADO. The need to query text files doesn't come up as often as some of the other situations addressed in this chapter. However, when you're faced with an extremely large text file, the result of a mainframe database data dump, for example, ADO can be a lifesaver.

Not only will it allow you to rapidly load large amounts of data into Excel, but using the power of SQL to limit the size of the resultset can also enable you to work with data from a text file that is simply too large to be opened directly in Excel. For the discussion on text file data access, use a comma-delimited text file, `Sales.csv`, whose contents are identical to the `Sales.xlsx` workbook used in the Excel examples in the previous section. The following example demonstrates how to construct a connection string to access a text file:

```
    szConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
                "Data Source=C:\Files\;" & _
                "Extended Properties=Text;"
```

Note that in the case of text files, the `Data Source` argument is set to the directory that contains the text file. Do not include the name of the file in this argument. Once again, the provider is informed of the format to be queried by using the `Extended Properties` argument. In this case, you simply set this argument to the value `"Text"`.

Querying a text file is virtually identical to querying an Excel workbook. The main difference is how the table name is specified in the SQL statement. When querying a text file, the filename itself is used as the table name in the query. This has the added benefit of allowing you to work with multiple text files in a single directory without having to modify your connection string.

As with Excel, you are limited to `SELECT` queries if the first row of your text file does not contain field names. You must also add the `HDR=No` setting to the `Extended Properties` argument if this is the case, in order to avoid losing the first row of data. The example text file has field names in the first row, and you should assume that you need to limit the number of records you bring into Excel by adding a restriction in the form of a `WHERE` clause to your query. The following procedure demonstrates this:

```
Public Sub QueryTextFile()

        Dim rsData As ADODB.Recordset
    Dim sConnect As String
    Dim sSQL As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\;" & _
        "Extended Properties=Text;"

    ' Create the SQL statement.
    sSQL = "SELECT * FROM Sales.csv WHERE Type='Art';"

    Set rsData = New ADODB.Recordset
    rsData.Open sSQL, sConnect, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    ' Check to make sure we received data.
    If Not rsData.EOF Then
        ' Dump the returned data onto Sheet1.
        Sheet1.Range("A1").CopyFromRecordset rsData
    Else
        MsgBox "No records returned.", vbCritical
    End If

    ' Clean up our Recordset object.
    rsData.Close
    Set rsData = Nothing

End Sub
```

Summary

Many years ago you could concentrate on understanding Excel, and that would have been enough to get by. Today, however, it is becoming increasingly difficult to avoid coordinating your Excel activities with a back-end data source of some kind. This chapter provided you with a solid introduction to using SQL and ADO to access various data sources. Due to space constraints, this chapter could only scratch the surface of possibilities in each section. So if data access is or might become a significant part of your development effort, you are strongly recommended to obtain and read more specialized books on the subject.

The next chapter continues to look at managing external data with Excel by examining some of the built-in features provided by Excel for this purpose.

21

Managing External Data

Chapter 20 discussed how to access data with essentially unlimited flexibility using ADO. Excel also provides some built-in data management features, primarily through the `QueryTable`, `ListObject`, and `WorkbookConnection` objects. Excel's built-in data management features have less flexibility than custom ADO programming. For example, you can only use these features to retrieve data, not modify it. But they are simpler and offer a number of useful capabilities right out of the box. This chapter examines some of Excel's built-in data management capabilities.

The External Data User Interface

The built-in data management features in Excel 2007 are accessed from two groups on the Data tab of the Ribbon in the Excel 2007 user interface. The Get External Data and Manage Connections groups are shown in Figure 21-1.

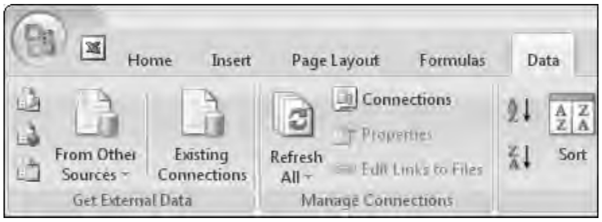
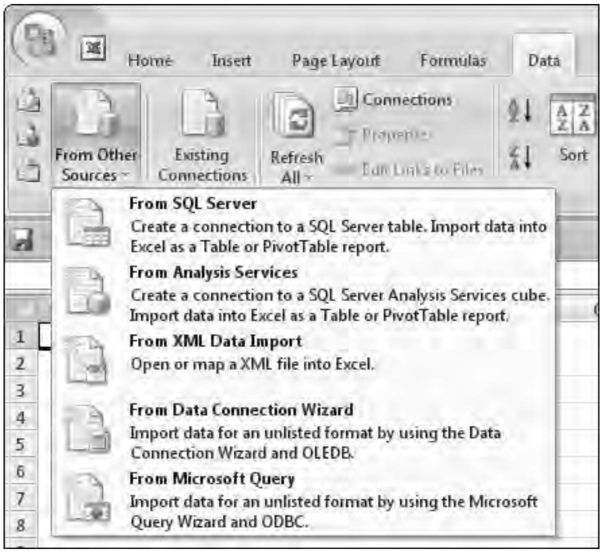


Figure 21-1

The controls on the Get External Data group are used to create new connections from your workbook to various data sources, and the controls on the Manage Connections group are used to manage data connections that already exist in your workbook.

Get External Data

The controls in the Get External Data group allow you to retrieve external data directly from various data sources or use predefined queries stored in various data connection files. The three buttons along the left side of the Get External Data group provide quick access for retrieving Microsoft Access data, data from the web, and data from text files, respectively. The From Other Sources button provides you with a drop-down list of all the other options available for retrieving data from external data sources, as shown in Figure 21-2.



From SQL Server
Create a connection to a SQL Server table. Import data into Excel as a Table or PivotTable report.

From Analysis Services
Create a connection to a SQL Server Analysis Services cube. Import data into Excel as a Table or PivotTable report.

From XML Data Import
Open or map a XML file into Excel.

From Data Connection Wizard
Import data for an unlisted format by using the Data Connection Wizard and OLEDB.

From Microsoft Query
Import data for an unlisted format by using the Microsoft Query Wizard and ODBC.

Figure 21-2

This chapter is concerned with the external data features that create a `QueryTable` object in the background. These features include Web Queries and all queries performed on data sources that return uncomplicated tabular data, including text files and relational databases like Access and SQL Server.

The Existing Connections button displays the Existing Connections dialog, shown in Figure 21-3. This dialog displays a list of connections that currently exist in the workbook, as well as a list of stored data connection files that you can use to retrieve external data.

The controls in the Get External Data group will be disabled if you have selected a cell within the range of an existing external data connection. If these controls are disabled, try selecting an unused cell outside of any existing data tables.

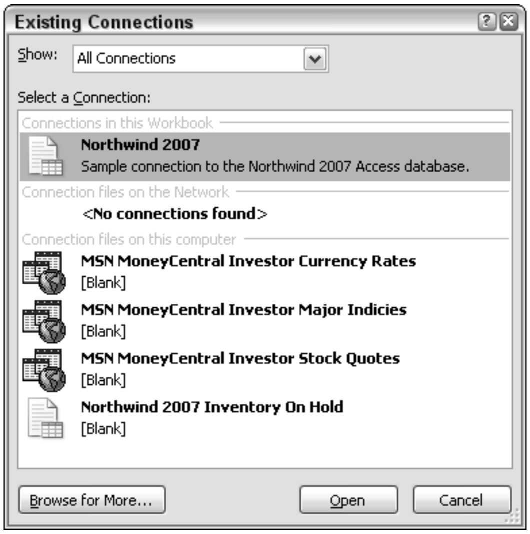

Existing Connections
Show: All Connections
Select a Connection:
Connections in this Workbook
Northwind 2007
Sample connection to the Northwind 2007 Access database.
Connection files on the Network
<No connections found>
Connection files on this computer
MSN MoneyCentral Investor Currency Rates
[Blank]
MSN MoneyCentral Investor Major Indicies
[Blank]
MSN MoneyCentral Investor Stock Quotes
[Blank]
Northwind 2007 Inventory On Hold
[Blank]
Browse for More...
Open
Cancel

Figure 21-3

Manage Connections

The controls on the Manage Connections group allow you to manipulate the individual connections in your workbook, as well as providing a central location for viewing and managing all connections in the currently active workbook. The buttons in this group that operate on a specific connection will be disabled until you select a cell within the range assigned to a connection. With a cell in a connection range selected, the Manage Connections group will look similar to Figure 21-4.

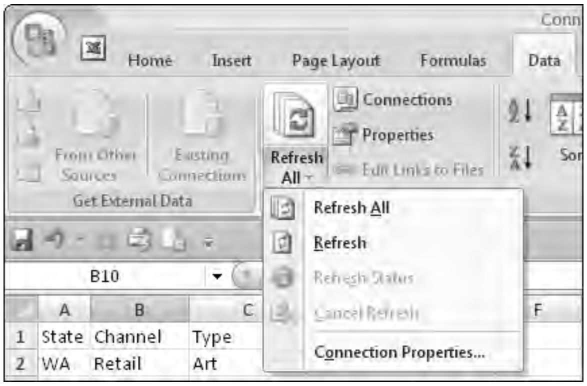
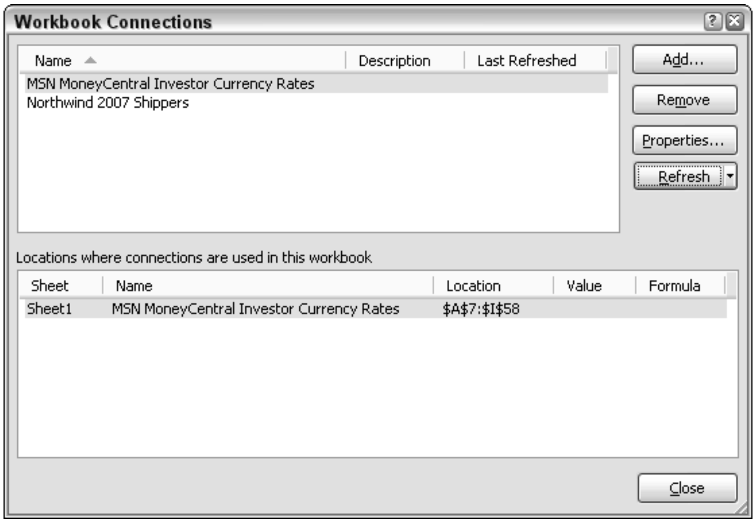


Figure 21-4

The most interesting control on the Manage Connections group is the new Connections button. This displays a dialog that lists all of the external data connections in the currently active workbook and allows you to see where they are used, as well as view and manipulate their properties. The Workbook Connections dialog is shown in Figure 21-5.



Name	Description	Last Refreshed
MSN MoneyCentral Investor Currency Rates		
Northwind 2007 Shippers		

Locations where connections are used in this workbook

Sheet	Name	Location	Value	Formula
Sheet1	MSN MoneyCentral Investor Currency Rates	$A$7:$I$58		

Figure 21-5

The Properties and Refresh buttons on this dialog expose exactly the same features as those shown on the Refresh All button in Figure 21-4.

The QueryTable and ListObject

When you use the Get External Data feature to create Web Queries or retrieve tabular data, you are creating a `QueryTable` to manage that data. This `QueryTable` can exist alone, or it can be associated with a `ListObject` (the `ListObject` is also covered in Chapter 6). Retrieving data using Web Queries or text files from the user interface will create a standalone `QueryTable`. Retrieving data from relational databases like Access or SQL Server will create a `ListObject` whose data source is a `QueryTable`. When you create your own `QueryTable` objects using VBA, you are free to create them either way. Both methods are demonstrated in this section.

This section utilizes the Northwind database, a sample database provided with Microsoft Access 2007. If you don't have this database available, you will need to install it to run the example code.

A QueryTable from a Relational Database

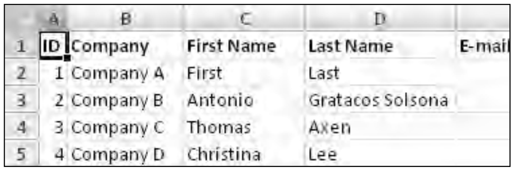
As a first introduction to the `QueryTable` object, create a simple QueryTable based on a table from the Access 2007 Northwind database used in Chapter 20. The code to create the `QueryTable` is as follows:

```
Sub CreateSimpleQueryTable()

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "OLEDB;Provider=Microsoft.ACE.OLEDB.12.0;" & _
                    "Data Source=C:\Files\Northwind 2007.accdb"
    Set rngDestination = Sheet1.Range("A1")

    ' Create the QueryTable.
    Set qryTable = Sheet1.QueryTables.Add(strConnection, rngDestination)

    ' Populate the QueryTable.
    qryTable.CommandText = "Customers"
    qryTable.CommandType = xlCmdTable
    qryTable.Refresh

End Sub
```

There are three steps involved in creating a `QueryTable`:

1. Define the connection string and destination range. If the connection string used here looks familiar, that's no coincidence. It's almost exactly the same connection string used to create an ADO connection to the Northwind 2007 Access database in Chapter 20. This is because both ADO and `QueryTables` use the same underlying OLE DB drivers to connect to relational databases. The only difference in this case is that the first item in the connection string must specify what type of connection string it is (in this case OLEDB). This is because a `QueryTable` can be based on several different connection types, so you need to tell it which type you're using. The destination range is a reference to the cell on the destination worksheet that you want to be the top-left cell in the resulting `QueryTable`.
2. Create the `QueryTable` object. This is a simple matter of calling the `QueryTables.Add` method, passing it the connection string and destination range defined in step 1, and assigning the result to a `QueryTable` object variable that will be used for further manipulation of the `QueryTable`.
3. Populate the `QueryTable` object. After step 2, the `QueryTable` exists, it knows what its data source is, and it knows where on the destination worksheet it will be located. However, it doesn't know what data to display. Remedy this by using the `CommandText` and `CommandType` properties of the `QueryTable` object to tell it to display the contents of the Customers table. Actual retrieval of data is accomplished by calling the `QueryTable.Refresh` method.

A section of the QueryTable created in the previous example is shown in Figure 21-6.



	A	B	C	D	
1	ID	Company	First Name	Last Name	E-mail
2	1	Company A	First	Last	
3	2	Company B	Antonio	Gratacos Solsona	
4	3	Company C	Thomas	Axen	
5	4	Company D	Christina	Lee	

Figure 21-6

A QueryTable that displays data from a relational database may need to be updated periodically — for example, in cases where other users may have added new records that need to be displayed. You can update the QueryTable data at any time by calling the QueryTable.Refresh method. The QueryTable will also perform this operation for you automatically, if you tell it to do so by adding the following line of code:

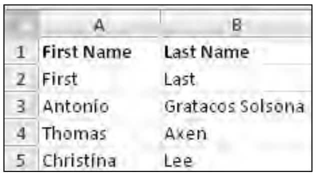
```
        ' Populate the QueryTable.
        qryTable.CommandText = "Customers"
        qryTable.CommandType = xlCmdTable
        qryTable.RefreshPeriod = 30
        qryTable.Refresh
```

This line of code tells the QueryTable to refresh itself automatically every 30 minutes, and it will continue to do so every 30 minutes for as long as the workbook is open. The value assigned to the QueryTable.RefreshPeriod property is persisted, so even after you close and reopen the workbook containing the QueryTable, it will continue to refresh automatically at the interval you've specified. The refresh period is specified in minutes, with valid values being 1 through 32,767. Setting the QueryTable.RefreshPeriod property to 0 disables automatic refreshing. This is also the default value if you don't specify this property.

You can think of a QueryTable as a container for whatever data you want to put in it. Just like you can refresh the QueryTable at any time to update it with the latest data from the data source, you can also change the data displayed by the QueryTable at any time by simply changing its CommandText and CommandType properties and calling its Refresh method:

```
        With Sheet1.QueryTables(1)
            .CommandText = "SELECT [First Name], [Last Name] FROM Customers"
            .CommandType = xlCmdSql
            .Refresh
        End With
```

This code changes the QueryTable from displaying the entire Customers table to deriving its data from a SQL statement that retrieves just the first and last names of each customer. See Chapter 20 for an introduction to SQL if you aren't familiar with it. As soon as this code is executed, the results displayed by the QueryTable will change accordingly, as shown in Figure 21-7.



	A	B
1	First Name	Last Name
2	First	Last
3	Antonio	Gratacos Solsona
4	Thomas	Axen
5	Christina	Lee

Figure 21-7

By default, a QueryTable will perform what is called a *background query*. This means that as soon as the query specified by the QueryTable has been submitted, VBA will return control of Excel back to the user. This may be a good choice if the QueryTable performs a long-running query and you don't want to require the user to sit and wait for it to finish. It may also be a bad choice if something critical in your application depends on the result of the query, and it therefore must be completed before your code continues.

In this case you may want to turn off background querying by setting the QueryTable.BackgroundQuery property to False:

```
        ' Populate the QueryTable.
        qryTable.CommandText = "Customers"
        qryTable.CommandType = xlCmdTable
        qryTable.RefreshPeriod = 30
        qryTable.BackgroundQuery = False
        qryTable.Refresh
```

The QueryTable.Refresh method also has a BackgroundQuery argument that can be set to False to accomplish the same thing without requiring an additional line of code. The difference is that the QueryTable.BackgroundQuery property is persistent and applies to all future refreshes, whereas the BackgroundQuery argument to the QueryTable.Refresh method must be specified each time you refresh the QueryTable or it will simply default to True.

A Query Table Associated with a ListObject

Standalone QueryTables are good for retrieving data that will be used for background or display purposes only. If you want the user to be able to interact with the data after it has been retrieved, a better option is to create a QueryTable associated with a ListObject. This creates a table in the Excel user interface with all of the built-in ease-of-manipulation features that users need to work with the data.

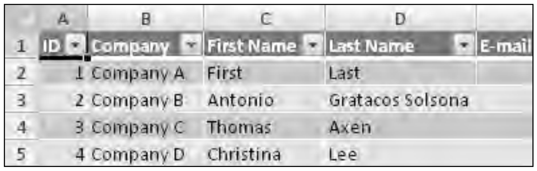
Note that there is some overlap between the QueryTable and ListObject properties and methods. For example, both the QueryTable and ListObject have a Refresh method that updates their data. When there is duplication, which object's property or method you decide to use is a matter of preference. Because this chapter focuses on QueryTables, the QueryTable properties and methods are used wherever there is duplication between the two object models.

The code for creating a QueryTable associated with a ListObject is very similar to the code for creating a QueryTable alone. In fact, the preceding QueryTable example can be modified to use a ListObject by changing a single line of code:

```
    Sub CreateQueryTableWithList()

        Dim qryTable As QueryTable
        Dim rngDestination As Range
        Dim strConnection As String

        ' Define the connection string and destination range.
        strConnection = "OLEDB;Provider=Microsoft.ACE.OLEDB.12.0;" & _
                        "Data Source=C:\Files\Northwind 2007.accdb"
        Set rngDestination = Sheet2.Range("A1")

        ' Create the ListObject and get a reference to its QueryTable.
        Set qryTable = Sheet2.ListObjects.Add(SourceType:=xlSrcExternal, _
            Source:=strConnection, Destination:=rngDestination).QueryTable

        ' Populate the QueryTable.
        qryTable.CommandText = "Customers"
        qryTable.CommandType = xlCmdTable
        qryTable.Refresh False

    End Sub
```

The process for creating a standalone `QueryTable` and the process for creating a `QueryTable` associated with a `ListObject` are fundamentally the same, the only difference being that you use the arguments of the `ListObjects.Add` method to specify the connection string and destination. Because a `ListObject` can have a number of additional data sources besides the external data used in this example, you also need to specify what type of data source your connection string represents, using the `SourceType` argument of the `ListObjects.Add` method. See Chapter 6 for more details on `ListObjects`.

A portion of the table created by the previous code is shown in Figure 21-8.



	A	B	C	D	
1	ID	Company	First Name	Last Name	E-mail
2	1	Company A	First	Last	
3	2	Company B	Antonio	Gratacos Solsona	
4	3	Company C	Thomas	Axen	
5	4	Company D	Christina	Lee	

Figure 21-8

Any time a `ListObject` is created from an external data source, either using VBA or through the Excel UI, an associated `QueryTable` object is created. You can use this `QueryTable` to manipulate the source data for any `ListObject` in a workbook.

Note that a `QueryTable` associated with a `ListObject` is not part of the `Worksheet.QueryTables` collection. It can only be accessed through the `ListObject.QueryTable` property of its associated `ListObject`. If you are using VBA to examine worksheets for the existence of query tables, you will need to look for them both directly in the `QueryTables` collection and indirectly in the `ListObjects` collection.

QueryTables and Parameter Queries

It is often useful to base your `QueryTable` on a parameter query rather than a fixed SQL statement. This allows you to determine which subset of the data you display, and even allows you to provide your users with the ability to modify the parameters when the `QueryTable` is refreshed.

One notable quirk of parameter queries is that `QueryTables` will not support them if you use the OLE DB provider used in the previous two sections. Instead, you must switch to the ODBC driver. This is a simple matter of changing the first argument of the connection string from OLEDB to ODBC and providing ODBC connection information:

```
    Sub CreateQueryTableWithParameters()

        Dim qryTable As QueryTable
        Dim rngDestination As Range
        Dim strConnection As String
        Dim strSQL As String

        ' Define the connection string and destination range.
        strConnection = "ODBC;DSN=MS Access Database;" & _
                        "DBQ=C:\Files\Northwind 2007.accdb;"
        Set rngDestination = Sheet3.Range("A1")

        ' Create a parameter query.
        strSQL = "SELECT [Product Name], [List Price], [Quantity Per Unit]" & _
                "  FROM Products" & _
                "  WHERE Category = ?;"   ' This is the parameter.

        ' Create the QueryTable.
        Set qryTable = Sheet3.QueryTables.Add(strConnection, rngDestination)

        ' Populate the QueryTable.
        qryTable.CommandText = strSQL
        qryTable.CommandType = xlCmdSql
        qryTable.Refresh False

    End Sub
```

In this example, you set the `CommandText` property of the `QueryTable` to a SQL statement that selects the Product Name, List Price, and Quantity Per Unit from the Products table of the Northwind 2007 database. The `WHERE` clause of the SQL statement contains a parameter. A parameter is created by placing a question mark character (?) where an actual value would normally go. A SQL statement may contain one or more parameters at any point where a value from the database would normally go. You cannot use parameters for table names, column names, or SQL keywords.

When this code is executed, VBA automatically recognizes that the SQL contains a parameter, and it will prompt you to enter a value for the parameter by displaying the dialog box shown in Figure 21-9.

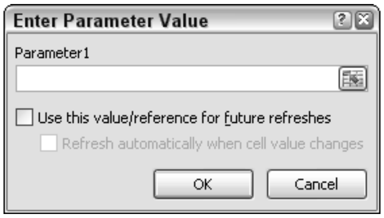

Enter Parameter Value
Parameter1
Use this value/reference for future refreshes
Refresh automatically when cell value changes
OK
Cancel

Figure 21-9

For each parameter in the `CommandText` property of the `QueryTable`, VBA creates a `Parameter` object that it adds to the `QueryTable.Parameters` collection. You can gain some additional control over how parameters are handled by creating these `Parameter` objects and adding them to the `Parameters` collection yourself.

You create parameters using the `QueryTable.Parameters.Add` method. This method adds a `Parameter` object to the `QueryTable` and returns a reference to it. You can then use that reference to modify the behavior of the `Parameter` object. The next example uses this capability to prepopulate the parameter with an initial value when the `QueryTable` is first created, and then to have it prompt the user for a new value each time the `QueryTable` is refreshed after that.

If you create your own `Parameter` objects, they must be created in the same order that the corresponding parameters appear in the SQL statement.

```
Sub CreateQueryTableWithParameters()

    Dim objParam As Parameter
    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String
    Dim strSQL As String

    ' Define the connection string and destination range.
    strConnection = "ODBC;DSN=MS Access Database;" & _
                    "DBQ=C:\Files\Northwind 2007.accdb;"
    Set rngDestination = Sheet3.Range("A1")

    ' Create a parameter query.
    strSQL = "SELECT [Product Name], [List Price], [Quantity Per Unit]" & _
            " FROM Products" & _
            " WHERE Category = ?;"  ' This is the parameter.

        ' Create the QueryTable.
    Set qryTable = Sheet3.QueryTables.Add(strConnection, rngDestination)

    ' Create the parameter and give it an initial value.
    Set objParam = qryTable.Parameters.Add("Select Category", _
            xlParamTypeVarChar)
    objParam.SetParam xlConstant, "Beverages"

    ' Populate the QueryTable.
    qryTable.CommandText = strSQL
    qryTable.CommandType = xlCmdSql
    qryTable.Refresh False

    ' Configure the parameter to prompt the user for a
    ' new value the next time the QueryTable is refreshed.
    objParam.SetParam xlPrompt, "Select Category"

End Sub
```

You can also configure a `Parameter` to retrieve its value from a cell on the worksheet, and to automatically refresh the `QueryTable` whenever the value in that cell changes. This allows you to, for example, provide the users with a data validation list of choices for the parameter, rather than forcing them to remember all the potentially valid values. This feature is also very useful when your query requires multiple parameters. Rather than having to contend with a prompt dialog for every parameter, the user can simply make the appropriate entries in the cells that specify the parameters.

If you replace the final line of code in the preceding procedure with the following two lines of code, the `Parameter` will retrieve its value from cell F1 and refresh the `QueryTable` automatically whenever the value in cell F1 changes:

```
    ' Configure the parameter to retrieve its value from cell F1
    ' and refresh the QueryTable whenever that value changes.
    objParam.SetParam xlRange, Sheet3.Range("F1")
    objParam.RefreshOnChange = True
```

In the previous examples, you used all three options provided by the `Parameter.SetParam` method. The first argument of the `SetParam` method specifies the option to be used, and the second argument provides additional data for that option. The three `SetParam` options are as follows:

- ❑ `xlConstant` — Tells the parameter to use the value specified by the second argument. The second argument can be anything that returns a value of the correct data type for the parameter, including a variable, a cell reference, or a hard-coded value.
- ❑ `xlPrompt` — Tells the parameter to prompt the user for its value. The second argument is the prompt string that will appear in the dialog box.
- ❑ `xlRange` — Tells the parameter to retrieve its value from the cell specified by the second argument. The second argument must be a valid `Range` object.

QueryTables from Web Queries

`QueryTables` are not limited to retrieving data from traditional databases. They can also retrieve data from web sites. Creating a `QueryTable` based on a web site is known as performing a Web Query. A simple example of a `QueryTable` based on a Web Query follows. It pulls in the most recent data on major U.S. financial indexes from the Wall Street Journal web site:

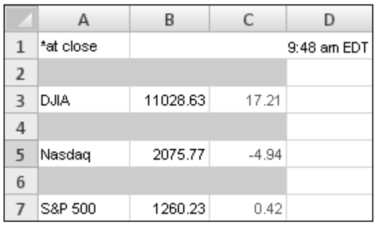
```
Sub CreateWebQuery()

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "URL;http://online.wsj.com/public/page/" & _
                    "markets.html?mod=hpp_us_indexes"
    Set rngDestination = Sheet4.Range("A1")

    ' Create the QueryTable.
    Set qryTable = Sheet4.QueryTables.Add(strConnection, rngDestination)

    ' Populate the QueryTable.
    qryTable.WebSelectionType = xlSpecifiedTables
    qryTable.WebTables = "19"
    qryTable.WebFormatting = xlWebFormattingAll
    qryTable.Refresh False

End Sub
```

Note the similarity between this code and the code used to create a `QueryTable` from a relational database. The differences are the content of the connection string and a different set of properties that must be set prior to refreshing the `QueryTable`.

You inform the `QueryTable` that it will be performing a Web Query by specifying `URL` as the first argument in the connection string. The rest of the connection string is a URL that specifies the web page from which you want to retrieve the data.

You could retrieve an entire web page with your Web Query, but this is rarely what you want to do. Most web pages are structured as a group of HTML tables nested and arranged to produce the visual layout you see in your browser. The Web Querying functionality of the `QueryTable` object lets you take advantage of this fact by allowing you to specify just the table or tables on the web site that you want to retrieve.

This is accomplished by setting the `WebSelectionType` property to `xlSpecifiedTables` and then listing the index numbers of the tables you want to retrieve in the `WebTables` property. If you want to retrieve more than one table, simply separate the list of index numbers with commas. There is no tried and true method for determining which table index number holds the data you want. The easiest way to determine this value is to start with code similar to that shown earlier, and increment the `WebTables` property beginning with the number 1 until you locate the index that corresponds to the data you want. A section of the result of the Web Query is shown in Figure 21-10.



	A	B	C	D
1	*at close			9:48 am EDT
2				
3	DJIA	11028.63	17.21	
4				
5	Nasdaq	2075.77	-4.94	
6				
7	S&P 500	1260.23	0.42	

Figure 21-10

Although the ability to specify a table index gives you significant flexibility in a Web Query, it also makes your code vulnerable to design changes on the target web site that might cause a change in the table index number of the table you want to retrieve.

You have three formatting options for the results returned by the Web Query, controlled by the value of the `WebFormatting` property:

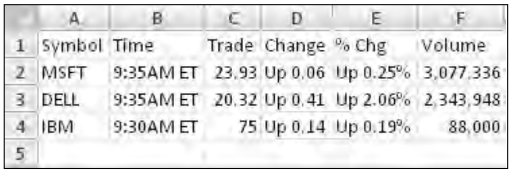
- `xlWebFormattingAll` — Imports the HTML format of the results exactly as they appear on the source web page, including functioning hyperlinks.
- `xlWebFormattingRTF` — Imports the data with the HTML formatting converted to rich-text format. This will produce results similar to `xlWebFormattingAll`, but without hyperlinks or merged cells.
- `xlWebFormattingNone` — Imports the data as plain text.

You will note the obvious thick gray bars separating the rows of data in Figure 21-10. These appear because that is how the table was structured on the source web page. You will often get artifacts like this when performing web queries. You can simply hide these rows and they will remain hidden, even when the `QueryTable` is refreshed. Just don't forget to unhide any rows and/or columns you've hidden if you do something that changes the structure of the Web Query such that the data you want to display ends up hidden.

Some web sites provide parameterized URLs specifically designed for use in web queries. The Yahoo Finance web site is one example. It provides a URL to which you can append a comma-delimited list of stock symbols to retrieve the information for. The following example creates a parameterized Web Query procedure that wraps this Yahoo Finance URL. Just call this procedure and pass it a comma-delimited list of stock ticker symbols, and it will return a table with the latest data for those symbols. You can pass as few or as many symbols as you like:

```
Sub ParameterizedWebQuery(strQuoteList As String, wksSheet As Worksheet)

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "URL;http://finance.yahoo.com/q/cq?d=v1&s=" & strQuoteList
    Set rngDestination = wksSheet.Range("A1")

    ' Create the QueryTable.
    Set qryTable = wksSheet.QueryTables.Add(strConnection, rngDestination)

    ' Populate the QueryTable.
    qryTable.WebSelectionType = xlSpecifiedTables
    qryTable.WebTables = "9"
    qryTable.WebFormatting = xlWebFormattingNone
    qryTable.Refresh False

End Sub

Sub CallParameterizedWebQuery()
    ParameterizedWebQuery "MSFT, DELL, IBM", Sheet6
End Sub
```

In this example, you call the parameterized Web Query procedure, pass it the stock symbols for Microsoft, Dell, and IBM, and tell it to place the resulting `QueryTable` on Sheet6. The results of this Web Query are shown in Figure 21-11.



	A	B	C	D	E	F
1	Symbol	Time	Trade	Change	% Chg	Volume
2	MSFT	9:35AM ET	23.93	Up 0.06	Up 0.25%	3,077,336
3	DELL	9:35AM ET	20.32	Up 0.41	Up 2.06%	2,343,948
4	IBM	9:30AM ET	75	Up 0.14	Up 0.19%	88,000
5						

Figure 21-11

In a real-world application, you would typically let the user specify the list of stock symbols — for example, by allowing them to pick the symbols from a list in a UserForm or allowing them to enter the symbols on a worksheet. You would then read the selected symbols, create a comma-delimited list from them, and pass this list to the `ParameterizedWebQuery` procedure.

There is no tried and true method for determining how to structure the URL used in a Web Query. If the format is not documented on the web site (and it rarely will be), you can use the macro recorder and experiment with the site directly in the web browser to determine the appropriate format for the Web Query URL. This is how both of the URLs in the previous examples were created.

Like all `QueryTables`, you can modify the data displayed by a `QueryTable` based on a Web Query at any time. Unlike `QueryTables` based on relational databases, however, you cannot modify the `CommandText` property of a `QueryTable` based on a Web Query. In fact, attempting to do so will render the `QueryTable` inoperable.

Instead, you change the data displayed in a `QueryTable` based on a Web Query by modifying its `Connection` property. The new connection string must be in exactly the same format as a connection string used to create a Web Query initially, and if the table index you want to retrieve is different from the current `WebTables` property value, you will have to update that property as well prior to refreshing the `QueryTable`.

A QueryTable from a Text File

You can also use a `QueryTable` to extract data from a text file. The advantage of using a `QueryTable` as opposed to the `Workbooks.OpenText` method is that the text file data can be loaded directly into the workbook you specify, as opposed to being opened in a new workbook.

This example uses the same `Sales.csv` source file used in the ADO text file example in Chapter 20. The code to load data from this text file using a `QueryTable` is as follows:

```
Sub QueryTableFromTextFile()

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "TEXT;C:\Files\Sales.csv"
    Set rngDestination = Sheet6.Range("A1")

    ' Create the QueryTable.
    Set qryTable = Sheet6.QueryTables.Add(strConnection, rngDestination)

    ' Populate the QueryTable.
    qryTable.TextFileStartRow = 1
    qryTable.TextFileParseType = xlDelimited
    qryTable.TextFileCommaDelimiter = True
```

```
        qryTable.TextFileTextQualifier = xlTextQualifierNone
        qryTable.TextFileColumnDataTypes = Array(2, 2, 2, 2, 1)
        qryTable.Refresh False

    End Sub
```

You inform the `QueryTable` that it will be extracting data from a text file by specifying `TEXT` as the first argument in the connection string. The second argument of the connection string is the full path and filename of the text file.

The `QueryTable` object has a series of text file-specific properties that allow you to control how the text file data is loaded. The following list describes the five of these properties that are most commonly used:

- `TextFileStartRow` — This property tells the query table which row of the text file to start with when it loads the data. A text file may have one or more initial rows that are not part of the data set. You can use this property to skip these initial rows by specifying some number greater than 1. A value of 1 tells the `QueryTable` to import the entire text file.
- `TextFileParseType` — This property tells the `QueryTable` whether the columns of data in the text file are separated by some delimiting character (`xlDelimited`) or are fixed width (`xlFixedWidth`). The value of this property will determine which additional properties you specify. Your text file is comma-delimited, so the options for a delimited text file are discussed next. If your text file has fixed-width columns, you would also set the `TextFileFixedColumnWidths` property to an array of column width values, one for each column in your text file.
- `TextFileCommaDelimiter` — When loading a delimited text file, you need to tell the `QueryTable` what delimiter to look for. You're loading a comma-delimited text file, so you set the `TextFileCommaDelimiter` property to `True`. The following additional properties are available to specify other delimiters: `TextFileTabDelimiter`, `TextFileSemicolonDelimiter`, `TextFileSpaceDelimiter`, and `TextFileOtherDelimiter`.
- `TextFileTextQualifier` — You will often encounter text files where values in text data type fields are surrounded by single or double quotes. This property is used to inform the `QueryTable` if this is the case, and if so, which of these characters it should treat as a qualifier and ignore when loading the data. The options are `xlTextQualifierSingleQuote`, `xlTextQualifierDoubleQuote`, and `xlTextQualifierNone`. You use the last option because your text file does not use any text field qualifiers.
- `TextFileColumnDataTypes` — This property is used to tell the `QueryTable` how to interpret each column of data in the text file. The property takes an array of integer values, one for each column in the text file, that specify what type of data is contained in those columns. For uncomplicated text and numeric data, you can normally pass the value 1 for every column. This tells the `QueryTable` to automatically determine what type of data is being loaded. If the text file contains data that may be misinterpreted, you can use this property to tell the `QueryTable` what type of data is contained in each column. A value of 2 means the column contains text data. A value of 9 tells the `QueryTable` to skip the column entirely. The additional seven allowable numeric values are used to handle various arrangements of date data types. See the VBA help for more details on these.

Creating and Using Connection Files

The information used to create query tables can be stored to a file on disk, called a connection file, and reused. Connection files provide several advantages when used in conjunction with `QueryTables`. They allow you to create a variety of query tables, store them to disk, and distribute them with your applications. Users can also use connection files directly by selecting them from the Existing Connections dialog on the Data tab of the Ribbon. Finally, if some detail of the data source changes — for example, the server name where a relational database is located, you can simply edit the connection file to reflect the new information and redistribute it. `QueryTables` based on the connection file will then be updated automatically with the new information.

Unfortunately, connection files do not present a very consistent story. There are more than half a dozen potential file types available, and most of them can be used for multiple types of source data. Each of the three data sources covered in this chapter (databases, web queries, and text files) requires a different type of connection file. This section focuses on Office Data Connect (.odc) and Web Query (.iqy) files. Office Data Connect files store connection information for databases, and Web Query files store connection information for web queries.

Connection files are simply text files with a file extension that identifies their type and contents, formatted according to what that type of connection requires. Connection files can be stored and used from any accessible directory on a computer, but connection files that you want to be visible automatically in the Get External Data ➪ Existing Connections list in the Excel user interface must be located in one of the following places:

- ❑ The `C:\...\My Documents\My Data Sources\` folder under the profile of the currently logged-in user.
- ❑ The `C:\Program Files\Common Files\ODBC\Data Sources\` folder.
- ❑ The `C:\Program Files\Microsoft Office\Office12\Queries\` folder.
- ❑ A custom location that has been defined by your network administrator through the use of Office policy settings.

Office Data Connect Files

An Office Data Connect (ODC) file contains XML data that specifies all of the information required to re-create an external connection to a database. See Chapter 12 for an introduction to XML if you aren't familiar with it. Although an ODC file contains XML data, it is not a well-formed XML file, so if you need to edit its contents, you may find it easier to use a text editor as opposed to an XML editor.

You can save the connection information for a `QueryTable` or `ListObject` to an ODC file by using the `SaveAsODC` method. The following line of code demonstrates how to save an ODC file that represents the first `QueryTable` created earlier in this chapter:

```
    Sheet1.QueryTables(1).SaveAsODC "C:\Files\CustomersTable.odc"
```

ODC files can only be saved from `QueryTables` of `ListObjects` that were created through OLEDB. Because utilizing parameter queries requires the use of ODBC, you cannot save an ODC file for a `QueryTable` or `ListObject` that utilizes a parameter query.

Upon opening the `CustomersTable.odc` file in a text editor, you will see a lot of content, most of which is beyond the scope of this chapter. All the data you would potentially need to edit is contained within the two XML elements located near the top of the file. An abbreviated version of these two elements is as follows:

```
<xml id=docprops>
 <o:DocumentProperties.....>
  <o:Name>Connection</o:Name>
 </o:DocumentProperties>
</xml>
<xml id=msodc>
 <odc:OfficeDataConnection.....>
  <odc:Connection odc:Type="OLEDB">
   <odc:ConnectionString>
    Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\Files\Northwind 2007.accdb;...
   </odc:ConnectionString>
   <odc:CommandType>Table</odc:CommandType>
   <odc:CommandText>Customers</odc:CommandText>
  </odc:Connection>
 </odc:OfficeDataConnection>
</xml>
```

The first XML element, called the `docprops` element, contains the name that will be displayed to the user in the Existing Connections dialog. This is automatically given the same name as the connection from which the ODC file was derived. Because you did not explicitly name your connection when you created it, VBA gave it the default name `Connection`. This is what you see stored in the `Name` element within the XML `docprops` element. You can simply edit this value to give the ODC file a more meaningful description.

The second XML element, called the `msodc` element, contains the connection string, command text, and command type used to create the `QueryTable`. The connection string that you see in the preceding example shows just the arguments you specified when you created the original `QueryTable`. If you open the ODC file yourself, you will see more than a dozen additional connection string arguments. These are simply the default values that OLEDB uses for any arguments you haven't specified.

Again, however, a connection string is simply plain text, so if you needed to modify the path to the database for all users of your application, you could simply type a new path into the `ConnectionString` element, save the file, and distribute it to everyone using your application. Any `QueryTables` built from the connection file would automatically update themselves with the new information.

To use the `CustomersTable.odc` file, you need to attach it to your `QueryTable` in the following manner:

```
    qryTable.SourceConnectionFile = "C:\Files\CustomersTable.odc"
    qryTable.RobustConnect = xlAlways
```

Be sure the `QueryTable` to which you are attaching the ODC file is the same `QueryTable` from which it was created. Once you have done this, the `QueryTable` will use the information contained in the specified ODC file to obtain its connection information whenever it needs to be refreshed.

It would be nice if you could simply create a new `QueryTable` using the ODC file as the data source in the first place. However, because the `Connection` argument of the `QueryTables.Add` method is required and does not accept an ODC file as its value, you must create the `QueryTable` initially in the manner shown in the previous section, and then attach the `QueryTable` to the ODC file afterward.

Web Query Files

Web Query (IQY) files are far simpler and easier to understand than ODC files. The process of generating an IQY file from an existing Web Query is a bit convoluted. There's no automated method of creating them like there is with ODC files. Creating an IQY file from an existing Web Query requires the following steps:

1. Open the Workbook Connections dialog using the Data ➪ Manage Connections ➪ Connections button.
2. In the Workbook Connections dialog, select the connection that corresponds to your Web Query and click the Properties button. This will display the Connection Properties dialog.
3. In the Connection Properties dialog, select the Definition tab and click the Edit Query button. This will display the Edit Web Query dialog.
4. You will see a toolbar across the top of the Edit Web Query dialog. The second button from the left on this toolbar is the Save Query button. Click this button and you will be prompted by a Save Workspace dialog to save your Web Query as an IQY file.

If you perform these steps on the connection for the Wall Street Journal Web Query created in the previous section, an IQY file with the following contents will be generated:

```
WEB
1
http://online.wsj.com/public/page/markets.html?mod=hpp_us_indexes

Selection=19
Formatting=All
PreFormattedTextToColumns=False
ConsecutiveDelimitersAsOne=False
SingleBlockTextImport=False
DisableDateRecognition=False
DisableRedirections=True
```

Only three entries in this file are important for your purposes. These are the `URL`, `Selection`, and `Formatting` entries. In fact, you could delete everything in the IQY file other than these three entries and the query would perform exactly as expected.

The URL is self-explanatory. It's the same URL used in the connection string when you originally created the Web Query. The `Selection` line specifies what the query should retrieve. This will be either the string value `EntirePage` or a number indicating the specific table you wish to retrieve. In this case the number 19 was generated, because that is the index number of the table specified when the Web Query was created.

The `Formatting` line specifies how the query should be formatted on the worksheet. Its potential values are `All`, `RTF`, or `None`, which correspond to the similarly named settings for the `WebFormatting` argument described in the previous section.

In addition to simplicity, IQY files have one additional advantage over ODC files for VBA programmers: you can create a new `QueryTable` directly from an IQY file using VBA. You could re-create the Wall Street Journal `QueryTable` example using the preceding IQY file in the following manner:

```
Sub CreateWebQueryFromIQY()

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "FINDER;C:\Files\Wall Street Journal Query.iqy"
    Set rngDestination = Sheet7.Range("A1")

    ' Create the QueryTable.
    Set qryTable = Sheet7.QueryTables.Add(strConnection, rngDestination)

    ' Populate the QueryTable.
    qryTable.Refresh False

End Sub
```

When creating a `QueryTable` from an IQY file, you pass `FINDER` as the first argument to the connection string and the full path and filename of the IQY file as the second argument. Notice that you don't need to set any additional properties of the `QueryTable` after creating it and prior to refreshing it. This is because all this information is contained in the IQY file.

The WorkbookConnection Object and the Connections Collection

New to Excel 2007 is an object and collection designed to manage all external data connections in a workbook. Each time you create any one of the built-in objects that Excel uses to manage external data, including `QueryTables`, `ListObjects`, and `PivotCaches`, you are also creating a new instance of a `WorkbookConnection` object. All of the `WorkbookConnection` objects in a given workbook are contained in the `Workbook.Connections` collection for that workbook.

You can also create a standalone `WorkbookConnection` object, one that is not associated with any external data container. The eventual intent for this feature is to allow you to create query tables, list objects, and all other external data containers using a `WorkbookConnection` object as the data source. However, as of Excel 2007, this feature has only been implemented for `PivotCache` objects. See Chapter 7 for more details on `PivotCache` objects.

You can create a new WorkbookConnection object using the Add or AddFromFile methods of the Workbook.Connections collection. The following example creates a new WorkbookConnection using the Add method:

```
Sub CreateNewConnection()

    Dim objWBConnect As WorkbookConnection

    Set objWBConnect = ThisWorkbook.Connections.Add( _
        Name:="New Connection", _
        Description:="My New Connection Demo", _
        ConnectionString:="OLEDB;Provider=Microsoft.ACE.OLEDB.12.0;" & _
                          "Data Source=C:\Files\Northwind 2007.accdb", _
        CommandText:="SELECT [First Name], [Last Name] FROM Customers", _
        lCmdtype:=xlCmdSql)

End Sub
```

After a WorkbookConnection object has been created, it is persisted when the workbook is saved. It will then be available any time the workbook is open, so there is no need to re-create it.

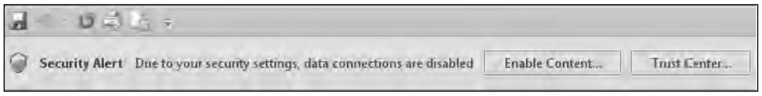
You can also use the Workbook.Connections collection to iterate through all the WorkbookConnection objects in a workbook and examine or modify their properties. The next example populates a worksheet with a list of all WorkbookConnection objects in the current workbook, and their type and their connection string if applicable:

```
Sub ExamineWorkbookConnections()

    Dim lOffset As Long
    Dim objWBConnect As WorkbookConnection

    Sheet8.UsedRange.Clear
    With Sheet8.Range("A1:C1")
        .Value = Array("Connection Name", "Connection Type", "Connection String")
        .EntireColumn.AutoFit
    End With

    For Each objWBConnect In ThisWorkbook.Connections
        lOffset = lOffset + 1
        Sheet8.Range("A1").Offset(lOffset, 0).Value = objWBConnect.Name
        Sheet8.Range("A1").Offset(lOffset, 1).Value = objWBConnect.Type
        If objWBConnect.Type = xlConnectionTypeODBC Then
            Sheet8.Range("A1").Offset(lOffset, 2).Value = _
                objWBConnect.ODBCConnection.Connection
        ElseIf objWBConnect.Type = xlConnectionTypeOLEDB Then
            Sheet8.Range("A1").Offset(lOffset, 2).Value = _
                objWBConnect.OLEDBConnection.Connection
        Else
            Sheet8.Range("A1").Offset(lOffset, 2).Value = "Not Applicable"
        End If
    Next objWBConnect

End Sub
```

Note that `WorkbookConnection` objects based on ODBC and OLE DB have additional child connection objects, the `ODBCConnection` object and `OLEDBConnection` object. These objects maintain the connection information required by ODBC and OLE DB.

External Data Security Settings

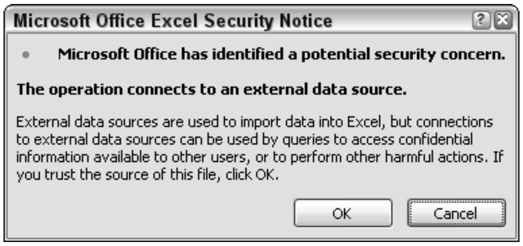
When you open an Excel 2007 workbook that contains connections to external data, you will encounter a security prompt like the one displayed in Figure 21-12.



Security Alert Due to your security settings, data connections are disabled

Figure 21-12

You can enable your external connections by clicking the Enable Content button and choosing the option to enable the content. If you don't do this, any automatic refreshing of your `QueryTables`, `ListObject`, or other data containers attached to external data will be disabled without further warning. If you attempt to manually refresh any of the data in your workbook, you will be prompted by another security warning, shown in Figure 21-13.



Microsoft Office Excel Security Notice

Microsoft Office has identified a potential security concern.

The operation connects to an external data source.

External data sources are used to import data into Excel, but connections to external data sources can be used by queries to access confidential information available to other users, or to perform other harmful actions. If you trust the source of this file, click OK.

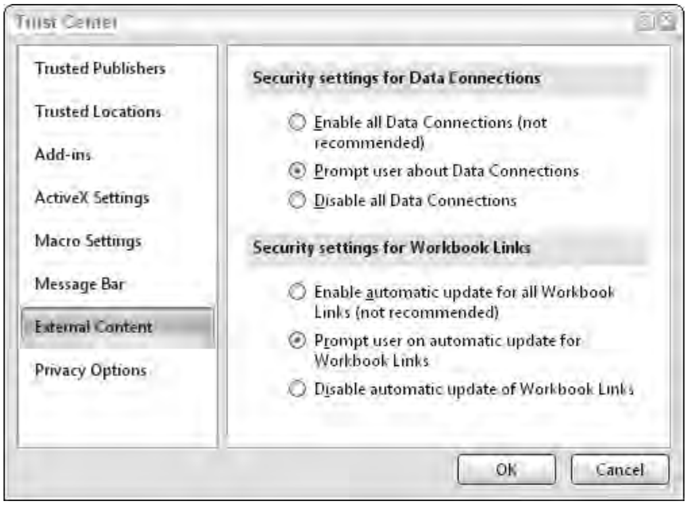
Figure 21-13

If you click OK in this dialog, all of your external data connections will be enabled. If you click Cancel, all of your external data connections will remain disabled and the refresh will not take place.

You can avoid these security issues by placing your file in a trusted location, or by selecting the Enable All Data Connections option in the Trust Center ➪ External Content section. Trusted locations are beyond the scope of this chapter, but you can enable all data connections in all workbooks you open in Excel 2007 using the following steps:

1. Click the Microsoft Office Button, and then click the Excel Options button. This will open the Excel Options dialog.
2. In the Excel Options dialog, select Trust Center from the list on the left side.
3. Next, click the Trust Center Settings button on the right side of the Excel Options dialog. This will open the Trust Center dialog.
4. In the Trust Center dialog, select External Content from the list on the left side.

You should now see at the dialog as it appears in Figure 21-14.



Security settings for Data Connections

- ○ Enable all Data Connections (not recommended)
- ◉ Prompt user about Data Connections
- ○ Disable all Data Connections

Security settings for Workbook Links

- ○ Enable automatic update for all Workbook Links (not recommended)
- ◉ Prompt user on automatic update for Workbook Links
- ○ Disable automatic update of Workbook Links

Figure 21-14

If you select the Enable all Data Connections option from the Security Settings for Data Connections section on the left side of the dialog, your external connections will be enabled automatically. Note that this setting can only be made manually; there's no way to use VBA to dynamically enable content on a user's computer. Also, if you're working on a corporate network, your network administrator may set security policies that do not allow you to make this change at all.

Summary

This chapter examined the built-in features Excel provides for dealing with external data, as exposed through the `QueryTable` object. It showed how to use query tables to retrieve data from relational databases, web sites, and text files. The chapter examined how connection files allow you to persist `QueryTable` connection information to an easily modified and distributed text file, and you took a brief look at the two new objects used to manage connection information in Excel 2007: the `Workbook.Connections` collection and the `WorkbookConnection` object.