3

MARS, Asynchronous Commands, and ObjectSpaces

THE DATA ACCESS CLASSES introduced into ADO.NET version 1.0 and the techniques for using them are aimed at the needs of modern applications—in particular the two distinct requirements that these applications encounter. The first is to be able to operate in a disconnected environment, where data needs to be remoted to clients across a network. The second is to provide fast, simple, and clean access to data in read-only fashion for populating controls and other data consumers.

However, there will always be many more specific scenarios in which your code will operate. To provide more opportunities for extracting better performance from your applications and to add new features that have been requested by users, ADO.NET version 2.0 supports some new approaches to accessing and updating data. The features we'll be covering in this chapter include the following:

- Multiple Active Results Sets (MARS), which provide the opportunity to open more than one results set over the same connection and access them concurrently
- Asynchronous access to the results of multiple commands over the same `Connection` instance, plus the ability to open connections asynchronously

- The new ObjectSpaces technology added to ADO.NET, which allows data to be defined as instances of a class but handled just like other intrinsic data types

The first two of these topics are related in that the ability to execute multiple commands asynchronously and to have more than one set of results active over the same connection provides huge opportunities for increasing performance when complex multiquery processes are required. We start this chapter with an overview of these two technologies and how they fit in with the existing data access capabilities of ADO.NET.

Expanded and Improved Data Access Features

To understand what the topics covered in this chapter are really about, how they fit together, and what kinds of problems they are designed to solve, you need to think about how existing data access methods are used when all the data an application needs resides in a single database and when it is in more than one database. These issues are becoming more common as applications have to work with multiple sources of data.

While the existing techniques from version 1.x of ADO.NET can easily accomplish the required tasks, both ease of access and performance can be improved. In particular, when the data sources are not local (as is becoming increasingly common in modern client-side applications and multiple-server Web farms), the procedural synchronous nature of current data access techniques can cause processing bottlenecks. You have to consider the individual steps that the code must go through to access the data when more than one query to the database is required.

Accessing a Single Data Source

In the simplest case, where data comes from a single server, the same connection can be used for all the commands that need to be executed to read or update data. Taking a `DataSet` as an example, the commands would be those required to fetch data from the database (the `SelectCommand`) and to update the data (the `InsertCommand`, `UpdateCommand`, and `DeleteCommand`). Figure 3.1 shows a simple scenario.

This process supports only synchronous access to the data. You can't read data through the connection while concurrently updating rows, and you can't execute concurrent insert, delete, and update operations. When the `Fill` method is called, no other access is possible through the connection until the method completes. And the `Update` operation automatically

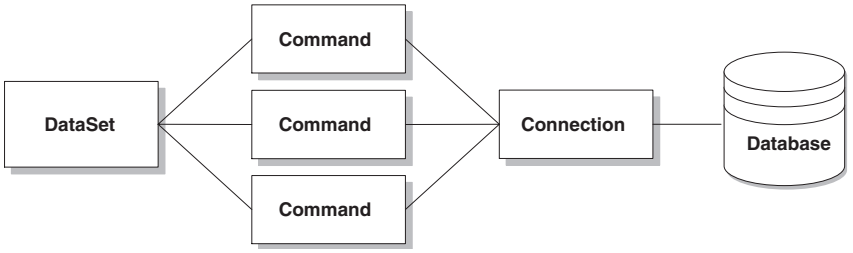

FIGURE 3.1. Multiple commands to the same database through a single connection

executes the commands sequentially when pushing updates back into the database. In most cases, however, this behavior is perfectly satisfactory and matches the requirements of the application.

Executing Multiple Commands Sequentially

An alternative situation is when you use several object instances to access data (here multiple `DataReader` instances, but it could be a mix of `DataReader` and `DataSet` instances) over the same connection. Again, only one of these can be executing a query over the single connection at a time. But because the ADO.NET code you write will usually call methods on the `DataReader` or other objects sequentially and will block on each method call until it is complete, there is no problem with using the same connection. Figure 3.2 shows this scenario.

In this case, however, there are two areas where performance and usability issues can arise:

1. If you open a rowset with a `DataReader` over the connection, you must close it before you attempt to open another `DataReader` or execute another command—if not, you'll get a "`Connection is busy . . .`"

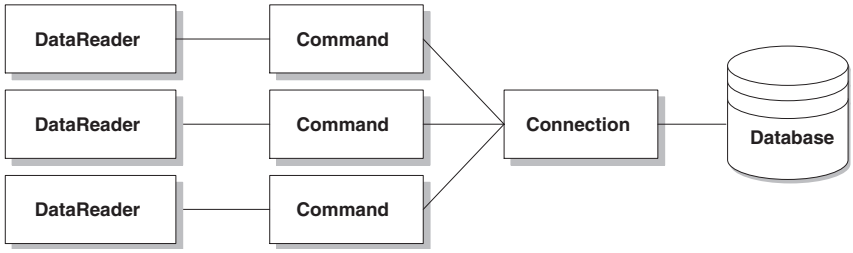

FIGURE 3.2. Multiple commands for separate DataReader instances through a single connection

error. To be able to open more than one rowset concurrently in ADO.NET version 1.x requires that each command have its own separate connection to the database. Because the number of database connections available is limited and they are expensive in terms of resource usage, this often isn't a feasible approach.

2. If one of the commands you're executing takes a long time to return results, the code in the application will block and wait until the query process is complete before it can execute the next command. So your code may be standing idle when it could be doing something else in the meantime.

The Requirements for Asynchronous Execution

It would be nice to be able to start the three commands together, allowing them to execute asynchronously and then handling the results as they are returned. That way the data or the results can be processed and/or displayed as they are received—making the applications appear more responsive and usable. But that requires two changes to the existing classes to make it work, and (not coincidentally) these are two of the new features in ADO.NET that we're covering in this chapter.

1. The MARS feature in ADO.NET 2.0 allows you to have more than one rowset or command open over a single connection. In other words, you can open a rowset with a `DataReader` and then send other commands to the database over the same connection while the `DataReader` is open. You can even have more than one rowset open in separate `DataReader` instances at the same time and read from either one as required.
2. The introduction of asynchronous processing in ASP.NET 2.0 for the `Command` class allows you to start more than one command executing concurrently and then wait until one or all of them have completed. You can also open connections asynchronously, though only a single connection request can be outstanding at any one time.

The combination of the two techniques means that, using only a single database connection, you can execute multiple commands concurrently and have the rowsets they return open at the same time. And, of course, update commands can also be executed over the same connection at the same time, even when rowsets are open on that connection. This provides a huge opportunity for increased performance and responsiveness for

your applications and Web pages. However, bear in mind that there are scenarios, such as when working with multiple separate data sources, when MARS is not directly useful.

Accessing Multiple Data Sources

When it comes to accessing multiple different data sources in an application, you have no option but to use separate connections. Of course, a connection can be to only a single database. So, for multiple data access operations against separate databases, you end up with something like the situation shown in Figure 3.3.

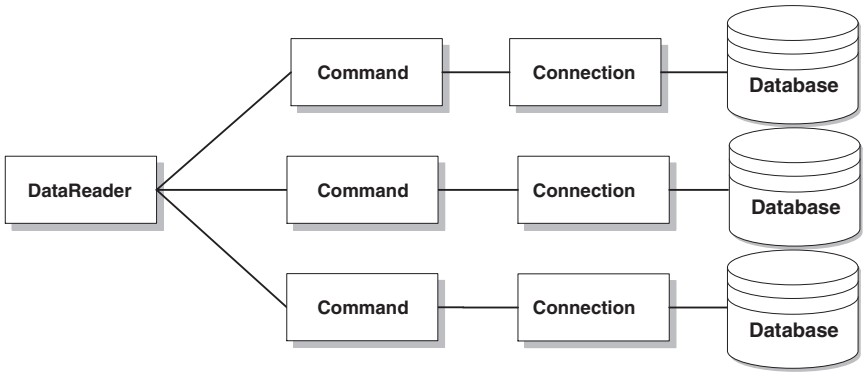

FIGURE 3.3. Multiple commands to different databases through separate connections

The MARS feature may not provide any advantage in this situation because each command will use its own dedicated connection. However, it may be that you have some hybrid scenario, where some commands are to the same database (in which case they can share a connection) while others are to a different database through a different connection (see Figure 3.4).

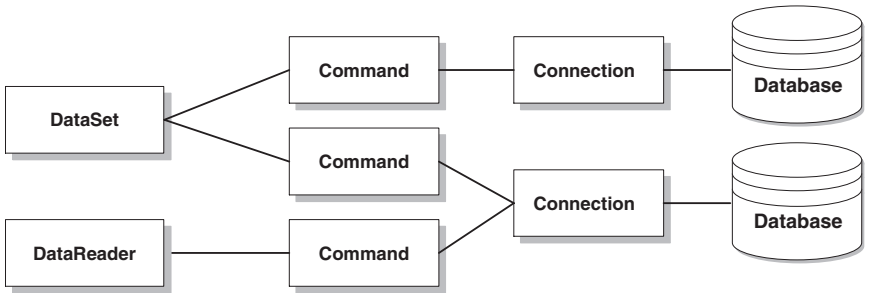

FIGURE 3.4. A mix of commands and connections for different databases

Here, MARS can provide the opportunity for improved access, for example, being able to open a rowset with a `DataReader` and then execute commands against the same database while the `DataReader` is open. But what is also useful, in both of these scenarios (Figures 3.3 and 3.4), is the ability to execute commands asynchronously. For example, you can start execution of the command that fetches the rows for the `DataReader`, then—while it is running—execute the `Fill` method for the `DataSet`.

Opening Connections Asynchronously

In ADO.NET 2.0 it's now also possible to open connections asynchronously. This offers even more opportunity to increase efficiency, especially when access to different data sources is required. It means that your code can continue to execute while waiting for a connection to be established—particularly useful when connecting to a remote data source rather than a local one.

When multiple connections are required, a common scenario is for the code to open these one by one. In fact, there's no alternative to this in ADO.NET 1.x. Being able to open each one asynchronously means that you can start to retrieve or update the data over that connection while waiting for other connections to open. You may even end up actually using fewer connections because you may be able to close some and return them to the connection pool before the last one has opened. Running the same code synchronously would mean holding all of the connections open until all of the processing completes.

Of course, you can open the connection asynchronously in situations where you need only one connection, but the advantage is not as obvious unless you start the opening process at the beginning of the code and then do some other work while you wait for it to open. The danger here is that you may hold on to the connection for longer than necessary, robbing other applications (and other instances of this application) of available connections and producing no overall advantage.

MARS in Action

ADO.NET 2.0 provides a great new feature called MARS. We discussed how useful this is in the first sections of this chapter, and here you'll see an example of it in action.

Listing 3.1 demonstrates how useful MARS can be when dealing with related tables (though this is not, of course, the only way it can be used). The sample `Northwind` database we're using here contains an `Orders` table

In future releases (following the Technology Preview release) you won't have to do anything to enable MARS. As long as you have version 9.0 of the Microsoft Data Access Components (MDAC) library on the client machine, MARS will just work transparently. However, for the Technology Preview version, **you must include in the connection string for your database the token** `"use mdac9=true;"`. For Windows Forms (compiled) applications, see the topic Using New ADO.NET Features That Require MDAC 9.0 in the Overview of ADO.NET section of the SDK for details of how to create an application manifest that specifies MDAC 9.0.

and a related Order Details table. The example fetches the first five rows from the Orders table, displays a couple of column values, and then—while this rowset is open over the connection—executes another command that returns a rowset of the related Order Details rows for the current order.

The code starts in the usual way by creating a single `Connection` and then two `Command` instances. The first `Command` selects the first five rows from the Orders table. The second selects the details of each line on an order, using a parameter named `@orderid` that will be set to the ID of the current order when this statement is executed. A parameter that corresponds to the `@orderid` parameter in the SQL statement is then added to the `Command` object's `Parameters` collection.

Listing 3.1. An Example of MARS in Action

```
' get connection string and specify SQL statements
Dim sConnectString As String = "your-connection-string"
Dim sOrderSQL As String = "SELECT TOP 5 OrderID, ShipName FROM Orders"
Dim sOrderLineSQL As String = "SELECT OrderID, Quantity, " _
           & "[Order Details].UnitPrice As Price, ProductName " _
           & "FROM [Order Details] JOIN Products " _
           & "ON [Order Details].ProductID = Products.ProductID " _
           & "WHERE OrderID = @orderid"

' create connection and two commands
Dim oConn As New SqlConnection(sConnectString)
Dim oCmdOrder As New SqlCommand(sOrderSQL, oConn)
Dim oCmdOrderLine As New SqlCommand(sOrderLineSQL, oConn)

' add parameter to second command to use for selecting order
oCmdOrderLine.Parameters.Add("@orderid", SqlDbType.Char, 5)
...
```

Fetching the Orders Rows

In Listing 3.2 the code continues by declaring a couple of `DataReader` variables and executing the command to get the Orders rowset. The `While` construct then starts by reading the first row and displaying the order ID and customer name.

Listing 3.2. Executing the Command to Get the Orders Rowset

```
...
' declare two DataReader variables and open connection
Dim oDROrder, oDROrderLine As SqlDataReader
oConn.Open()

' get rowset from Orders table
oDROrder = oCmdOrder.ExecuteReader(CommandBehavior.CloseConnection)

' iterate through orders
While oDROrder.Read()

  ' display order ID and name
  lblResult.Text &= oDROrder("OrderID").ToString() & " " _
                 & oDROrder("ShipName") & "<br />"
  ...
```

Fetching the Order Details Rows

Having discovered the ID of the current order, the code in Listing 3.3 now sets the value of the `@orderid` parameter and executes the second command to fetch the matching rows from the Order Details table. It iterates through this rowset, displaying the product name and quantity, and then closes this `DataReader`. However, the first `DataReader` is still open against the Orders table, so when the `While` loop ends, the code goes back and reads the next Orders row, then repeats the process.

Listing 3.3. Executing the Command to Get the Order Details Rowset

```
  ...
  ' set parameter for second command to value of order ID
  oCmdOrderLine.Parameters("@orderid").Value = oDROrder("OrderID")

  ' get list of lines for this order from Order Details table
  oDROrderLine = oCmdOrderLine.ExecuteReader()

  ' iterate through order lines displaying details
  While oDROrderLine.Read()
    lblResult.Text &= oDROrderLine("Quantity") & " " _
                   & oDROrderLine("ProductName") & " at " _
```

```
                    & oDROrderLine("Price") & "<br />"
  End While

  ' close DataReader for Order Details table
  oDROrderLine.Close()
  lblResult.Text &= "<br />"

End While

' close DataReader for Orders table
oDROrder.Close()
```

Once all the Orders rows have been read and the related Order Details rows for each one fetched and displayed, this `DataReader` is closed. Because the `CloseConnection` option was specified when this `DataReader` was opened, the connection to the database is closed automatically at this point.

Asynchronous Command Execution

When accessing data in a data store such as a database using ADO or ADO.NET 1.x, each query is executed in turn, and the code waits for each one to complete before processing the next one. This allows the same connection to be used if the queries access the same data store or a different connection if they access different data stores.

> In future releases (following the Technology Preview release) you won't have to do anything to enable the asynchronous command execution feature. As long as you have version 9.0 of the MDAC library on the client machine, asynchronous command execution will just work transparently. However, for the Technology Preview version, **you must include in the connection string for your database the token** `"use mdac9=true;"`. For Windows Forms (compiled) applications, see the topic Using New ADO.NET Features That Require MDAC 9.0 in the Overview of ADO.NET section of the SDK for details of how to create an application manifest that specifies MDAC 9.0.

The previous section of this chapter demonstrates how MARS can be used to enable a single connection to handle multiple sets of results from the same data store, but this still doesn't solve the issue of executing more

than one command concurrently. Neither does it help when you want to access different data stores concurrently.

However, ADO.NET 2.0 now includes features that allow multiple commands to execute concurrently and asynchronously. This means, in essence, that code can create and start execution of several commands without having to wait for them to complete individually. However, unless you use the MARS feature described earlier in this chapter, you must provide a separate connection for each one.

The asynchronous model adopted in ADO.NET 2.0 aims to give maximum flexibility when using this feature by implementing three distinct techniques:

1. **The polling model**, where you start all the processes and then repeatedly test each one for completion. Other tasks can be accomplished between each test.
2. **The callback model**, where you specify a routine that will be executed when each process is complete.
3. **The wait model**, where a wait handle is created for each method that starts the command execution. This allows control of the timeout for each process, and the code can sleep while waiting for a result to be returned or for a process to time out.

It's also possible to open connections to a data source asynchronously, allowing code to continue with other work while waiting for the connection to be made. The process is much the same as executing a command, and we look at this topic later in the chapter. But before we examine the actual code required to perform asynchronous command execution, the next section lists the classes, properties, and methods used. At present, this feature is implemented only for the classes in the `SqlClient` namespace.

Asynchronous Classes, Methods, and Properties

To a greater extent, all three asynchronous execution techniques use the same methods that are added to the `SqlCommand` class in ADO.NET 2.0 to start a process and then capture the results. These methods are listed in Table 3.1.

TABLE 3.1. Asynchronous Execution Methods of the SqlCommand Class

Method	Description
`BeginExecuteNonQuery`	Starts an asynchronous query to the data source that is not expected to return any rows. The return value is a reference to an object that implements the `IAsyncResult` interface, in this case an instance of the `SqlAsyncResult` class, which is used to monitor and access the process as it runs and when it is complete. There are two overloads for this method:
async-result = command`.BeginExecuteNonQuery()`	Starts execution of this command and returns a reference to a `SqlAsyncResult` instance for this process.
async-result = command`.BeginExecuteNonQuery(`*callback, state*`)`	As above, but takes an `AsyncCallback` instance that specifies the callback routine to be executed when the process is complete, plus an `Object` that defines state information for the process.
`EndExecuteNonQuery`	Once the command execution started by a `BeginExecuteNonQuery` call has completed, this method is called to access the results. The single parameter is a reference to the `SqlAsyncResult` for the command, and the method returns an `Integer` that is the number of rows affected (in the same way the `ExecuteNonQuery` method does for synchronous processes).
rows-affected = command`.EndExecuteNonQuery(`*async-result*`)`	
`BeginExecuteReader`	Starts an asynchronous query to the data source that is expected to return some rows. The return value is a reference to an object that implements the `IAsyncResult` interface, in this case an instance of the `SqlAsyncResult` class, which is used to monitor and access the process as it runs and when it is complete. There are four overloads for this method:
async-result = command`.BeginExecuteReader()`	Starts execution of this command and returns a reference to a `SqlAsyncResult` instance for this process.
async-result = command`.BeginExecuteReader(`*command-behavior*`)`	Starts execution of this command using the specified command behavior value, such as `CloseConnection`, and returns a reference to a `SqlAsyncResult` instance for this process.
async-result = command`.BeginExecuteReader(`*callback, state*`)`	As above, but takes an `AsyncCallback` instance that specifies the callback routine to be executed when the process is complete, plus an `Object` that defines state information for the process.

continues

TABLE 3.1. Asynchronous Execution Methods of the SqlCommand Class (continued)

Method	Description
async-result = *command*.BeginExecuteReader(*callback, state, command-behavior*)	As above, but using the specified command behavior value, such as CloseConnection.
EndExecuteReader	Once the command execution started by a BeginExecuteReader call has completed, this method is called to access the results. The single parameter is a reference to the SqlAsyncResult for the command, and the method returns a DataReader that references the rowset(s) returned by the query (in the same way the ExecuteReader method does for synchronous processes). The single overload is:
data-reader = *command*.EndExecuteReader(*async-result*)	
BeginExecuteXmlReader	Starts an asynchronous query to the data source that is expected to return some rows as XML elements. The return value is a reference to an object that implements the IAsyncResult interface, in this case an instance of the SqlAsyncResult class, which is used to monitor and access the process as it runs and when it is complete. There are two overloads for this method:
async-result = *command*.BeginExecuteXmlReader()	Starts execution of this command and returns a reference to a SqlAsyncResult instance for this process.
async-result = *command*.BeginExecuteXmlReader(*callback, state*)	As above, but takes an AsyncCallback instance that specifies the callback routine to be executed when the process is complete, plus an Object that defines state information for the process.
EndExecuteXmlReader	Once the command execution started by a BeginExecuteXmlReader call has completed, this method is called to access the results. The single parameter is a reference to the SqlAsyncResult for the command, and the method returns an XmlReader that references the XML data returned by the query (in the same way the ExecuteXmlReader method does for synchronous processes). The single overload is:
xml-reader = *command*.EndExecuteXmlReader(*async-result*)	

The IAsyncResult Interface and SqlAsyncResult Class

All of the methods for the SqlCommand class listed in the previous section that start an asynchronous process return a reference to an object that exposes the IAsyncResult interface. The specific class used for asynchronous command execution is SqlAsyncResult, and the properties of this class are shown in Table 3.2.

TABLE 3.2. Asynchronous Execution Properties of the SqlCommand Class

Property	Description
`AsyncState`	Returns an `Object` that describes the state of the process. Read-only.
`AsyncWaitHandle`	Returns a `WaitHandle` instance that can be used to set the timeout, force code to wait for completion, and test when and how the process completed. Read-only.
`CompletedSynchronously`	Returns a `Boolean` value indicating whether the process was executed synchronously (`True`) or asynchronously (`False`). Read-only.
`IsCompleted`	Returns a `Boolean` value indicating whether the process is complete (`True`) or is still executing (`False`). Read-only.

The AsyncCallback Delegate

The `AsyncCallback` delegate is a system class that is not specifically allied to ADO.NET but is used by many objects within the Framework. It is used to specify the callback routine or function that will be executed when the asynchronous process completes. The only member of this class that's important here is the constructor, shown in Table 3.3, which is used to generate an instance of the class to pass to the `BeginExecuteNonQuery`, `BeginExecuteReader`, and `BeginExecuteXmlReader` methods of the `Command`.

TABLE 3.3. The AsyncCallback Constructor

Constructor	Description
`AsyncCallback(`*`callback-handler`*`)`	Creates a new `AsyncCallback` instance that defines the name of the method, function, or routine that will be called when the process is complete.

See the section on the asynchronous callback model later in this chapter for more details.

The WaitHandle Class

The `WaitHandle` class is an abstract system class used for many purposes within the Framework. By calling the methods of the `WaitHandle`, you

effectively tell the code to wait for the specified process to complete. If there are multiple processes involved, you can create an array of `WaitHandle` instances and then tell the code to "sleep" until any one of them completes. The `WaitHandle` class exposes a `Static` property, shown in Table 3.4, for the timeout that will be allocated to the process.

There are also three methods that allow you to start the wait process and one to close and release the handle afterwards, as shown in Table 3.5.

TABLE 3.4. The Properties of the WaitHandle Class

Property	Description
`WaitTimeout`	An `Integer` value that is the timeout for the process in milliseconds.

TABLE 3.5. The Methods of the WaitHandle Class

Method	Description
`WaitOne`	Waits for a single process to complete or time out. Returns a `Boolean` value that is `True` if the process completed or `False` if it timed out. There are three overloads of this method:
`completed = WaitHandle``.WaitOne()`	Starts the wait process using the default value for the timeout.
`completed = wait-handle``.WaitOne(`*`milliseconds, exit-context`*`)`	Starts the wait process, with the timeout set to the number of milliseconds specified as an `Integer`. The `exit-context` parameter specifies whether the method will reacquire a synchronized context and should be set to `False` for the ADO.NET asynchronous execution.
`completed = WaitHandle``.WaitOne(`*`time-span, exit-context`*`)`	Starts the wait process, with the timeout specified using a `TimeSpan` instance that describes the timeout in terms of the time unit and value. The *`exit-context`* parameter is as described above.
`WaitAny`	A `Static` method that waits for any one of the processes in an array of `WaitHandle` instances to complete or time out. Returns an `Integer` value that is the index within the array of the first process that completed. If any of the processes times out, the method returns the value of the timeout instead of the handle index. There are three overloads of this method:

TABLE 3.5. The Methods of the WaitHandle Class (continued)

Method	Description
index = WaitHandle.WaitAny(*wait-handle-array*)	Starts the wait process for any of the WaitHandles, using the default value for the timeout.
index = WaitHandle.WaitAny(*wait-handle-array, milliseconds, exit-context*)	Starts the wait process for any of the WaitHandles, with the timeout in milliseconds and the *exit-context* parameter as described for the WaitOne method.
index = WaitHandle.WaitAny(*wait-handle-array, time-span, exit-context*)	Starts the wait process for any of the WaitHandles, with the timeout specified as a TimeSpan instance and the exit-context parameter as described for the WaitOne method.
WaitAll	A Static method that waits for all of the processes in an array of WaitHandle instances to complete or time out. Returns a Boolean value that is True if all the processes completed or False if any one of them timed out. There are three overloads of this method:
completed = WaitHandle.WaitAll(*wait-handle-array*)	Starts the wait process for all the WaitHandles, using the default value for the timeout.
completed = WaitHandle.WaitAll(*wait-handle-array, milliseconds, exit-context*)	Starts the wait process for all the WaitHandles, with the timeout in milliseconds and the *exit-context* parameter as described for the WaitOne method.
completed = WaitHandle.WaitAll(*wait-handle-array, time-span, exit-context*)	Starts the wait process for all the WaitHandles, with the timeout specified as a TimeSpan instance and the *exit-context* parameter as described for the WaitOne method.
Close	Closes and releases the WaitHandle instance and any resources it is using.

See the section on the asynchronous wait model later in this chapter for more details.

The Asynchronous Polling Model

The simplest approach to handling asynchronous execution of one or more commands is through the polling model. It simply involves starting off the process (in Listing 3.4 this is a SQL `UPDATE` statement, but it could be any statement or stored procedure that does not return rows) and then repeatedly checking the `IsCompleted` property of the `SqlAsyncResult` instance until it returns `True`.

Of course, the code can go off and do other things between checking to see whether the process is complete. However, this approach is not recommended unless it is a simple and "tight" loop that handles a specific required task and uses only minimal processing time. If there are large or unrelated tasks to accomplish, you should consider using the callback or wait models instead.

Listing 3.4. An Example of the Asynchronous Polling Model

```
' create connection and command as usual
Dim oConn As New SqlConnection("your-connection-string")
oConn.Open()
Dim oCmd As New SqlCommand("UPDATE table SET ....", oConn)

' start the command execution, and collect the AsyncResult instance
Dim oResult As SqlAsyncResult = oCmd.BeginExecuteNonQuery()

While Not oResult.IsCompleted

  ' do something else

End While

' must now be complete, so get result from AsyncResult instance
Dim iRowsAffected As Integer = oCmd.EndExecuteNonQuery(oResult)
```

If the query you execute will return rows, you just swap the `BeginExecuteNonQuery` and `EndExecuteNonQuery` method calls with `BeginExecuteReader` and `EndExecuteReader`, then access the `DataReader` that's returned after the `EndExecuteReader` call. Similar logic applies if you want to return an `XmlReader`, whereupon you would use the `BeginExecuteXmlReader` and `EndExecuteXmlReader` methods.

All this assumes that you want to execute only one command and that you need to wait for it to complete. It's also possible to start off more than one `Command` execution process; however, in this case you must execute each one over a separate connection unless you are using the MARS feature of ADO.NET 2.0.

The Asynchronous Callback Model

The second approach to asynchronous execution involves providing a routine in the code that will act as a callback. It will be executed when the action you specify occurs, rather like the way that an event handler is called to handle a user's interaction with an application.

To specify the callback routine you create a new instance of the `AsyncCallback` class, providing the name of that routine as the parameter, and pass this `AsyncCallback` instance into the method you use to start execution of the command. In Listing 3.5, we use a query that returns some rows, so it is executed with the `BeginExecuteReader` method.

Listing 3.5. An Example of the Asynchronous Callback Model

```
' create connection and command as usual
Dim oConn As New SqlConnection("your-connection-string")
oConn.Open()
Dim oCmd As New SqlCommand("SELECT * FROM table", oConn)

' start the command execution, and collect the AsyncResult instance
oCmd.BeginExecuteReader(New AsyncCallback(AddressOf DisplayRows), _
                        CommandBehavior.CloseConnection)

' do something else here
' ...
' and then finish
```

Code can continue to do other things after the query in the `Command` has been executed. During this period, as soon as the query has completed, the callback handler named `DisplayRows` is called (on an arbitrary thread, which is unlikely to be the same thread as the rest of the code). The callback event handler receives a reference to an `AsyncResult` instance that is created automatically by the `BeginExecute`*xxx* method, from which it can obtain a reference to the results of the query by calling the appropriate `EndExecute`*xxx* method, as shown in Listing 3.6.

Listing 3.6. The DisplayRows Callback Event Handler

```
' callback handler routine to display results
Sub DisplayRows(oResult As SqlAsyncResult)

  ' must now be complete, so get result from AsyncResult instance
  Dim oReader As DataReader = oResult.EndExecuteReader(oResult)

  ' display results using oReader - which is a normal DataReader
  ' ...

End Sub
```

The Asynchronous Wait Model

The most complex of the asynchronous methods is also the most efficient if all you want to do is start some commands running against one or more data sources (they can all use separate connections and therefore different databases if required) and not execute other code in the meantime. You simply want to wait until one, more, or all of the commands have completed and then perhaps display some results.

In this case, you start each process in the same way as the previous examples but then use the `AsyncResult` to create a `WaitHandle` that you use to monitor each process. If there is more than one process and you want to wait for one or all of these to complete, you use an array of `WaitHandle` references. The code in Listing 3.7 shows the simplest case of waiting for one command to complete.

Listing 3.7. A Simple Example of the Asynchronous Wait Model

```
' create connection and command as usual
Dim oConn As New SqlConnection("your-connection-string")
oConn.Open()
Dim oCmd As New SqlCommand("UPDATE table SET ...", oConn)

' execute command and get back AsyncResult instance
Dim oResult As SqlAsyncResult = oCmd.BeginExecuteNonQuery()

' use AsyncResult instance to create a WaitHandle
Dim oHandle As WaitHandle = oResult.AsyncWaitHandle

' tell code to sleep here until process is complete
oHandle.WaitOne()

' process is complete, so get results
Dim iRowsAffected As Integer = com.EndExecuteNonQuery(oResult)
```

Checking Whether the Process Timed Out

In the code in Listing 3.7, there is no check to see whether the process completed successfully or failed through a timeout. It's easy enough to specify the timeout you want and then to detect whether the process actually did complete within that period, as shown in Listing 3.8.

LISTING 3.8. Waiting for a Single Process to Complete or Time Out

```
...
' use AsyncResult instance to create a WaitHandle
Dim oHandle As WaitHandle = oResult.AsyncWaitHandle

' set the timeout to 10 seconds
oHandle.WaitTimeout = 10000

' tell code to sleep here until the process is complete
' on return, check the status of the process
If oHandle.WaitOne() = True Then

  ' process is complete, so get results
  Dim iRowsAffected As Integer = oCmd.EndExecuteNonQuery(oResult)

Else

  ' process timed out
  lblError.Text = "Process failed to complete within 10 seconds"

End If
```

Using Multiple Wait Handles

If you are executing more than one process at a time, you use an array of `WaitHandle` instances to detect when one or all of the processes have completed. First you need to create the separate `Connection` and `Command` instances (unless you are using MARS over a single connection), as shown in Listing 3.9.

LISTING 3.9. An Example of Multiple Wait Handles

```
' create three connections and three commands, and open connections
Dim oConnA As New SqlConnection("connection-string-A")
Dim oCmdA As New SqlCommand("SELECT * FROM tableA", oConnA)
oConnA.Open()

Dim oConnB As New SqlConnection("connection-string-B")
Dim oCmdB As New SqlCommand("UPDATE tableB SET ...", oConnB)
oConnB.Open()
```

continues

```
Dim oConnC As New SqlConnection("connection-string-C")
Dim oCmdC As SqlCommand
oCmdC = New SqlCommand("SELECT * FROM tableC FOR XML AUTO", oConnC)
oConnC.Open()
...
```

Now the code can start the three processes running, collecting an `AsyncResult` instance for each one (Listing 3.10). These `AsyncResult` instances are queried to get the three `WaitHandles`, and they are assigned to an array named `aHandle`. In Listing 3.10, the code will wait for all three processes to complete because it uses the static `WaitAll` method of the `WaitHandle` class and passes in the array of three `WaitHandle` instances. At the point where all are complete, the code "wakes up" and retrieves the three results using the appropriate `EndExecute`*xxx* methods.

Listing 3.10. Executing the Commands with Multiple Wait Handles

```
...
' execute commands and get back AsyncResult instances
Dim oResultA As SqlAsyncResult = oCmdA.BeginExecuteReader()
Dim oResultB As SqlAsyncResult = oCmdB.BeginExecuteNonQuery()
Dim oResultC As SqlAsyncResult = oCmdC.BeginExecuteXmlReader()

' use all three AsyncResult instances to create WaitHandle array
Dim aHandle(2) As WaitHandle
aHandle(0) = oResultA.AsyncWaitHandle
aHandle(1) = oResultB.AsyncWaitHandle
aHandle(2) = oResultC.AsyncWaitHandle

' tell code to sleep here until all the processes are complete
WaitHandle.WaitAll(aHandle)

' processes are all complete, so get results
Dim oDataReader As DataReader = oCmdA.EndExecuteNonQuery(oResultA)
Dim iRowCount As Integer = oCmdB.EndExecuteNonQuery(oResultB)
Dim oXmlReader As XmlTextReader = oCmdC.EndExecuteXmlReader(oResultC)
```

If you want to test whether any of the processes did not complete because of a timeout, you just check the returned value from the `WaitAll` method. If it's `False`, at least one of the processes timed out. And if you want to specify the timeout, you can do so in the call to the `WaitAll` method. The following code sets the timeout to 10 seconds:

```
WaitHandle.WaitAll(aHandle, 10000, False)
```

Handling Multiple Processes as Each One Completes

Rather than waiting for all processes to complete, you can improve your code efficiency even more by using the `WaitAny` method. This wakes up

your code as soon as any one of the processes completes. You handle the results of this process and then put the code back to sleep again until the next process completes. This must be repeated until all the processes in the array of `WaitHandle` instances are complete or have timed out.

The code in Listing 3.11 shows how this works. The code that creates the three connections and three commands, executes these commands, and creates the array of `WaitHandle` instances is not repeated here. All this is the same as in Listings 3.9 and 3.10.

What differs is that the code calls the `WaitAny` method this time. The `Integer` value this method returns is either the index within the `WaitHandle` array of the process that completed successfully or the value of the timeout if one of the processes timed out before it was complete. Note that the order in which processes complete or time out will probably not be the same as the order of their `WaitHandle` instances within the array.

To be sure of handling every one of the commands that are executing asynchronously, you must force the code to call the `WaitAny` method once for every command that is executing. If you call `WaitAny` only once, you'll get only one indication of a process completion. Trying to access all the results at this point is likely to cause an error because the rest of the commands may not have completed when you call the respective `End Execute`*`xxx`* method.

So, in Listing 3.11, a `For..Next` loop executes the `WaitAny` method three times. As each process completes, the code checks the index value to see which process just finished and calls the appropriate `EndExecute`*`xxx`* method. (The results can also be displayed at this point to take maximum advantage of the asynchronous processing model.)

LISTING 3.11. Handling Completion of Multiple Asynchronous Processes

```
' code as before to create and execute three commands
...

' code here to create array of WaitHandle instances
...

' following code is executed once for each WaitHandle instance
' so that all three end up being handled at some point
For iLoop As Integer = 1 To 3

  ' tell code to sleep here until any one of the processes completes
  ' collect the index of the next one to complete successfully
  ' NB: this will be the (Static) timeout value if one times out
  Dim iIndex As Integer = WaitHandle.WaitAny(aHandle, 5000, False)
```

continues

```
  ' one of the processes has completed or timed out
  Select Case iIndex

    Case 0:
      ' command A completed successfully
      Dim oDataReader As DataReader
      oDataReader = oCmdA.EndExecuteReader(oResultA)

    Case 1:
      ' command B completed successfully
      Dim iRowCount As Integer
      iRowCount = oCmdB.EndExecuteNonQuery(oResultB)

    Case 2:
      ' command C completed successfully
      Dim oXmlReader As XmlTextReader
      oXmlReader = oCmdC.EndExecuteXmlReader(oResultC)

    Case WaitHandle.WaitTimeout:
      ' this one timed out - could use Case Else here instead
      lblError.Text &= "One process has timed out"

  End Select

Next
```

Canceling Processing for Asynchronous Commands

There may be an occasion where a process is executing asynchronously and you want to provide the user with the opportunity to cancel it part-way through. To do so, all you have to do is call the `Cancel` method of the `Command` instance, as shown in Listing 3.12.

LISTING 3.12. Canceling Asynchronous Command Execution

```
...
Dim oResult As SqlAsyncResult = oCmd.BeginExecuteNonQuery()

' ... do something else here ...

If (some-condition-is-met) Then
  oCmd.Cancel()
End If
```

Asynchronously Opening Connections

As well as executing a `Command` asynchronously, ADO.NET 2.0 also allows you to open connections asynchronously. This is useful, of course, when you are accessing more than one database at a time so that you can execute

multiple asynchronous commands. The principle for asynchronous connections is the same as that we looked at for commands, with the same three options of using the polling, callback, or wait approaches. And, like the asynchronous command execution feature, asynchronous connection opening applies only to the `SqlClient` namespace classes—in this case, `SqlConnection`.

The `SqlConnection` class exposes a couple of properties that are useful when working asynchronously, as shown in Table 3.6.

There is also a method that allows you to start the process of opening the connection and one to complete the process, as shown in Table 3.7.

TABLE 3.6. The Asynchronous Properties of the SqlConnection Class

Property	Description
`Asynchronous`	Returns a `Boolean` value indicating whether the connection has been opened asynchronously. Read-only.
`State`	Returns a value from the `System.Data.Connection State` enumeration indicating the state of the connection. The possible values are: `Closed`, `Connecting`, `Open`, `Executing`, `Fetching`, and `Broken`.

TABLE 3.7. The Asynchronous Methods of the SqlConnection Class

Method	Description
`BeginOpen`	Starts opening a connection asynchronously. The return value is a reference to an object that implements the `IAsyncResult` interface, in this case an instance of the `SqlAsyncResult` class, which is used to monitor and access the process as it runs and when it is complete. There are two overloads for this method:
`async-result = connection``.BeginOpen()`	Starts opening the connection as detailed above.
`async-result = connection``.BeginOpen(`*`callback, state`*`)`	As above, but takes an `AsyncCallback` instance that specifies the callback routine to be executed when the process is complete, plus an `Object` that defines state information for the process.
`EndOpen`	Ends the process of opening the connection specified in the `IAsyncResult` class instance passed to the method. The single overload is:
`connection``.EndOpen(`*`async-result`*`)`	

Asynchronous Connection Examples

The code in Listings 3.13, 3.14, and 3.15 demonstrates the three asynchronous models for opening a connection. The polling model, shown in Listing 3.13, collects the `AsyncResult` instance from the `BeginOpen` method. Then it repeatedly tests the connection state until the connection is open and calls the `EndOpen` method with the `AsyncResult` instance.

Listing 3.13. Opening a Connection Using the Asynchronous Polling Model

```
' create a new connection and open asynchronously
Dim oConn As New SqlConnection("your-connection-string")
Dim oResult As SqlAsyncResult = oConn.BeginOpen()

While Not oConn.State = ConnectionState.Open

  ' do something else

End While

oConn.EndOpen(oResult)
```

The callback model, shown in Listing 3.14, creates a new `AsyncCallback` instance that defines the name of the callback routine and passes this to the `BeginOpen` method to start opening the connection. Once the connection is open, the callback routine (named `ConnectionOpened` in this example) is called.

Listing 3.14. Opening a Connection Using the Asynchronous Callback Model

```
' create a new connection and open asynchronously
Dim oConn As New SqlConnection("your-connection-string")
oConn.BeginOpen(New AsyncCallback(AddressOf ConnectionOpened))

' do something else here
' ...
' and then finish
'----------------

' callback handler routine
Sub ConnectionOpened(oResult As SqlAsyncResult)

  oConn.EndOpen(oResult)

End Sub
```

Finally, the wait model, shown in Listing 3.15, creates a new `WaitHandle` for the process from the `AsyncResult` instance and calls its `WaitOne` method

to "sleep" the code until the connection is open, then calls the `EndOpen` method.

Listing 3.15. Opening a Connection Using the Asynchronous Wait Model

```
' create a new connection and open asynchronously
Dim oConn As New SqlConnection("your-connection-string")
Dim oResult As SqlAsyncResult = oConn.BeginOpen()

' use AsyncResult instance to create a WaitHandle
Dim oHandle As WaitHandle = oResult.AsyncWaitHandle

' tell code to sleep here until process is complete
oHandle.WaitOne()

' process is complete
oConn.EndOpen(oResult)
```

Catching Asynchronous Processing Errors

One thing that the examples of asynchronous processing do not demonstrate is handling any errors that might occur. Connection errors are particularly common in data access applications, and you will probably decide to use a `Try..Catch` construct and error handling code in every case.

However, remember that in asynchronous processing situations, the error will be raised somewhere between calling the `Beginxxx` and `Endxxx` methods—rather than at the specific point where you call the `Open` method (for a `Connection`) or the `Executexxx` method (for a `Command`) when using synchronous processing. This means that you must enclose the complete section of code in a `Try..Catch` construct and then use smaller nested constructs to catch specific errors as and where required.

An Overview of ObjectSpaces

ObjectSpaces is probably one of the most revolutionary and exciting of the new techniques being introduced into .NET as far as data access is concerned. At the moment it is less than fully mature, but it already provides a useful framework for building applications that access real-world data.

The concept of ObjectSpaces is to allow you to access, manipulate, and update data in a relational database using standard .NET "objects," rather than having to write stored procedures or build SQL statements that specifically match the data you want to work with to the structure of the source table in the database. And, in the future, the technology will be extended to

support access to information in other types of data stores, not just relational databases. However, in the current release, support extends only to SQL Server 2000 and SQL Server "Yukon."

The objects you use to store your data are ordinary .NET classes that can contain public and private methods and properties and expose one or more constructors. The ObjectSpaces technology then maps these objects to the tables and columns in the database and automatically wires them up so that you can read, manipulate, and update the data in the tables simply by calling methods within the ObjectSpaces interface.

For all this to work, ObjectSpaces exposes a series of classes that include the following:

- A base class called `ObjectSpace` that performs the basic operations on the data and from which the class instances you require to manipulate the data can be created
- A class named `ObjectSources` that contains the connection and transaction information to be used when accessing databases
- A class named `ObjectQuery`, together with a language called OPath (broadly based on XPath), which is used to select objects from a results set
- A class named `ObjectReader` that follows the same broad principles as other `Reader` classes and is used to fetch objects from the database and expose them as a stream
- A class named `ObjectSet` that broadly mirrors the `DataSet` but is used to hold copies of objects, allowing them to be remoted or accessed in a disconnected environment
- Two classes named `ObjectList` and `ObjectHolder`, which are used to manage delayed loading for objects and hence maximize performance

Basic ObjectSpaces Techniques

The basic principles for using ObjectSpaces are much the same as the existing .NET data access approaches. You create an `ObjectSpace` instance, specifying the mappings to the data and the `ObjectSources` connection and transaction information:

```
Dim oSpace As ObjectSpace
oSpace = New ObjectSpace(mappings-file, connection)
```

The *`mappings-file`* parameter can be the path to a file containing the mappings or a `MappingSchema` instance that defines the mappings. The

connection parameter can be a standard database connection (`SqlConnection` or `DbConnection`) or an instance of an `ObjectSources` object that defines connection and transaction information.

Then you can create an `ObjectReader` instance from the `ObjectSpace`. Any filtering or selection operation you need to accomplish is specified in an `ObjectQuery` instance that indicates the object type and uses `OPath` as the query language. The following code retrieves objects of type `Customer`, where the `region` value is `Europe`:

```
Dim oQuery As New ObjectQuery(Customer, "region='Europe'")
Dim oReader As ObjectReader = oSpace.GetObjectReader(oQuery)
```

The `Customer` object instances can then be retrieved from the `DataReader` using the `Read` method. Each instance is cast (converted) to the correct `Customer` type as it is retrieved from the `ObjectReader` (see Listing 3.16).

LISTING 3.16. Retrieving and Accessing a Customer Object

```
...
Dim oCust As Customer

While oReader.Read()

  ' retrieve object and cast to Customer type
  oCust = CType(oReader.Current, Customer)

  ' display or use values in Customer object
  lblResult.Text &= oCust.CustomerName

End While
oReader.Close
```

In future releases, `ObjectSpace` will also be able to return an `ObjectReader` that uses paging, rather like the new `ExecutePageReader` method does for a standard `DataReader`. However, the exact details of how or when this will be implemented are still under discussion.

Using an ObjectReader

An `ObjectReader` provides stream-based access to objects, rather like a `DataReader` does to data rows when executing a normal SQL query. You can use it for data binding to Windows Forms and Web Forms controls. However, bear in mind that, like all `Reader` objects, it works in the connected scenario only, and you must be sure to close it when you have finished using it.

Filling and Manipulating an ObjectSet

To hold copies of objects for remoting, processing, and updating, you use an `ObjectSet`. The `ObjectSet` is basically a collection of your objects, but it also performs other tasks like tracking the original values when objects are updated and allowing the collection to be sorted.

An `ObjectSet` is created using the `GetObjectSet` method of the current `ObjectSpace` instance. The objects you require can be specified using an existing `ObjectQuery` instance:

```
Dim oSet As ObjectSet = oSpace.GetObjectSet(oQuery)
```

or by specifying the object type as a `Type` instance, and an `OPath` query as a `String`:

```
Dim oSet As ObjectSet = oSpace.GetObjectSet(object-type, query)
```

> If you only want to retrieve a single instance of an object from the database, you can use the `GetObject` method of the `ObjectSpace` instead. It uses the same combination of parameters as the `GetObjectSet` method but just returns a new instance of the object. However, `GetObject` will raise an error if the query results in more than one object being returned.

The `ObjectSet` also exposes methods that allow you to add objects to the collection or remove them from the collection. The `Add` method takes either a single object instance or a collection of object instances and adds them to the `ObjectSet` collection. To remove an object you use the `MarkForDeletion` method. As the name suggests, the object remains in the `ObjectSet` but is marked as being deleted. When the changes to the `ObjectSet` are persisted to the database, the object is removed from the database (this is, of course, just the same as the way the standard ADO.NET `DataSet` works).

Saving the Changes to an ObjectSet

One of the features that make ObjectSpaces such an exciting technology is that the `ObjectSet`, like the ADO.NET `DataSet`, provides methods that push changes to the objects back into the database. The `PersistChanges` method of the `ObjectSpace` pushes the changes made in an `ObjectSet`

back into the database, and the `Resync` method synchronizes the `ObjectSet` and the database. Both methods can be applied to a single object or to a collection of objects (or the complete `ObjectSet` contents).

Listing 3.17 shows a simple example of the `ObjectSet` in action. It collects a set of `Customer` objects from the database into an `ObjectSet` using the `GetObjectSet` method. Then, for each one, it updates the value of the `Status` field to `Checked`. Next, one of the `Customer` objects is marked for deletion, and a new one added, before the changes are persisted to the database.

Listing 3.17. Manipulating Customer Objects

```
' get ObjectSpace using mappings and connection
Dim oSpace As ObjectSpace = New ObjectSpace(mappings-file, connection)

' create ObjectSet from ObjectSpace using OPath query
Dim oSet As New ObjectSet _
    = oSpace.GetObjectSet(Customer, "State='WA'")

' iterate through objects in ObjectSet setting the Status field
For Each oCust As Customer In oSet
  oCust.Status = "Checked"
Next

' mark the Customer object at index 7 for deletion
oSet.MarkForDeletion(oSet(7))

' add a new Customer object
oSet.Add(New Customer("name", "address", "status"))

' persist the changes into the database
oSpace.PersistChanges(oSet)
```

Object Mappings

The ObjectSpaces technology seems like a natural progression from existing ADO.NET data management techniques, and indeed it is. The power comes from the mappings that are created between the objects and the columns and tables in the source database. These mappings are defined in XML and take advantage of a new approach that Microsoft has introduced to help blur the distinctions between different types of data and different types of data stores.

This new approach uses a three-part mapping system to map different types of data sources and data consumers. The details of the generic three-part mapping technology are described in Chapter 7, where they are applied to mapping XML to relational data. However, exactly the same approach is taken to map objects to relational data stores.

The process involves three mapping files. The relational database schema model describes the way data is stored in the relational database, and an object schema model describes how the objects themselves store and view the data they hold. The third file actually maps the elements in each of the other two to each other, to provide the "bridge" between the two disparate models.

Hierarchical Data

In this brief introduction we've concentrated on the way that ObjectSpaces can be used and what the technology offers to developers. We haven't spent time looking at the actual details of the objects it can manipulate because in essence they are just standard .NET classes. The important feature of ObjectSpaces is that it is designed from the ground up to make it much easier to work with hierarchical or related data in the form of objects.

What Are Hierarchical Objects?

As an example of hierarchical objects, a `Customer` object may be related to one or more `Contact` objects, each of which describes a person at the company defined by the `Customer` object. A property exposed on the `Customer` class—for example a `Contacts` property—can define this relationship. The `Contacts` property is simply a reference to a collection of `Contact` objects.

Looked at from the relational database point of view, however, the `Customer` object is the parent in a one-to-many relationship with child `Contact` objects. ObjectSpaces provides methods to work with related objects like these; for example (using the description of the objects in the previous paragraph), you can add a new `Contact` object as a child of an existing `Customer` object using the following code:

```
Dim oCust As Customer = oSpace.GetObject(Customer, "ID=2756")
oCust.Contacts.Add(New Contact("name", "position", "phone"))
```

More than that, ObjectSpaces is designed to handle graphs of interrelated objects. Even in cases where an XML-type hierarchy is not natural (because of multiple parents for a given object), you can easily represent and build the data model using ObjectSpaces. Likewise, it can handle

many-to-many relationships that are not always expressed naturally as a hierarchy, for example, a `Customers` to `Products` relationship.

Loading Child Objects

One issue that springs to mind where hierarchical or related-object data access is concerned is performance. By default, a request that fetches objects matching an `OPath` query (such as `"State=NV"`) will return only the objects that match the query, not any child or related objects. However, this means that you have to go back to the database each time you want to access a child object.

ObjectSpaces addresses this by using the concept of a **span**. Basically a span is a comma-delimited list of the objects you want to retrieve and is used in conjunction with the `ObjectQuery` class. So, if you specify `Customer, Customer.Contact` as the span for the operation, the related child objects are extracted and returned as well, without requiring a further query to the database.

A third option, called **lazy loading**, is also worth considering. In this case, where objects are stored in an `ObjectHolder` and `ObjectList`, a request for an object will be intercepted and all of the related target objects are automatically retrieved as well.

SUMMARY

This chapter has taken a focused look at three new topics in ADO.NET 2.0 that can really improve the performance of your data access code—in a variety of ways. As discussed at the start of the chapter, accessing multiple databases, or executing multiple commands against the same database, can lead to code blocking as it waits for expensive processes like opening a database connection or fetching rows from a database to complete.

Instead, the new MARS and asynchronous command execution features in ADO.NET 2.0 can be used individually or together, in a range of scenarios, to reduce the occurrence of code blocking and to minimize the number of connections required by your code.

We also looked at one of the most exciting of the new technologies to come to ADO.NET—ObjectSpaces. This technology allows developers to work easily with objects that expose data in a structured and natural way. On top of this, objects can be reusable, and this approach suits modern application design methodologies. ObjectSpaces removes the need for developers to build complex code into their objects to support serialization, data

store persistence, and interaction with Web Services. By simply defining mappings between the objects and the data source (tools will be available to automate this), you can retrieve, manipulate, and persist data as pure .NET objects.

The next chapter completes our tour of the ADO.NET relational data features in version 2.0 of the Microsoft .NET Framework with a look at some more new topics. This time, however, they are all related to the new version of SQL Server, code-named "Yukon."