5

ADO.NET and SQL Server 2005

ADO.NET AND SQL SERVER, in previous incarnations, generally enjoyed separate design and development routes. But ADO.NET 2.0 and SQL Server 2005 (formerly code-named "Yukon") share a great deal of functionality and complementary technologies. As these two separate products began to approach final release, it was necessary to merge their functionality and release schedules so that they can provide an integrated platform for the new technologies they both implement.

Chapter 1 discussed the many areas where ADO.NET 2.0 and SQL Server have converged, and in this and subsequent chapters you'll see these features described in more detail. The support for hosting ADO.NET code within SQL Server 2005 is covered in Chapter 6, and the new `xml` data type and the way it can be used as a type in a table column are covered in Chapter 7. In this chapter, we concentrate on three core areas of functionality:

- Multiple Active Results Sets (MARS)
- SQL Server Query Notifications
- SQL Server user-defined types (UDTs)

We start, however, with a trip to another planet—let's pay a visit to MARS.

Multiple Active Results Sets

ADO.NET 2.0 provides a great new feature called **Multiple Active Results Sets**. In effect, the name alone is enough to indicate the aims of this technology: to allow you to have more than one results set active on the same data store connection. In ADO.NET 1.x, if you open a `DataReader` over a connection, you cannot use that connection for any other process until the `DataReader` is closed. If you attempt to reuse the connection while the `DataReader` is open, you get an exception with a message indicating that the connection is in use.

In ADO.NET 2.0, you can open more than one `DataReader`, more than one `XmlReader` (when using the `ExecuteXmlReader` method), or any mixture of these concurrently over the same connection to SQL Server 2005. Plus, while one or more readers are open, you can execute any other commands over the same connection. For example, you can execute SQL statements to update the database, call stored procedures (including procedures that return values as parameters or rowsets), and execute Data Manipulation Language (DML) statements that change the structure of the database tables or otherwise manipulate the contents of the data store.

You can also interleave reads from the open readers. In other words, you can read rows (or row elements in the case of the `XmlReader`) from any of the open readers in any order. For example, you can read a row from one reader, execute a command to create a new reader, read rows from it or from the original reader, and close the readers in whatever order suits your application requirements.

One point to bear in mind is that transactions in .NET are connection-based. When you create a `SqlTransaction` for a connection, you assign this to the `SqlCommand` by setting the `Transaction` property as follows:

```
myCommand.Transaction = myTransaction
```

This means that all the processes you execute over a single connection share the same transaction. Of course, you can create nested transactions for specific operations and define a transaction within a stored procedure if you need more granular process control.

Scenarios for Using MARS

Although the theory behind the MARS technology is simple enough to grasp, the interesting point is deciding where this approach is useful. At first glance, you may think that it could be used almost everywhere you perform data access and that it would make data access easier because you

don't have to be concerned about when and where in your code you open a reader. However, this is not the best way to approach the technology. Instead, in this section we look at the main situations where MARS is of real benefit.

> **NOTE:** The ability to have multiple readers open at the same time increases the possibility that one or more readers (previously opened elsewhere in your code) may be left open. When using multiple readers, be sure to close each one as soon as possible. The presence of multiple open readers on a connection reduces scalability, and leaving them open when not required affects your application response and resource usage.

Batch Processing with MARS

One situation your application might have to handle is that of processing a large number of rows from a table, perhaps running into hundreds of thousands or more. Although the `DataSet` in version 2.0 can handle a large number of rows, it makes sense in many cases to use a `DataReader` rather than loading all the rows into a `DataSet`. However, your code may have to take other actions based on the values in each row you process, for example, reading or updating other tables in the same database. This may require loading all the data into a `DataSet` at once, perhaps as a set of multiple tables, or breaking the process down into separate batches of a specific number of rows.

In a batch processing scenario where there are very large numbers of rows, MARS is useful because you can open another `DataReader` over the connection without closing the first one. Without MARS, when you wanted to perform actions on other tables in the database, you would have to do one of the following:

- Open a new connection, with the subsequent processing overhead, resource usage, and connection delay this involves
- Close the existing reader, storing a marker for the current position within the rowset, and then reexecute the query, open a new reader, and move from the start of the rowset to the next row that requires processing

With MARS, you can leave the main rowset open at the current row, execute other commands and open other readers as required, and then continue with the next row in the original rowset simply by calling the `Read` method of that reader.

MARS also has another useful advantage: All the operations over a single connection can be accomplished within a single lightweight local transaction. If, on the other hand, you are using separate connections to perform multiple updates that require overall transactional support, you must take advantage of a distributed transaction coordinator and associated service, such as Windows Component Services.

Complex Data Processing with MARS

In another situation, you might want to perform complex processing that requires access to multiple tables while you process another table, but you want to do this only on some occasional basis. Although this seems to be similar to the batch processing scenario just described, here we're discussing even quite small original rowsets. For example, consider the task of calculating the cost of several types of discounts that you offer to your customers. There might be only 50 different types of sales promotions, but to calculate the cost of each one probably involves access to all the product and order details in the database.

Again, you could load all the relevant data from all the tables into a `DataSet` and then process it there. Or you could load just the discounts rowset into a `DataSet` or `DataTable` and then iterate through the rows, accessing the other data in the database as and when required. But MARS offers you the alternative to perform all the processing by using readers. Listing 5.1 shows an example of how the process might work. This example involves extracting values as rowsets and as singleton values (using `ExecuteScalar`), as well as executing an `INSERT` statement or stored procedure to insert a new row into another table—all while the original `DataReader` is open.

LISTING 5.1. A Scenario for Calculating Customer and Product Discount Costs

```
Open a connection to the Discounts table
Execute a Command to get a rowset of Discounts rows
For Each row in the Discounts rowset
  Read the discount identifier key from the row
  Read the start and end dates for this discount
  Execute a Command to get a rowset of order lines for this ...
          ... discount and time period
```

```
  Calculate the discount given for each item and total discount ...
          ... for the quantity ordered
  Look up the row in the Orders table for this order line row
  Look up the row in the Customers table that matches this Orders row
  Calculate the cost of any other customer-specific extra discount
  Add a new row containing the results of these calculations ...
          ... to an audit table
Process the next Discounts row
```

Using Asynchronous Commands with MARS

In Chapter 2, you saw how ADO.NET 2.0 supports asynchronous execution of commands to SQL Server through the `Command` class in the `SqlClient` namespace. It may seem like a useful approach to combine this with MARS so that you can open multiple readers asynchronously over a single connection. However, this is generally not recommended, and you should avoid mixing the technologies on the same connection.

The reason for this is that both asynchronous commands and MARS reduce the "parallelism" of the processes using the connection. Multiple operations may be completing simultaneously on multiple threads, and correct synchronization of these operations is very difficult to achieve. This is particularly the case when you use the asynchronous callback model (described in Chapter 2), where the multiple callbacks from a connection are most likely to be on different threads.

So, unless you have some specific requirement that demands asynchronous execution but can use only a limited number of connections, you should avoid the combination. And if you do use it, the best plan is to use the `Beginxxx` and `Endxxx` methods that run on a single thread and to avoid the callback model.

Displaying Customer and Order Details with MARS

Now that you know how useful MARS can be, you'll see an example of it in action in this section. This example shows how you can interleave reads from multiple readers over a single connection.

The sample AdventureWorks database we're using in this example contains a series of tables that expose data about the orders placed by stores that sell equipment supplied by the fictional AdventureWorks Company. To get rowsets that can be used to display information about customers (their names and addresses) and their orders, we have to join and query several tables. Listing 5.2 shows the SQL statements that retrieve the following rowsets used in this example:

- The name and address details of two customers
- The set of orders for a specific customer, including the order number, order date, and total value
- The set of order lines for a specific order, including the quantity, price, discount, and line value

LISTING 5.2. The SQL Statements Used to Retrieve the Three Rowsets for This Example

```
' declare SQL statements to extract rows
Const queryCust As String = "SELECT Sales.Store.CustomerID, " _
  & "Sales.Store.Name, AddressLine1, City, PostalCode, " _
  & "Person.StateProvince.Name " _
  & "FROM Sales.Store JOIN Sales.CustomerAddress ON " _
  & "Sales.Store.CustomerID = Sales.CustomerAddress.CustomerID " _
  & "JOIN Person.Address ON " _
  & "Sales.CustomerAddress.AddressID = Person.Address.AddressID " _
  & "JOIN Person.StateProvince ON " _
  & "Person.Address.StateProvinceID = " _
  & "Person.StateProvince.StateProvinceID " _
  & "WHERE Sales.Store.CustomerID = 501 " _
  & "OR Sales.Store.CustomerID = 503"

Const queryOrders As String = "SELECT SalesOrderID, " _
  & "PurchaseOrderNumber, OrderDate, TotalDue " _
  & "FROM Sales.SalesOrderHeader WHERE CustomerID = @CustID"

Const queryLines As String = "SELECT Production.Product.Name, " _
  & "OrderQty, UnitPrice, UnitPriceDiscount, LineTotal " _
  & "FROM Sales.SalesOrderDetail JOIN Production.Product " _
  & "ON Sales.SalesOrderDetail.ProductID = " _
  & "Production.Product.ProductID " _
  & "WHERE SalesOrderID = @OrderID"
```

The example fetches the rows for the two specified customers and displays their name and address details. Then—while this rowset is open over the connection—it executes another command that returns a rowset of the related rows from the `SalesOrderHeader` table for the current customer. As it iterates over the list of orders, the code displays the purchase order number, the order date, and the total order value. Next—while this second reader is also open—it executes a third command to fetch all the rows that contain details of the individual items or order lines for the current order. Figure 5.1 shows the result.

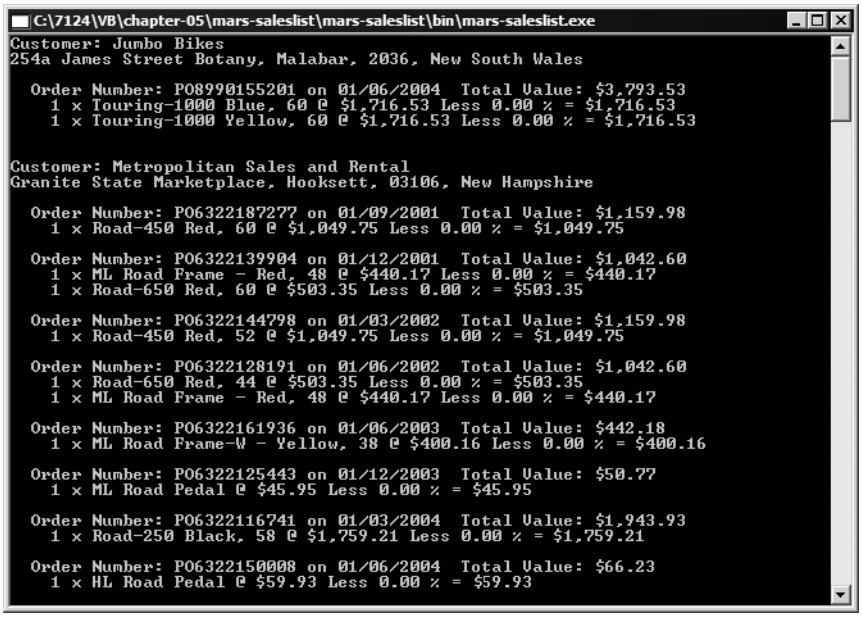

FIGURE 5.1. The output of the example page that uses MARS to open three DataReaders over the same connection

> This example is among the samples you can download from our Web site at http://www.daveandal.net/books/7124/. You'll find it in the `chapter-05` folder as a console application named `mars-saleslist`. The examples are available in both Visual Basic .NET and C#.

Creating the Connection and Commands

The code starts in the usual way by creating a single `Connection` and then three `Command` instances (you can't reuse a `Command` instance while the `DataReader` it generated is still open, as is the case in this example). Each `Command` instance uses one of the three SQL statements declared earlier in the page.

The second and third SQL statements contain a parameter placeholder used to select the appropriate orders and order lines for the current customer and order. The two parameters are added to the second and third `Command` instances as they are created. Both are used to specify the primary key of the rows that should be selected, and both are of type `SqlDbType.Int` to match the `int` column types in the database (see Listing 5.3).

LISTING 5.3. Creating the Connection and Command for the MARS Example

```
' create the connection
Dim conn As New SqlConnection(connect)

' create a Command to retrieve customer rows
Dim cmdCustomers As New SqlCommand(queryCust, conn)

' create a Command to retrieve order rows
' cannot use the existing command while the rowset is open
Dim cmdOrders As New SqlCommand(queryOrders, conn)

' add a parameter to the command for selecting orders
cmdOrders.Parameters.Add("@CustID", SqlDbType.Int)

' create a Command to retrieve order rows and add
' a parameter for selecting order lines
Dim cmdLines As New SqlCommand(queryLines, conn)
cmdLines.Parameters.Add("@OrderID", SqlDbType.Int)
...
```

Iterating through the Rows

Listing 5.4 shows the remaining code (we've omitted the `Try..Catch` construct to simplify the listing). The code starts to iterate through the customer rows, displaying the details and then using the value of the `CustomerID` column to populate the `@CustID` parameter of the second `Command` instance. This `Command` is then executed to get a rowset containing the list of orders for the current customer. Next, the code starts to iterate through the order rows, again displaying some values from the row and then using the `OrderID` value to populate the `@OrderID` parameter of the third `Command`. This `Command` is then executed, and the list of order items (order lines) is displayed.

LISTING 5.4. Iterating through the Rows in the MARS Example

```
...
' open a rowset containing customer rows
' can use CommandBehavior.CloseConnection here to
' close the connection when this reader is closed
conn.Open()
Dim rdrCustomer As SqlDataReader = _
      cmdCustomers.ExecuteReader(CommandBehavior.CloseConnection)

' iterate through the customer rows
While rdrCustomer.Read()

  ' display details of this customer
  Console.WriteLine("Customer: {0}", rdrCustomer(1))
  Console.WriteLine("{0}, {1}, {2}, {3}", rdrCustomer(2), _
                    rdrCustomer(3), rdrCustomer(4), rdrCustomer(5))
  Console.WriteLine()

  ' set the parameter value for this customer
  cmdOrders.Parameters("@CustID").Value = rdrCustomer(0)

  ' open a rowset containing the list of orders for this customer
  ' note: cannot use CommandBehavior.CloseConnection here because
  ' the single connection must remain open when this reader is closed
  Dim rdrOrder As SqlDataReader = cmdOrders.ExecuteReader()

  ' iterate through the order rows
  While rdrOrder.Read()

    ' display the details of each order
    Console.WriteLine("  Order Number: {0} on {1:d}  " _
      & "Total Value: {2:C}", rdrOrder(1), rdrOrder(2), rdrOrder(3))

    ' set the parameter value for this order
    cmdLines.Parameters("@OrderID").Value = rdrOrder(0)

    ' open a rowset containing the list of order lines for this order
    ' note: again, cannot use CommandBehavior.CloseConnection here
    Dim rdrLines As SqlDataReader = cmdLines.ExecuteReader()

    ' iterate through the order line rows
    While rdrLines.Read()

      ' display the details of each order line
      Console.WriteLine("   {0} x {1} @ {2:C} Less {3:P} = {4:C}", _
                        rdrLines(1), rdrLines(0), rdrLines(2), _
                        rdrLines(3), rdrLines(4))

    End While
```

continues

```
    ' close the order lines reader
    rdrLines.Close()
    Console.WriteLine()

  End While

  ' close the orders reader
  rdrOrder.Close()
  Console.WriteLine()

End While

' close the customer reader
rdrCustomer.Close()
```

Notice that the code to open the root `DataReader` (the one containing the customer details) uses the `CommandBehavior.CloseConnection` value. This reader will remain open throughout the entire process, so it's safe to use this option because the connection will be closed only when this reader is closed at the end of the process. However, you cannot use this option when creating a `DataReader` that will be closed before the process completes, as is the case with the second and third `Command` instances.

SQL Server Query Notifications

In environments where different users regularly update data, there is often a need to refresh cached data when the original data changes. One approach to this is to poll the server at preset intervals to detect changes or to simply fetch a new copy of the data after a prescribed interval has elapsed.

However, these techniques are not the most efficient approach. It leads to extra work and resource consumption that are simply not required if the source data has not changed. To reduce this extra load, wait times between polling and refreshing the data are often increased, which has the negative effect of increasing the latency in seeing changes and, as a result, the frequency with which outdated data is served. Arranging for the server to notify the application that is caching the data of any changes to the source data when, and only when, these changes occur can eliminate this waste of processing and resources.

The caching features implemented in ASP.NET 2.0 and the changes to the underlying request handling mechanism in Internet Information Services (IIS) 6.0 both take this approach, automatically invalidating the cache content based on user-defined rules. Now, in conjunction with updates to some of the classes in the `SqlClient` namespace, SQL Server 2005 offers the same opportunities for relational data.

NOTE: There is a cost associated with registering for and receiving notifications. Query Notifications are not designed to be associated with every ad hoc query (where each result is different) but are intended for use where multiple users or requests share the same query results.

The notification infrastructure scales well to support large numbers of similar queries that vary only on parameter values. It is not designed to scale to support hundreds of different ad hoc queries. As a result, notifications work well for invalidating cached Web responses and Windows Forms `DataSet` instances based on common database queries but not for notifying any arbitrary queries posed by numerous users.

An Overview of Query Notifications

When a query that generates a rowset is executed, code can instruct SQL Server 2005 that it wants to be notified of any change to the source data included in the rowset returned. The changes that cause a notification to occur include the following:

- Tables being truncated, altered, or dropped
- Data being inserted into, updated within, or deleted from the tables, which would result in different values being returned if the query were reexecuted
- An internal server error or the server being restarted

When a notifiable event occurs, SQL Server creates a notification message and places it into a queue on the server. You can think of a notifiable event as *anything* that occurs on the server that would cause the results of the query to be different from the original if it were executed again. A notification can occur immediately when a command is executed. Examples include but are not limited to situations such as a query that is not a suitable `SELECT` query (see the next subsection for details) or a query that causes an error.

It's important to remember that a notification is sent only once. After it has been sent, you will get no other notifications if the data changes again. If you want to continue to monitor for changes, you must reregister for notifications again by subscribing and reexecuting the query.

In some cases a default queue is used, while in other cases you can create and use custom queues (see Tables 5.3 and 5.4 later in this chapter). A list of the queues for a database can be found in SQL Server 2005 Management Studio, in the Service Broker section of the Object Explorer tree for each database.

Query Types Suitable for Notifications

In general, the rules that govern the query types suitable for notifications are the same as those for materialized views, as listed in SQL Server 2005 Books Online. Query Notifications are supported only for suitable `SELECT` queries. This means a single SQL `SELECT` statement that returns rows, or a stored procedure or batch statement that contains one or more `SELECT` queries.

However, there are other limitations. So that the notifications architecture can track the tables and columns in the database correctly, the `SELECT` statement must define the columns explicitly and not use the asterisk (all columns) syntax. The name of the table must also be qualified with the schema name as *`schema.table-name`*. For example, the following `SELECT` statement is *not* suitable and may cause a notification to be sent immediately:

```
SELECT * FROM table
```

The next `SELECT` statement *is* suitable (assuming that the named table and columns exist in the database), and a notification will be sent only when the data changes, the table is modified, or a server error occurs:

```
SELECT ColumnA, ColumnB FROM dbo.MyTable
```

Note that the syntax requirements just stated apply to the current release, although breaking these "rules" may not result in an error or immediate notification. It may also be possible to use a more relaxed syntax in future releases.

Detecting a Notification

How the application detects and reacts to a change notification depends on the type of application and the way that the notification is set up in the first place. Applications can poll the queue to see if a notification has occurred. OK, so this doesn't seem like an advantage over polling the server in some other way to check whether the source data has changed. However, the

process of polling the queue for a notification message is far quicker and far more efficient than querying the source data again.

It's likely that you would only poll the queue just before your application actually uses the data it cached from the original request, rather than at regular intervals, though both approaches are perfectly valid. You might even set it up as a user-controlled action, perhaps with a Refresh button or command, so that users can check for changes when they want to.

Alternatively, with the new asynchronous execution model, you can execute a query against the queue that will block until a change notification is posted while allowing other code in the application to continue to execute. Because change notifications for multiple queries can be delivered to the same queue, you can have this single query listening for a change from multiple queries. Because SQL Server 2005 supports ADO.NET 2.0 asynchronous command execution (as demonstrated in Chapter 2), executing this blocking query asynchronously means that you don't tie up a thread on the client.

Subscribing to Notification Events

An even better way to maximize efficiency is to arrange for the notifications service to raise an event that an application can react to, rather than having to poll the queue. This approach is also implemented in the notifications feature of SQL Server 2005 and ADO.NET 2.0.

Setting Up a Notification

The way that your application arranges to receive notifications determines which of the two available techniques you must use. Setting up a notification to raise an event requires a different set of classes to be used from setting up a notification where your application will poll the queue directly.

To set up a notification that raises an event, you simply create an instance of the `SqlDependency` class and specify an event handler that will be called when the source data changes. This event handler can be associated with a cache entry. For example, in ASP.NET 2.0 the event can invalidate any cached data, forcing the application to refresh the rowset from the database to get the updated data.

Alternatively, you can use the `SqlNotificationRequest` class to send a request for notifications to the server. The notification messages are placed in a queue that you have previously created in the database and that you specify when you create the `SqlNotificationRequest` instance. You then poll the queue (as just described) to detect a notification.

Under the hood, the `SqlDependency` class uses a `SqlNotificationRequest` instance to register for notifications with the server. The `SqlDependency` class effectively acts as a higher-level abstraction, making it easier to create applications that work with notifications using the default behavior. When you need to tailor the behavior, however, you may prefer to use the `SqlNotificationRequest` directly instead. We'll look at this class first.

The SqlCommand Class and the SqlNotificationRequest Class

The SQL notifications feature in ADO.NET 2.0 adds the new `SqlNotificationRequest` class to the `System.Data.Sql` namespace and also adds two new properties to the `SqlCommand` class that allow it to interface with notifications.

The SqlCommand Notification-Related Properties

The `SqlCommand` class in ADO.NET 2.0 can now be bound to an instance of the `SqlNotificationRequest` class through a new property named `Notification` (see Table 5.1), and auto-enlistment can be specified using the new `NotificationAutoEnlist` property.

TABLE 5.1. The Notification-Related Properties of the SqlCommand Class

Property	Description
`Notification`	Sets or returns the `SqlNotificationRequest` instance that is bound to this command and must be set before the command is executed. By default the value is `null`, indicating that no notification request will be registered.
`NotificationAutoEnlist`	Sets or returns a `Boolean` value that indicates whether the command will automatically enlist the notifications service when required. This is particularly useful in controlling notifications behavior in ASP.NET pages, where enlistment is automatic for all commands on the page if a `SqlDependency` is declared as part of a `Cache` directive.

The SqlNotificationRequest Constructors

The two overloads of the constructor for the `SqlNotificationRequest` class are shown in Table 5.2.

TABLE 5.2. The Constructors for the SqlNotificationRequest Class

Constructor	Description
`SqlNotificationRequest()`	Creates a new `SqlNotificationRequest` instance with the properties set to `null`, ready for values to be assigned to them.
`SqlNotificationRequest (id, service, timeout)`	Creates a new `SqlNotificationRequest` instance and sets the properties. See the property descriptions in Table 5.3 for details of the parameters.

The SqlNotificationRequest Properties

The three properties of the `SqlNotificationRequest` class, which are equivalent to the parameters to the constructor in Table 5.2, are shown in Table 5.3.

TABLE 5.3. The Properties of the SqlNotificationRequest Class

Property	Description
`Id`	Sets or returns a `String` that defines application-specific information for this notification. Although not actually used by the notifications system, the ID allows the user to associate a notification with specific parts of the application's state. The value is included in the queue message when a notification occurs, and the application can then invalidate the appropriate cached data. This is useful if notifications from several commands are posted to the same queue.
`Service`	Sets or returns a `String` that is the name of the SQL Broker Service queue to which notification messages will be posted. The service (queue) must be configured on the server.
`Timeout`	Sets or returns an `Integer` that specifies the length of time, in seconds, that SQL Server will continue to monitor for changes. When the timeout expires, a timeout notification is sent, even if no change to the data has taken place. The default is `0`, which means that the default timeout defined for the server will be used.

Using a SqlNotificationRequest

To specify that an application should be notified of changes to the result of executing a query that returns rows, you create a connection and command as usual but then bind a new or an existing `SqlNotificationRequest` to the command before executing it.

> **NOTE:** The `SqlNotificationRequest` class is defined within the `System.Data.Sql` namespace, not `System.Data.SqlClient`. When creating instances of the `SqlNotificationRequest` class, you must reference or import the `System.Data.Sql` namespace into your project.

The code in Listing 5.5 uses the simpler of the two constructors for the `SqlNotificationRequest` and then sets the `Id`, `Service`, and `Timeout` properties before binding the new `SqlNotificationRequest` instance to the `Command` and calling the `ExecuteReader` method to get back a `DataReader` containing the result.

LISTING 5.5. Creating and Registering a Notification Request

```
' create a new SqlNotificationRequest
Dim notify As New SqlNotificationRequest()
notify.Id = "MyNotificationID"
notify.Service = "AdventureWorks.dbo.ServiceBrokerQueue"
notify.Timeout = 300

' create a Command and bind the notification request to it
Dim SQL As String = "SELECT colA, colB FROM database.schema.table"
Dim con As New SqlConnection("connection-string")
Dim cmd As New SqlCommand(SQL, con)
cmd.Notification = notify

' get the results and use them
Dim reader As SqlDataReader = cmd.ExecuteReader()
...
```

The `DataReader` can then be used in the application to display the data. Alternatively, you can use the `Command` to fill a `DataSet` with the returned rows.

Polling the Queue

Because the code has defined a notification request and sent it to the server, any changes to the original data will cause a notification message to be sent to the queue that was specified for the `Service` property. This message will contain the value `MyNotificationID` that was assigned to the `Id` property.

There is no built-in client-side support for polling the notification queue. However, SQL Server 2005 does have the ability to poll the queue

and return a notification message in response to a SQL query. The format of this query is:

```
RECEIVE <field-list> FROM <queue-name>
```

You can approach polling the queue in different ways in terms of timing. The best option depends on the requirements of your application. For example, you can execute the query when the user clicks a refresh button or when an appropriate number of minutes or seconds have passed. Another alternative is to set up a stored procedure that is bound to the Broker Service queue, which is automatically executed when the queue receives the notification.

Another approach, should the previous ones be unsuitable for your requirements, is to execute an asynchronous "blocking" command, such as that shown in Listing 5.6, that will return only when a message is delivered to the queue. It uses a `Command` object that has a polling query defined as the `CommandText` and executes this command asynchronously with the `BeginExecuteReader` method. An event handler named `CommandCompleted` is declared as the callback for this method.

LISTING 5.6. Creating a Blocking Command to Receive a Notification

```
...
' create a blocking command to receive a notification from the queue
Dim sqlPoll As String = "WAITFOR (RECEIVE * " _
    & "FROM AdventureWorks.dbo.ServiceBrokerQueue)"
Dim con2 As New SqlConnection("async-connection-string")
Dim cmd2 As New SqlCommand(sqlPoll, con2)
con2.Open()
cmd2.BeginExecuteReader( _
     New AsyncCallback(AddressOf CommandCompleted), cmd2)
...
```

When a change to the data occurs and a notification message is delivered to the queue, the `CommandCompleted` event handler is automatically called (see Listing 5.7). It can retrieve the message from the queue by using the `DataReader` returned by the `EndExecuteReader` method. (For more details on asynchronous command execution, see Chapter 2.)

LISTING 5.7. Handling the Delivery of a Notification

```
' called when a notification message is delivered to MyQueue
Sub CommandCompleted(ByVal result As IAsyncResult)
  ' get a reference to the original Command from AsyncState
  Dim cmd As SqlCommand = CType(result.AsyncState, SqlCommand)
```

continues

```
  Try
    ' get a DataReader and display the results
    Dim reader As SqlDataReader = cmd.EndExecuteReader(result)
    Console.WriteLine("Data changed at {0}", _
            DateTime.Now.ToLongTimeString())
    ' ... read data from the DataReader here ...
    Console.WriteLine(reader(0))
  Catch e As Exception
    Console.WriteLine("* ERROR: " & e.Message)
  End Try
  cmd.Connection.Close()
End Sub
```

The samples you can download from our Web site at http://www.daveandal.net/books/7124/ include an example of using the `SqlNotificationRequest` class. This is in the `chapter-05` folder as a console application named `notifications-request`. The examples are available in both Visual Basic .NET and C#.

The SqlDependency Class

Instead of creating and binding a `SqlNotificationRequest` instance to the `Command`, you can use the alternative `SqlDependency` approach to define a required notification. In the beta 1 release, the `SqlDependency` class has three overloads for the constructor and exposes three properties, two methods, and a single event. The range of constructors available changes in the beta 2 release, however, with fewer parameters being supported.

The SqlDependency Constructors (Beta 1 Release)

The constructor for the `SqlDependency` class creates a new `SqlDependency` and automatically binds it to a `Command` instance if one is specified. Table 5.4 shows the three overloads.

TABLE 5.4. The Constructors for the SqlDependency Class

Constructor	Description
`SqlDependency()`	Creates a new `SqlDependency` instance with the properties set to `null`, ready for values to be assigned to them.

TABLE 5.4. The Constructors for the SqlDependency Class (*continued*)

Constructor	Description
`SqlDependency(`*`command`*`)`	Creates a new `SqlDependency` instance that is bound to the specified `SqlCommand` instance. The remaining properties are set to `null`, ready for values to be assigned to them.
`SqlDependency(`*`command, service-name, authentication, encryption, transport, timeout`*`)`	Creates a new `SqlDependency` instance that is bound to the specified `SqlCommand` instance. The remaining parameters are: *`service-name`*: The name of the SQL Broker Service queue as a `String`. *`authentication`*: The authentication type to be used by the client listener. This value is one from the `SqlNotificationAuthType` enumeration: • `None`: The connection to the client doesn't use authentication. Any connection will be considered valid. This is the default. • `Integrated`: Use NT integrated authentication. *`encryption`*: The encryption type to be used when sending notification data from the server to the client. This is a value from the `SqlNotificationEncryptionType` enumeration: • `None`: Data is exchanged in plain text. This is the default. • `Certificate`: Use a server certificate to encrypt the SSL channel. Note that the *`encryption`* parameter will be removed from the beta 2 and final release versions. *`transport`*: The method used by the server to send a notification to the client. This value is one from the `SqlNotificationTransports` enumeration: • `Any`: Use HTTP if available, or use TCP otherwise. This is the default. • `Tcp`: Use raw sockets as the transport. This is useful where a kernel-mode HTTP listener is not available. A separate port is required for each application domain that uses notifications, which can reduce scalability and performance. • `Http`: Use HTTP through the kernel-mode HTTP listener in IIS 6.0, allowing a single port to be shared across all application domains. • `None`: No transport type is specified. *`timeout`*: The timeout in seconds for this notification. If not specified, the server default timeout is used.

The SqlDependency Properties

The three properties of the `SqlDependency` class are shown in Table 5.5.

TABLE 5.5. The Properties of the SqlDependency Class

Property	Description
`Id`	Returns a `String` representation of the GUID that identified this dependency. Used with the computer name, authentication type, and encryption type to define the `Id` property of the underlying `SqlNotificationRequest` object. Read-only.
`HasChanges`	Returns `True` if any of the results bound to this dependency have changed, or returns `False` otherwise. Read-only.

`InvalidationString`	Returns the information required by SQL Server to send back the notification, as an XML document fragment, such as:
	`<MachineAddress>`*address*`</MachineAddress>`
	`<AuthType>`*authentication_type*`</AuthType>`
	`<EncryptionType>`*encryption_type*`</EncryptionType>`
	`<Key>`*id*`</Key>`

The SqlDependency Method

There is a single method for the `SqlDependency` class, which can be used to bind a dependency to a command (see Table 5.6).

TABLE 5.6. The Method of the SqlDependency Class

Method	Description
`AddCommandDependency(`*command*`)`	Binds this dependency to the specified `SqlCommand` instance. More than one command can be added to a dependency if required. The dependency will raise a notification when any result fetched by any of the commands changes. No return value.

The SqlDependency Event

There is a single event raised by the `SqlDependency` instance when the values in the source data change or when any other event in the database (such as an error or a timeout) requires a notification to be sent to the client (see Table 5.7). Remember that a notification can be raised when the command is executed if there is an error or if the command is not valid for notifications, and that a notification is only sent once.

TABLE 5.7. The Event of the SqlDependency Class

Event	Description
`OnChanged`	Calls the specified event handler when a notifiable event occurs in the database. Passes an instance of a `SqlNotificationEventArgs` class to that event handler. The properties of the `SqlNotificationEventArgs` class are: `Info`: Indicates the reason for the notification. This is a value from the `SqlNotificationInfo` enumeration: • `Alter`: An underlying object related to the query was modified. • `Delete`: Data was changed by a `DELETE` statement. • `Drop`: An underlying object related to the query was dropped. • `Error`: An internal error occurred in the server. • `Insert`: Data was changed by an `INSERT` statement. • `Invalid`: A non-notifiable statement was provided (e.g., an `UPDATE` statement). • `Isolation`: An invalid isolation mode (e.g., `Snapshot`) was used when executing the query. • `Options`: The `SET` options were not correctly specified when subscribing for notifications. • `Query`: A statement for which notifications are enabled was provided (i.e., a suitable `SELECT` statement) and the data has changed. • `Restart`: The database server was started or restarted (notifications are sent when it starts). • `Truncate`: One or more tables were truncated. • `Update`: Data was changed by an `UPDATE` statement. `Source`: Indicates the item in the database that generated the notification or the reason it was generated. This is a value from the `SqlNotificationSource` enumeration: • `Data`: The source data has changed, for example, there was an `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, or similar statement executed that affected the source data. • `Database`: The state of a database that affects the source data has changed, for example, the database was detached or dropped. • `Environment`: The runtime environment was not compatible with notifications, for example, snapshot isolation or incompatible `SET` options were specified. • `Execution`: A runtime error occurred while executing the statement. • `Object`: A database object has changed, for example, a table containing the source data was modified or dropped from the database. • `Statement`: The statement being executed is not valid for notifications, for example, a non-notifiable `SELECT` statement or any statement other than a `SELECT` statement. • `System`: A system event occurred, for example, an internal error, the server was restarted, or lack of resources caused invalidation of data. • `Timeout`: The specified timeout period for the notification has expired. `Type`: Indicates whether this notification is due to an actual change in the data. This is a value from the `SqlNotificationType` enumeration: • `Change`: The notification occurred because the data has changed. • Subscribe: The notification occurred when the command was executed or when the notification request was being processed.

Using a SqlDependency

Specifying and requesting notifications using the `SqlDependency` class is just as easy as using the `SqlNotificationRequest` class. You can create the dependency and then bind it to the `Command` object. Or, as shown in Listing 5.8, you can specify the `Command` instance in the constructor for the `SqlDependency`. Before executing the command, you have to attach the event handler that will be called when a notification occurs. Then you use the command in the normal way to return a `DataReader` or to fill a `DataSet`.

LISTING 5.8. Creating a SqlDependency

```
' create a new Command object from an existing Connection instance
Dim SQL As String = "SELECT colA, colB FROM database.schema.table"
Dim con As New SqlConnection("connection-string")
Dim cmd As New SqlCommand(SQL, con)

' create a dependency and associate it with the command
Dim depend As New SqlDependency(cmd)

' subscribe to the dependency event
AddHandler depend.OnChanged, _
           New OnChangedEventHandler(AddressOf MyHandler)

' get the results and use them
Dim reader As SqlDataReader = cmd.ExecuteReader()
...
```

The difference between the `SqlDependency` and `SqlNotificationRequest` classes is that you don't have to poll the notifications queue when using the `SqlDependency` class. Your code can continue to execute, and the event handler you specified is called only when a notification occurs. In the event handler you can access the three properties, `Info`, `Source`, and `Type`, to see what caused the event. If the data has changed, you can invalidate any cached data and even perform a refresh to fetch the updated data (see Listing 5.9).

LISTING 5.9. Handling the OnChanged Event of a SqlDependency

```
' event handler routine
Sub MyHandler(sender As Object, args As SqlNotificationEventArgs)

  ' display the properties of the SqlNotificationEventArgs event...
  Console.WriteLine("OnChanged event raised...")
  Console.WriteLine("Reason why notification was raised: Info=" _
         & "SqlNotificationInfo{0}", args.Info.ToString())
```

```
  Console.WriteLine("Reason why notification was sent: Type=" _
          & "SqlNotificationType.{0}", args.Type.ToString())
  Console.WriteLine("Source that generated notification: Source=" _
          & "SqlNotificationSource.{0}", args.Source.ToString())

  ' probably invalidate the cached data, and maybe refresh it
  ' maybe even use the asynchronous feature new in ADO.NET 2.0!
  If args.Type = SqlNotificationType.Change Then
    ' invalidate cached data, and maybe refresh
    ...
  End If

End Sub
```

The samples you can download from our Web site at http://www.daveandal.net/books/7124/ include an example of using the `SqlDependency` class. This is in the `chapter-05` folder as a console application named `notifications-cachedependency`. The examples are available in both Visual Basic .NET and C#.

Using SqlDependency in ASP.NET Applications

ASP.NET has provided a `Cache` object for which dependencies can be specified since version 1.0. In version 2.0 of the Framework, the ASP.NET cache infrastructure is extended to work with the new `SqlDependency` class. This means that you can now specify `Cache` dependencies as being dependent on a result returned from SQL Server 2005. For example, when saving data in the `Cache` instance exposed for the application, you can use a `SqlDependency` instance to specify when the cache will be invalidated.

The code in Listing 5.10 creates a `DataSet` and fills it with rows, after binding a `SqlDependency` to the `Command` that the `DataSet` uses. This `SqlDependency` is then specified in the call to the `Insert` method of the `Cache` instance. When the notification of a change is received, the `DataSet` is invalidated and removed from the cache automatically.

LISTING 5.10. Using a SqlDependency in ASP.NET

```
' create a new Command object from an existing Connection instance
Dim SQL As String = "SELECT colA, colB FROM database.schema.table"
Dim con As New SqlConnection("connection-string")
Dim cmd As New SqlCommand(SQL, con)
```

continues

```
' create a dependency and associate it with the command
Dim depend As New SqlDependency(cmd)

' create a new DataSet and fill with rows
Dim ds As New DataSet()
Dim adapter As New SqlDataAdapter(cmd)
adapter.Fill(ds, "TableName")

' insert into cache to use as and when required
Cache.Insert("MyDataSet", ds, depend)
...
```

Of course, this means that the code will have to check whether the `DataSet` is in the cache before attempting to use it later and recreate it if it's not there.

Page-Level Output Caching Dependencies

SQL query notifications can also be used to invalidate a response generated by an ASP.NET page. An attribute is specified in the `@OutputCache` directive of the page to denote that the cached response is dependent on all `SqlClient` commands used to retrieve and generate the content for the page. If the results of any of these commands change, the page is automatically removed from the cache.

When ASP.NET sees `SqlDependency="CommandNotification"` in the `@OutputCache` page directive, it automatically creates a `SqlDependency` under the covers. All commands created within the page register with that `SqlDependency`. Then, if any one of the results becomes invalid due to changes in the database or to other notifiable events on the database server, the ASP.NET page output is invalidated on the Web server. The next request will then cause fresh and up-to-date copies of the data to be fetched from the database. Here's an example of such a directive:

```
<%@ OutputCache Duration="300" VaryByParam="none"
    SqlDependency="CommandNotification" %>
```

The same kind of process can also be implemented for individual commands in code at runtime, rather than being automatically applied to all commands through the `@OutputCache` directive. The `AddCacheDependency` method adds the `SqlDependency` to the response (output) cache, causing the cached data to be invalidated when the data retrieved by the `Command` changes in the database (see Listing 5.11).

LISTING 5.11. Using a SqlDependency to Invalidate Cache Entries

```
...
' create a dependency and associate it with the command as before
Dim depend As New SqlDependency(cmd)
...

' set any other values required for response cache
Response.Cache.SetExpires(DateTime.Now.AddSeconds(300))
Response.Cache.VaryByParams("None") = True
Response.Cache.SetCacheability(HttpCacheability.Public)
Response.Cache.SetValidUntilExpires(True)

' add the SqlDependency
Response.AddCacheDependency(depend)
...
```

Remember that you can set the `Boolean` property `NotificationAutoEnlist` of a `Command` instance to specify whether that command will automatically enlist the notifications service in an ASP.NET page that uses `SqlDependency`.

> See the companion book *ASP.NET v. 2.0—The Beta Version* (Boston, MA: Addison-Wesley, 2005, ISBN 0-321-25727-8) for a more detailed look at how object-level and output (page-level) caching works in ASP.NET 2.0.

SQL Server User-Defined Types

In conjunction with the changes taking place to SQL Server in the forthcoming release, it's now possible to execute managed code—in any of the supported .NET languages—on the Web or application server, the client, and within SQL Server itself. This is due to the Common Language Runtime (CLR) being embedded within SQL Server 2005, allowing managed code to run within the database itself. One opportunity this raises is an extension of the data types that can be stored in a SQL Server database. This is just one of the many extensions to SQL Server; more are covered in Chapter 6.

User-defined data types can easily be created in .NET by defining a structure or class that exposes the individual items of data as properties. For example, consider a complex number; this cannot be stored directly within the standard SQL types. Within SQL Server 2005, however, you

could create a structure called `ComplexNumber` and use it as the data type for the column. Likewise, you could create data types that contain multiple pieces of information: a `Point` to combine x- and y-coordinates, `Longitude` and `Latitude` to combine degrees, minutes, seconds, and hemisphere. These could easily be stored as separate columns, but they would need to be recombined when required. This may not seem such a burden, but consider dates, which store three sets of information: a day, a month, and a year. You certainly wouldn't want every date to be stored as three separate columns.

When to Use UDTs

The idea of using UDTs does not mean that you should store your classes directly in SQL Server as UDTs. For example, consider classes such as `User`, `Product`, and `Invoice`. These occur frequently as classes to define the entities of a business, but they are not suitable as UDTs. The reason is simple—SQL Server is not an object-oriented database. These types of classes are business abstractions, not data types. For data to be considered as a UDT, it should be a **scalar** object (a value that can be represented in a single dimension) that requires more than one field to store it.

This definition of a scalar value is restrictive and excludes defining types such as `Point`, `Longitude`, and `Latitude` as UDTs. One view is that a UDT is suitable if the "between" rule can be applied. That is, is it easy to define one value as being between two other values? For example, with dates this is clearly so because it is easy to determine whether one date lies between two others. For points the answer becomes vague; technically, a point is between other points only when on a line—a scalar object because a line is single dimensional. If the middle point is not directly on the line between two other points, is it still classified as between? Strictly speaking, three points like this define a triangle; the middle point is not between the others and therefore a point shouldn't be used for a UDT. However, using this rigid rule to define what can be considered a UDT is restrictive. Using SQL Server 2005 to store shape data for a graphical application is not something new, and using UDTs for point storage is a natural extension.

Creating a UDT

Creating a UDT is simply a matter of creating a class or structure and decorating it with certain attributes to identify it to SQL Server. Visual Studio 2005 makes this easy by supplying a SQL Server Project template, providing you with the opportunity to connect to a SQL Server database. Then from the Project menu you can select Add User-Defined Type, which provides

a base class for your type. This class is a standard .NET class, but for it to be available as a UDT, it must implement the interfaces, methods, and properties defined in Table 5.8.

TABLE 5.8. Class Requirements for a User-Defined Type

Task	Reason
Implement the `INullable` interface	Types in SQL Server can contain `null` values, so the UDT must be able to contain a `null` value.
Implement the `INullable.IsNull` property	This is called to identify whether or not the UDT contains a `null` value.
Override the `ToString` method	`ToString` is called automatically when a textual representation of the UDT is required. In this method, you convert the internal representation into a human-readable form.
Implement the `Null` property	This property defines a `null` value for the UDT, typically by setting the internal values to some known value that can be used to represent `null` (e.g., a `Boolean` flag).
Implement the `Parse` method	This method is used to convert string values into the internal representation. It will be called when values are inserted into the column.

In addition to the required implementation of methods and properties, you can also define custom ones. To make this clearer, consider the `Point` structure as a UDT in which we want to store x- and y-coordinates. We'd want to implement `X` and `Y` as properties to allow clients to access the individual parts of the type, as shown in Listing 5.12.

LISTING 5.12. The Point User-Defined Type

```
Imports System
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports System.Runtime.Serialization

<Serializable(), SqlUserDefinedTypeAttribute(Format.Native)> _
Public Structure Point
  Implements INullable

  Private _isNull As Boolean
  Private _x As Double
  Private _y As Double
```

continues

```
Public Sub New(ByVal x As Double, ByVal y As Double)
  _x = x
  _y = y
End Sub

Public ReadOnly Property IsNull() As Boolean _
  Implements INullable.IsNull
    Get
      Return _isNull
    End Get
End Property

Public Overrides Function ToString() As String

  If Me.IsNull Then
    Return "NULL"
  Else
    Return Me._x & ":" & Me._y
  End If

End Function

Public Shared Function Parse(ByVal s As SqlString) As Point

  If s.IsNull OrElse s.ToString() = "NULL" Then
    Dim pt As New Point()
    pt._isNull = True
    Return pt
  Else
    ' parse the input string here to separate out points
    Dim xy() As String = s.ToString().Split(":")

    If xy.Length <> 2 Then
      Throw New Exception("Point must be supplied as x:y")
    End If

    ' construct a new point from the given coordinates
    Dim pt As New Point()
    pt.X = Convert.ToDouble(xy(0))
    pt.Y = Convert.ToDouble(xy(1))

    Return (pt)
  End If

End Function

Public Shared ReadOnly Property Null() As Point

  Get
    Dim pt As New Point
    pt._isNull = True
    Return (pt)
```

```
    End Get

  End Property

  Public Property X() As Double

    Get
      Return (Me._x)
    End Get

    Set(ByVal Value As Double)
      _x = Value
    End Set

  End Property

  Public Property Y() As Double

    Get
      Return (Me._y)
    End Get

    Set(ByVal Value As Double)
      _y = Value
    End Set

  End Property

End Structure
```

Internally, the x- and y-coordinates are stored as private variables of type `Double`. There is also a private `Boolean` flag to identify whether the type is null; the `IsNull` property simply returns this value. The `ToString` method returns the internal types formatted as a string, with the x- and y-coordinates separated by a colon. The `Parse` method provides the opposite service—accepting a string that contains the colon-separated x- and y-coordinates (or the string `"NULL"`) and converting that into the internal representation.

As you can see, this is a simple structure that obeys some standard rules, and you can add additional properties and methods as required.

One requirement not yet mentioned is the decoration of the structure with attributes:

```
<Serializable(), SqlUserDefinedTypeAttribute(Format.Native)>
```

The `Serializable` attribute indicates that the structure can be serialized, a requirement for SQL Server 2005 to store the data internally. The

`Format.Native` parameter of the `SqlUserDefinedTypeAttribute` indicates that serialization should take place by the native method—this means that SQL Server 2005 should control the serialization. One important point to note is that when using `Native` serialization, the internal types can be only value types. These types are `bool`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `SqlByte`, `SqlInt16`, `SqlInt32`, `SqlInt64`, `SqlDateTime`, `SqlSingle`, `SqlDouble`, `SqlMoney`, and `SqlBoolean`. Notice that `string` is not among this list—the `string` type is a reference type and is therefore not allowed when using `Native` serialization. If you require a type other than these, you need to serialize the type yourself. For more details, see the SQL Server 2005 and Visual Studio 2005 documentation. Additionally, Microsoft recommends that UDTs should be structures, not classes. This is mostly due to structures being value types, which are cheaper to construct, don't involve boxing, and have less overhead on the garbage collector. This is particularly important in the SQL CLR because a large number of UDT instances will be created when querying over large tables with UDT columns.

Installing a UDT into SQL Server 2005 with Visual Studio 2005

Once you've created a UDT, you need to install it into SQL Server 2005. All custom types are compiled into an assembly that is loaded into SQL Server. This means that once deployed into SQL Server 2005, the original code is not required; the database is self-contained.

Using Visual Studio 2005 to deploy the UDT is simply a matter of selecting the Deploy Solution item from the Build menu. The assembly is compiled and loaded into SQL Server 2005, and the UDT is created. You can then create other types, such as tables, that depend upon the UDT.

Installing a UDT into SQL Server 2005 Manually

Installing a UDT manually requires more steps. First you have to install the assembly into SQL Server 2005:

```
CREATE ASSEMBLY PointAssembly
FROM "c:\SQLProjects\Point.dll"
```

This creates the assembly within SQL Server but doesn't create the type. So you then enter:

```
CREATE TYPE Point
EXTERNAL NAME PointAssembly.Point
```

If the structures are contained within a namespace, you must also include the namespace when creating the type:

```
CREATE TYPE Point
EXTERNAL NAME PointAssembly.PointNamespace.Point
```

The point can now be used within tables, as demonstrated in this example:

```
CREATE TABLE ShapeData
(
  ShapeID int NOT NULL,
  Position Point
)

INSERT INTO ShapeData
VALUES(1, '2:4')

INSERT INTO ShapeData
VALUES (2, CAST('2:4' As Point))
```

Notice that the second column is the point and that the data has been entered as a string with the x and y values separated by a colon—the format required by the `Parse` method.

Accessing a UDT

Accessing a UDT from clients is much like accessing any other data type but with one exception: The client must have a reference to the assembly containing the type. For example, SQL Server Management Studio (SSMS) is a client application so you cannot simply run this query:

```
SELECT * FROM ShapeData
```

The client application has no knowledge of the UDT and therefore cannot return a column of that type, so the query results in the following error:

```
An error occurred while executing batch. Error message is: File or
assembly name 'Point, Version=1.0.1710.21243, Culture=neutral,
PublicKeyToken=null', or one of its dependencies, was not found.
```

You can, however, access methods of the type directly, making the following query possible:

```
SELECT ShapeID, Position.ToString() FROM ShapeData
```

Because the `ToString` method is executed within the database, which does know about the type, only text is returned to the client. Remember that the

term *client* refers not to a physical entity but to the application requesting the UDT. So an ASP.NET application would be a client even if it resided physically on a server.

Making a UDT Available to Clients

To be made available to clients, a UDT needs to be referenced in the same way that other user types (such as custom classes) are referenced—by the client application directly (a private assembly) or installed into the Global Assembly Cache (GAC) and referenced from there. For Visual Studio 2005 this is simply a matter of using the Add Reference item from the Project menu.

Using the GAC avoids the explicit reference and also means that the UDT becomes available to SSMS, enabling the `SELECT *` type of query to work.

Accessing a UDT in SQL

The UDT column can be used like any other column when used from client applications—within SQL statements (either directly in the Query window in the SSMS or from within stored procedures), or in SQL on the client. Methods and properties can be called directly, as shown in Listing 5.13.

LISTING 5.13. Directly Accessing UDTs from SQL

```
SELECT Position FROM ShapeData

SELECT Position.X FROM ShapeData

DECLARE @pos Point
SET @pos = CONVERT(Point, '5:3')
PRINT @pos.X
PRINT @Pos.Y
```

Notice that `Point` behaves like any other data type, so you can use it to declare local variables, to write `CONVERT` statements, and so on.

Accessing a UDT via a DataReader

Within client ADO.NET code, the UDT acts just like any other .NET type. Consider the code fragment shown in Listing 5.14, where the rows from a query are simply iterated through with a reader, and `ToString` is called on the UDT column.

LISTING 5.14. Accessing a UDT from Client Code

```
Dim connect As String = _
    "Server=localhost; Database=AWL; Trusted_Connection=True"
Dim sql As String = "SELECT * FROM ShapeData"

Dim conn As New SqlConnection(connect)
Using (conn)

  SqlCommand cmd = new SqlCommand(sql, conn)
  conn.Open()

  SqlDataReader rdr = cmd.ExecuteReader()

  While rdr.Read()
    Console.WriteLine("Point: " & rdr("Position").ToString())
  End While

End Using
```

You can call other methods and properties in addition to `ToString`. You can also reference the column as a strong type, as shown in Listing 5.15.

LISTING 5.15. Strongly Typed Access to a UDT

```
...
Dim pt As Point
While rdr.Read()
  pt = CType(rdr("Position"), Point)
  Console.WriteLine("X=" & pt.X.ToString() & _
                    "Y=" & pt.Y.ToString())
End While
...
```

Accessing a UDT via a DataSet

The process of accessing a UDT stored in a `DataSet` is much the same as for one stored in a `DataReader`. For example, consider Listing 5.16, where `myDataSet` contains data from the ShapeData table.

LISTING 5.16. Accessing a UDT Column from a DataSet

```
Dim pt As Point = myDataSet.Tables(0).Rows(0)("Position")
Console.WriteLine("X=" & pt.X.ToString() & _
                  "Y=" & pt.Y.ToString())
```

Using UDTs in Query Parameters

When you work with UDTs in ADO.NET, you'll come across the situation where you need to use a UDT as a parameter to a SQL statement or stored procedure. The `SqlParameter` class has been modified in ADO.NET 2.0 to support a new constructor and new methods that allow UDTs to be used. They hinge on the way that a UDT is identified.

The identification is done by using the type name as the value for a new property on the `SqlParameter` class named `UdtTypeName`. The new addition to the `SqlDbType` enumeration, `Udt`, is specified for the `DbType` property to identify the data type as being a UDT. For example:

```
Dim param As New SqlParameter()
param.DbType = SqlDbType.Udt
param.UdtTypeName = "dbo.Point"
```

The constructors for a `SqlParameter` accept a value from the `SqlDbType` enumeration, so a parameter for a `Point` type can alternatively be created using the following code:

```
param = oCmd.Parameters.Add("@pos", SqlDbType.Udt)
param.UdtTypeName = "dbo.Point"
```

Therefore, you could create a new `Point` UDT instance and pass it to a SQL statement or a stored procedure as a UDT parameter, using code like that shown in Listing 5.17.

LISTING 5.17. Using a UDT as a Query Parameter

```
' create a new Command object from an existing Connection instance
' SQL statement updates a UDT position column value in a table
Dim oCmd As New SqlCommand("UPDATE ShapeData " & _
                           "SET Position = @pos " & _
                           "WHERE ID = @RowID", oConn)

Dim oParam As SqlParameter

' add an Integer parameter for the RowID
oParam = oCmd.Parameters.Add("@RowID", SqlDbType.Int)
oParam.Value = 1

' now add a UDT parameter of type Vehicle
param = oCmd.Parameters.Add("@Pos", SqlDbType.Udt)
param.UdtTypeName = "dbo.Point"

' set the parameter's value using the constructor for the Point class
param.Value = New Point(5, 5)

' execute the command
Dim rowsAffected As Integer = oCmd.ExecuteNonQuery()
```

Interface Changes to Support UDTs

Several changes have been made to the classes in the `SqlClient` namespace to support UDTs. These include changes to the `SqlParameter` and `SqlDataReader` classes, and a class called `SqlMetaData` that is added to SQL Server 2005 as a general extension.

Changes to the SqlParameter Class

The `SqlParameter` class allows a UDT to be specified as a parameter to a stored procedure or SQL statement. There are two new constructors, two new properties, and changes to the way that a couple of the existing properties work to support UDTs. The new constructors are documented in Table 5.9.

TABLE 5.9. The Constructors for the SqlParameter Class for UDT Support

Constructor	Description
`SqlParameter(`*`name, dbtype, size, direction, nullable, precision, scale, source-col, source-version, value, offset, compare-info, locale, database, schema, udt-type-name`*`)`	All the parameters to this constructor are the same as existing constructors with two exceptions. The *`dbtype`* is a value from the `SqlDbType` enumeration, which now contains the value `Udt`. The new *`udt-type-name`* parameter takes a `String` that contains the name of the UDT type. If `SqlDbType.Udt` is specified for *`dbtype`*, *`udt-type-name`* must be a non-`null` value that is the type. If *`dbtype`* is any other value, *`udt-type-name`* must be `null`.
`SqlParameter(`*`meta-data`*`)`	This takes a `SqlMetaData` instance that fully describes the data type for the parameter. It can be used to create a parameter of other types as well, but for UDT parameters the `SqlMetaData` instance must include the UDT type name.

The two new properties for the `SqlParameter` are `UdtTypeName` and `MetaData`. The two properties that have been updated are `SqlDbType` and `Value` (see Table 5.10).

TABLE 5.10. The Properties of the SqlParameter Class for UDT Support

Property	Description
`UdtTypeName`	Sets or returns the UDT type as a `String`. This should be a two-part dotted name such as `dbo.Point`.
`MetaData`	Returns a `SqlMetaData` instance that fully describes the metadata for the parameter, or sets it using a `SqlMetaData` instance. To change the parameter type, you must create a new `SqlMetaData` instance and assign it to the parameter. You cannot change individual values for the metadata.
`SqlDbType`	Sets or returns the parameter type using values from the `SqlDbType` enumeration, which now includes the value `Udt` for a UDT. If this is set, the `UdtTypeName` must be set to a UDT type name. If any other value is used for this property, the `UdtTypeName` property must be `null`.
`Value`	Sets or returns the value of the parameter. When the parameter is a UDT, this property accepts or returns an instance of the UDT. Note that the value is a reference type, so changes to the values of this instance of the UDT will be reflected in the parameter value.

Changes to the SqlDataReader Class

The `SqlDataReader` class exposes one new method to return metadata about a column, and six of the existing methods have been updated to support the use of UDTs (see Table 5.11).

TABLE 5.11. The Methods of the SqlDataReader Class for UDT Support

Method	Description
`GetSqlMetaData(`*index*`)`	Returns the `SqlMetaData` instance that describes the column at the specified index. New in this release.
`GetDataTypeName(`*index*`)`	This method has been updated to return the three-part type name for the specified column if the column contains a UDT.
`GetFieldType(`*index*`)`	This method has been updated to return a `System.Type` instance that represents the UDT type for the specified column. Calling this method may trigger an assembly download if the assembly for the UDT is not present locally and the option to download assemblies is enabled.

TABLE 5.11. The Methods of the SqlDataReader Class for UDT Support (*continued*)

Method	Description

`GetValue(index)`
`GetSqlValue(index)`
`GetValues(values-array)`
`GetSqlValues(values-array)`
	These methods have been updated to return a new instance of the UDT with the values for the specified column if it contains a UDT. Calling them may trigger an assembly download if the assembly for the UDT is not present locally and the option to download assemblies is enabled. This will block the `DataReader` even when asynchronous execution is taking place.

The SqlMetaData Class

The `SqlMetaData` class introduced in this release of SQL Server can be used to hold metadata about a column in a `DataReader` or the value in a `SqlParameter`, and for other purposes within the `SqlClient` namespace of the Framework. It exposes a field named `UdtTypeName` (see Table 5.12).

TABLE 5.12. The SqlMetaData Class UdtTypeName Field

Field	Description
`UdtTypeName`	Returns the metadata for a column or a parameter. In the beta 2 release, this is the three-part dotted name of the UDT as defined in SQL Server. In future releases, the property will expose a `System.Type` instance instead. Read-only.

SUMMARY

In this chapter we examined some of the new features for client access to SQL Server 2005. We started with a look at MARS, which allows you to have more than one results set active on the same data store connection. Although this at first looks like a way to save resources because fewer connections could be required, the use of multiple readers, for example, could lead to potential performance problems if these readers are not closed appropriately. Therefore, MARS is best suited to batch or complex data processing scenarios.

We then looked at Query Notifications, through which client applications can register to be notified when data changes. This solves the prob-

lem of stale data being displayed and gets around the existing methods of keeping data up-to-date, such as constant polling.

Finally, we looked at the CLR within SQL Server 2005 and how its use allows the type system to be extended by the addition of user-defined types. These allow you to add complex data types to the existing type system, giving more flexibility in data storage.

Now it's time to look at the CLR in SQL Server 2005 in more depth.