4
In-Process Data Access

WHETHER YOU ACCESS data from the client, middle tier, or server, when you're using SQL Server and the .NET Framework, you use the `SqlClient` data provider. Your data access code is similar regardless of its location, but the .NET Framework 2.0 version of `SqlClient` contains code to encapsulate differences when you're programming inside SQL Server and optimize the in-server programming environment.

Programming with SqlClient

Accessing data from outside SQL Server entails connecting to SQL Server through a network library and a client library. When you use the .NET Framework with SQL Server, you use System.Data.dll as the client library and the ADO.NET programming model. ADO.NET is a provider-based model, similar in concept to ODBC, OLE DB, and JDBC. The model uses a common API (or a set of classes) to encapsulate data access; each database product has its own provider. ADO.NET providers are known as *data providers,* and the data provider for SQL Server is `SqlClient`. The latest release of `SqlClient`, installed with .NET Framework 2.0, includes new client-side functionality to take advantage of new features in SQL Server 2005. In addition, `SqlClient` contains extensions to allow ADO.NET code to be used inside the database itself. Though T-SQL is usually preferred when a stored procedure, user-defined function, or trigger accesses database data, you can also use ADO.NET when writing procedural code in the

.NET Framework. The programming model when using `SqlClient` in .NET Framework stored procedures is similar to client-side code but in-database access is optimized because no network libraries are needed. Let's start by writing some simple client database code and then convert it to run on the server.

Simple data access code is very similar regardless of the programming model used. To summarize, using ADO.NET and `SqlClient` as an example:

1. Connect to the database by instantiating a `SqlConnection` class and calling its `Open` method.
2. Create an instance of a `SqlCommand` class. This instance contains a SQL statement or procedure name as its `CommandText property`. The `SqlCommand` is associated with the `SqlConnection`.
3. Execute the `SqlCommand`, and return either a set of columns and rows called `SqlDataReader` or possibly only a count of rows affected by the statement.
4. Use the `SqlDataReader` to read the results, and close it when finished.
5. Dispose of the `SqlCommand` and `SqlConnection` to free the associated memory, network, and server resources.

The ADO.NET code to accomplish inserting a row into a SQL Server table would look like Listing 4-1.

LISTING 4-1: Inserting a row using SqlClient from the client

```
// Code to insert data from client
//   See chapter 14 for an implementation of
//   the GetConnectionStringFromConfigFile method.
string connStr = GetConnectionStringFromConfigFile();
SqlConnection conn = new SqlConnection(connStr);
conn.Open();
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "insert into test values ('testdata')";
int rows_affected = cmd.ExecuteNonQuery();
cmd.Dispose();
conn.Dispose();
```

The previous ADO.NET code ignored the fact that an exception might cause the execution of `cmd.Dispose` or `conn.Dispose` to be skipped. The preferred and simple way to prevent this from happening is to use the `using` syntax in C#. One or more object instance declarations are followed by a block of code. The `Dispose` method is called automatically at the end of the code block. We'll be using the `using` construct a lot in the code in this book. Rewritten using this syntax, the code above would look like Listing 4-2.

LISTING 4-2: Inserting a row using SqlClient from the client, C# using construct

```
//code to insert data from client
string connStr = GetConnectionStringFromConfigFile();
using (SqlConnection conn = new SqlConnection(connStr))
using (SqlCommand cmd =
  new SqlCommand("insert into test values ('testdata')", conn))
{
  conn.Open();
  int rows_affected = cmd.ExecuteNonQuery();
}  // Dispose called on cmd and conn here
```

Other classes in a typical ADO.NET data provider include a transaction class (`SqlTransaction`) to tie Connections and Commands to a database transaction, a parameter collection (`SqlParameterCollection`) of parameters (`SqlParameter`) to use with parameterized SQL queries or stored procedures, and specialized Exception and Error classes (`SqlException`, `SqlErrorCollection`, `SqlError`) to represent processing errors. `SqlClient` includes all the typical classes; Figure 4-1 shows the object model.

The same basic model is used inside the server to access data in .NET Framework stored procedures. It's familiar to ADO.NET programmers, and using it inside the server makes it easy for programmers to use their

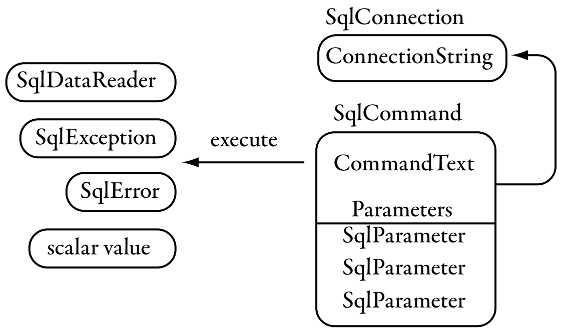

FIGURE 4-1: The SqlClient provider object model (major classes only)

existing skills to write procedures. The big difference is that when you're writing a .NET Framework procedure, you're already inside the database. No explicit connection is needed. Although there is no network connection to the database, there is a `SqlConnection` instance. The difference is in the connection string. Outside the database, the connection string should be read from a configuration file and contains items like the SQL Server instance to connect to (server keyword), the SQL Server login (either `User ID` and `Password` keywords or `Integrated Security=true`), and the initial database (database keyword). The connection string that indicates to `SqlClient` that we're already inside the database and the provider should just use the existing database context contains only the keyword `"context connection=true"`. When you specify `"context connection=true"`, no other connection string keyword can be used. Listing 4-3 is the same code as above but executing inside a .NET Framework stored procedure.

LISTING 4-3: Inserting a row using SqlClient in a SQLCLR stored procedure

```
//code to insert data in a stored procedure
public static void InsertRowOfTestData()
{
  string connStr = "context connection=true";
  using (SqlConnection conn = new SqlConnection(connStr))
  using (SqlCommand cmd =
    new SqlCommand("insert into test values ('testdata')", conn))
  {
    conn.Open();
    int rows_affected = cmd.ExecuteNonQuery();
  }
}
```

Note that this code is provided as a stored procedure only to explain how to access data on the server. Not only is the code faster as a Transact-SQL (T-SQL) stored procedure but also SQL Server will check the SQL statements for syntactic correctness at `CREATE PROCEDURE` time. This is not the case with the .NET Framework stored procedure above. When you execute SQL statements by using `SqlCommand`, it's the equivalent of using `sp_executesql` (a system-supplied store procedure for dynamic string execution of commands) inside a T-SQL stored procedure. There is the same potential for SQL injection as with `sp_executesql`, so don't execute commands whose `CommandText` property is calculated by using input parameters passed in by the procedure user.

dotNET_C04 4/6/06 11:51 AM Page 115

This code is so similar to the previous client-side code that if we knew whether the code was executing in a stored procedure on the server or on the client, we could use the same code, changing only the connection string. But a few constructs exist only if you are writing server-side code. Enter the `SqlContext` class.

Context: The SqlContext Class

The `SqlContext` class is one of the new classes that are available only if you're running inside the server. When a procedure or function is executed, it is executed as part of the user's connection. Whether that user connection comes from ODBC, ADO.NET, or T-SQL doesn't really matter. You are in a connection that has specific properties, environment variables, and so on, and you are executing within that connection; you are in the context of the user's connection.

A command is executed within the context of the connection, but it also has an execution context, which consists of data related to the command. The same goes for triggers, which are executed within a trigger context.

Prior to SQL Server 2005, the closest we came to being able to write code in another language that executed within the process space of SQL Server was writing extended stored procedures. An *extended stored procedure* is a C or C++ DLL that has been catalogued in SQL Server and therefore can be executed in the same manner as a "normal" SQL Server stored procedure. The extended stored procedure is executed in process with SQL Server and on the same Windows thread[1] as the user's connection.

Note, however, that if you need to do any kind of database access—even within the database to which the user is connected—from the extended stored procedure, you still need to connect to the database explicitly through ODBC, OLE DB, or even DBLib exactly as you would do from a client, as Figure 4-2 illustrates. Furthermore, when you have created the connection from the procedure, you may want to share a common transaction lock space with the client. Because you now have a separate connection, you need to

1. Strictly speaking, thread or fiber, depending on the setting in the server. See Chapter 2 for information about fiber mode.

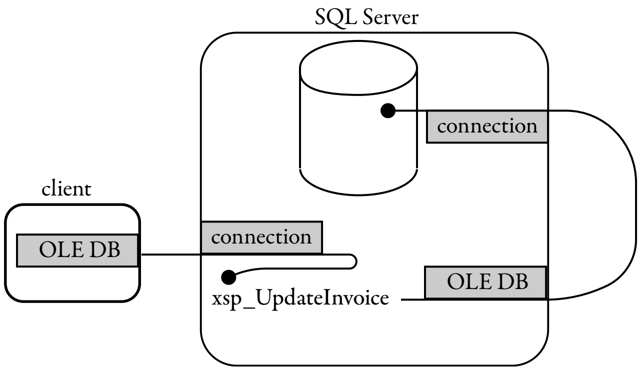

FIGURE 4-2: Connections from extended stored procedures

ensure explicitly that you share the transaction lock space by using the `srv_getbindtoken` call and the stored procedure `sp_bindsession`.

In SQL Server 2005, when you use the .NET Framework to write procedures, functions, and triggers, the `SqlContext` is available. The original program can now be rewritten in Listing 4-4 so that the same code works either on the client/middle tier or in the server if it's called as part of a stored procedure using the `SqlContext` static `IsAvailable` property.

LISTING 4-4: Using IsAvailable to determine whether the code is running on the server

```
// other using statements elided for clarity
using System.Data.SqlClient;
using Microsoft.SqlServer.Server; // for SqlContext

public static void InsertRowOfTestData2()
{
  string connStr;
  if (SqlContext.IsAvailable)
    connStr = "context connection=true";
  else
    connStr = GetConnectionStringFromConfigFile();
  // the rest of the code is identical
  using (SqlConnection conn = new SqlConnection(connStr))
  using (SqlCommand cmd =
    new SqlCommand("insert into test values ('testdata')", conn))
  {
    conn.Open();
    // The value of i is the number of rows affected
    int i = cmd.ExecuteNonQuery();
  }
}
```

You can see the SqlContext as a helper class; static read-only properties that allow you to access the class encapsulate functionality that exists only on the server. These properties are shown in Table 4-1.

Using SqlContext is the only way to get an instance of the classes in Table 4-1; you cannot create them by using a constructor (New in Visual Basic .NET). You can create the other classes that are part of the SqlClient provider in the same way that you normally would create them if used from an ADO.NET client. Some of the classes and methods in SqlClient act a little differently if you use them on the server, however.

SQLCLR stored procedures can do data access by default, but this is not the case with a SQLCLR user-defined function. As was discussed in the previous chapter, unless DataAccessKind or SystemDataAccessKind is set to DataAccessKind.Read/SystemDataAccessKind.Read, any attempt to do data access using the SqlClient provider will fail. Even if DataAccessKind is set to DataAccessKind.None (the default), however, SqlContext.Is-Available returns true. SqlContext.IsAvailable is an indication of whether you're running in the server, rather than whether data access is permitted.

By now, you may be wondering: If some of the managed classes are calling into SQL Server, does that mean that the internals of SQL Server are managed as well, and if not, are interoperability calls between managed and native code space happening? The answers are no and yes. No, the internals of SQL Server are not managed; Microsoft did not rewrite the whole of SQL Server in managed code. And yes, interoperability calls happen. The managed classes are making Platform Invoke (PInvoke) calls against the executable of SQL Server, sqlservr.exe, as shown in Figure 4-3, which exposes a couple dozen methods for the CLR to call into.

TABLE 4-1: SqlContext Static Properties

Property	Return Value
IsAvailable	Boolean
WindowsIdentity	System.Security.Principal.WindowsIdentity
Pipe	Microsoft.SqlServer.Server.SqlPipe
TriggerContext	Microsoft.SqlServer.Server.SqlTriggerContext

dotNET_C04 4/6/06 11:51 AM Page 118

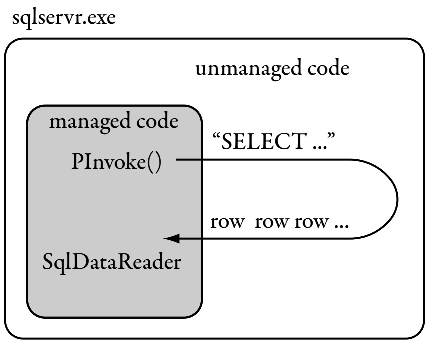

FIGURE 4-3: Interop between .NET framework and SQL Server code in process

When you read this about interop, you may become concerned about performance. Theoretically, a performance hit is possible, but because SQL Server hosts the CLR (as discussed in Chapter 2), and the `SqlClient` provider runs in process with SQL Server, the hit is minimal. In the last sentence, notice that we said *theoretically.* Remember that when you execute CLR code, you will run machine-compiled code, which is not the case when you run T-SQL. Therefore, for *some* code executing in the CLR, the result may be a performance improvement compared with pure T-SQL code.

Now that we have discussed the `SqlContext` class, let's see how we go about using it.

Connections

As already mentioned, when you are at server side and a client executes, you are part of that client's connection context, which in SQL Server 2005 is exposed by using a special connection string. The `SqlConnection` object exposes the public methods, properties, and events listed in Table 4-2. (Note that the table doesn't show members inherited from `System.Object`.)

You can create only one `SqlConnection` at a time with the special `"context connection=true"` string. Attempting to create a second `SqlConnection` instance will fail, but you can create an "internal" `SqlConnection` and another external `SqlConnection` back to the same instance using an ordinary connection string. Opening this additional `SqlConnection`

dotNET_C04 4/6/06 11:51 AM Page 119

TABLE 4-2: Public Members of SqlConnection

Name	Return Value/Type	Member Type
`Constructor`		Constructor
`Constructor(String)`		Constructor
`BeginTransaction()`	`SqlTransaction`	Method
`BeginTransaction (IsolationLevel)`	`SqlTransaction`	Method
`BeginTransaction (IsolationLevel, String)`	`SqlTransaction`	Method
`BeginTransaction(String)`	`SqlTransaction`	Method
`ChangeDatabase(String)`	`void`	Method
`ChangePassword (String, String)`	`void`	Static Method
`ClearAllPools`	`void`	Static Method
`Close()`	`void`	Method
`CreateCommand()`	`SqlCommand`	Method
`EnlistDistributedTransaction (ITransaction)`	`void`	Method
`EnlistTransaction (Transaction)`	`void`	Method
`GetSchema()`	`DataTable`	Method
`GetSchema(String)`	`DataTable`	Method
`GetSchema(String, String[])`	`DataTable`	Method
`Open()`	`void`	Method
`ResetStatistics`	`void`	Method
`RetrieveStatistics`	`Hashtable`	Method
`ConnectionString`	`String`	Property

(*Continued*)

TABLE 4-2: Public Members of SqlConnection (*Continued*)

Name	Return Value/Type	Member Type
`ConnectionTimeout`	`Int32`	Property
`Database`	`String`	Property
`DataSource`	`String`	Property
`FireInfoMessageOnUserErrors`	`Boolean`	Property
`PacketSize`	`Int32`	Property
`ServerVersion`	`String`	Property
`State`	`String`	Property
`StatisticsEnabled`	`Boolean`	Property
`WorkStationId`	`String`	Property
`InfoMessage`	`SqlInfoMessage EventHandler`	Event

will start a distributed transaction, however,[2] because you have multiple SPIDs (SQL Server sessions) possibly attempting to update the same data. There is no way to knit the two sessions together through the ADO.NET API into a single local transaction, however, as you can in an extended stored procedure with `sp_bindtoken`. You can call the `SqlConnection`'s `Close()` method and reopen it, if you like, although it's unlikely that you ever actually need to do this. Keeping the `SqlConnection` open doesn't use any additional resources after you originally refer to it in code.

Although the same `System.Data.SqlClient.SqlConnection` class is used for both client and server code, some of the features and methods will not work inside the server:

- `ChangePassword` method
- `GetSchema` method

2. Technically, you can avoid a distributed transaction by using `"enlist=false"` in the connection string of the new `SqlConnection`. In this case, the second session does not take part in the context connection's transaction.

- Connection pooling and associated parameters and methods
- Transparent failover when database mirroring is used
- Client statistics
- `PacketSize`, `WorkstationID`, and other client information

Commands: Making Things Happen

The `SqlClient` provider implements the `SqlCommand` class to execute action statements and submit queries to the database. When you have created your connection, you can get the command object from the `CreateCommand` method on your connection, as the code in Listing 4-5 shows.

LISTING 4-5: Create a command from the connection object

```
//get a command through CreateCommand
SqlConnection conn = new SqlConnection("context connection=true");
SqlCommand cmd = conn.CreateCommand();
```

Another way of getting to the command is to use one of the `SqlCommand`'s constructors, which Listing 4-6 shows.

LISTING 4-6: Using SqlCommand's constructor

```
//use constructor that takes a CommandText and Connection
string cmdStatement = "select * from authors";
SqlConnection conn = new SqlConnection("context connection=true");
SqlCommand cmd = new SqlCommand(cmdStatement, conn);
```

We have seen how a `SqlCommand` is created; now let's look at what we can do with the command. Table 4-3 lists the public methods, properties, and events. (The table doesn't show public members inherited from `System.Object` or the extra asynchronous versions of the execute-related methods.)

For those of you who are used to the `SqlClient` provider, most of the members are recognizable, but as with the connection object when used inside SQL Server, there are some differences:

- The new asynchronous execution methods are not available when running on the server.
- You can have multiple `SqlCommand`s associated with the special context connection, but cannot have multiple active `SqlDataReader`s at

TABLE 4-3: Public Members of SqlCommand

Name	Return Value/Type	Member Type
`Constructor()`		Constructor
`Constructor(String)`		Constructor
`Constructor(String, SqlConnection)`		Constructor
`Constructor(String, SqlConnection, SqlTransaction)`		Constructor
`Cancel()`	`void`	Method
`CreateParameter()`	`SqlParameter`	Method
`Dispose()`	`void`	Method
`ExecuteNonQuery()`	`int`	Method
`ExecuteReader()`	`SqlDataReader`	Method
`ExecuteReader (CommandBehavior)`	`SqlDataReader`	Method
`ExecuteScalar()`	`Object`	Method
`ExecuteXmlReader()`	`XmlReader`	Method
`Prepare()`	`void`	Method
`ResetCommandTimeout`	`void`	Method
`CommandText`	`String`	Property
`CommandTimeout`	`int`	Property
`CommandType`	`CommandType`	Property
`Connection`	`SqlConnection`	Property
`Notification`	`SqlNotificationRequest`	Property
`NotificationAutoEnlist`	`Boolean`	Property
`Parameters`	`SqlParameterCollection`	Property

(*Continued*)

TABLE 4-3: Public Members of SqlCommand (*Continued*)

Name	Return Value/Type	Member Type
`Transaction`	`SqlTransaction`	Property
`UpdatedRowSource`	`UpdateRowSource`	Property
`StatementCompleted`	`StatementCompleted EventHandler`	Event

the same time on this connection. This functionality, known as *multiple active resultsets* (MARS), is available only when using the data provider from a client.

- You cannot cancel a `SqlCommand` inside a stored procedure using the `SqlCommand`'s `Cancel` method.
- `SqlNotificationRequest` and `SqlDependency` do not work with commands issued inside SQL Server.

When you execute parameterized queries or stored procedures, you specify the parameter values through the `Parameters` property of the `SqlCommand` class. This property can contain a `SqlParameterCollection` that is a collection of `SqlParameter` instances. The `SqlParameter` instance contains a description of the parameter and also the parameter value. Properties of the `SqlParameter` class include parameter name, data type (including precision and scale for decimal parameters), parameter length, and parameter direction. The `SqlClient` provider uses named parameters rather than positional parameters. Use of named parameters means the following:

- The parameter name is significant; the correct name must be specified.
- The parameter name is used as a parameter marker in parameterized `SELECT` statements, rather than the ODBC/OLE DB question-mark parameter marker.
- The order of the parameters in the collection is not significant.
- Stored procedure parameters with default values may be omitted from the collection; if they are omitted, the default value will be used.
- Parameter direction must be specified as a value of the `ParameterDirection` enumeration.

This enumeration contains the values `Input`, `Output`, `InputOutput`, and `ReturnCode`. Although Chapter 3 mentioned that in T-SQL, all parameters defined as `OUTPUT` can also be used for input, the `SqlClient` provider (and ADO.NET is general) is more precise. Attempting to use the wrong parameter direction will cause an error, and if you specify `ParameterDirection.Output`, input values will be ignored. If you need to pass in a value to a T-SQL procedure that declares it as `OUTPUT`, you must use `ParameterDirection.InputOutput`. Listing 4-7 shows an example of executing a parameterized T-SQL statement.

LISTING 4-7: Using a parameterized SQL statement

```
SqlConnection conn = new SqlConnection("context connection=true");
conn.Open();
SqlCommand cmd = conn.CreateCommand();

// set the command text
// use names as parameter markers
cmd.CommandText =
 "insert into jobs values(@job_desc, @min_lvl, @max_lvl)";

// names must agree with markers
// length of the VarChar parameter is deduced from the input value
cmd.Parameters.Add("@job_desc", SqlDbType.VarChar);
cmd.Parameters.Add("@min_lvl", SqlDbType.TinyInt);
cmd.Parameters.Add("@max_lvl", SqlDbType.TinyInt);

// set values
cmd.Parameters[0].Value = "A new job description";
cmd.Parameters[1].Value = 10;
cmd.Parameters[2].Value = 20;

// execute the command
// should return 1 row affected
int rows_affected = cmd.ExecuteNonQuery();
```

Obtaining Results

Execution of SQL commands can return the following:

- A numeric return code
- A count of rows affected by the command
- A single scalar value

- One or more multirow results using SQL Server's default (cursorless) behavior
- A stream of XML

Some commands, such as a command that executes a stored procedure, can return more than one of these items—for example, a return code, a count of rows affected, and many multirow results. You tell the provider which of these output items you want by using the appropriate method of `SqlCommand`, as shown in Table 4-4.

When you return data from a `SELECT` statement, it is a good idea to use the lowest-overhead choice. Because of the amount of internal processing and the number of object allocations needed, `ExecuteScalar` may be faster than `ExecuteReader`. You need to consider the shape of the data that is returned, of course. Using `ExecuteReader` to return a forward-only, read-only cursorless set of results is always preferred over using a server cursor. Listing 4-8 shows an example of when to use each results-returning method.

LISTING 4-8: Returning rows with SqlClient

```
SqlConnection conn = new SqlConnection("context connection=true");
conn.Open();
SqlCommand cmd = conn.CreateCommand();

// 1. this is a user-defined function
// returning a single value (authorname) as VARCHAR
cmd.CommandText = "GetFullAuthorNameById";
// required from procedure or UDF
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue("@id", "172-32-1176");

String fullname = (String)cmd.ExecuteScalar();
// use fullname
cmd.Parameters.Clear();

// 2. returns one row
cmd.CommandText = "GetAuthorInfoById";
// required from procedure or UDF
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue("@id", "172-32-1176");

SqlDataReader rdr1 = cmd.ExecuteReader();
// use fields in SqlDataReader
```

```
rdr1.Close();
cmd.Parameters.Clear();

// 3. returns multiple rows
cmd.CommandText = "select * from authors";
cmd.CommandType = CommandType.Text;

SqlDataReader rdr2 = cmd.ExecuteReader();
while (rdr2.Read())
  // process rows in SqlDataReader
  { }
rdr2.Close();
```

`SqlDataReader` encapsulates multiple rows that can be read in a forward-only manner. You move to the next row in the set by using the `SqlDataReader`'s `Read()` method, as shown in Listing 4-8. After you call `ExecuteReader`, the resultant `SqlDataReader` is positioned before the first row in the set, and an initial `Read` positions it at the first row. The `Read` method returns false when there are no more rows in the set. If more than one rowset is available, you move to the next rowset by calling `SqlDataReader`'s `NextResult` method. While you are positioned on a row, the `IDataRecord` interface can be used to read data. You can use loosely typed ordinals or names to read the data in single columns. Using ordinals or names is a syntactic shortcut to using `IDataRecord.GetValue(n)`. This returns the value as a .NET Framework `System.Object`, which must be cast to the correct type.

TABLE 4-4: How to Obtain Different Result Types

Result Desired	Mechanism to Obtain It
Return code	Parameter with `ParameterDirection` of `ReturnCode`
Count of rows affected	Returned value from `SqlCommand.ExecuteNonQuery` or Use `SqlCommand.ExecuteReader` and `SqlDataReader.RecordsAffected`
Scalar value	Use `SqlCommand.ExecuteScalar`
Cursorless mode results	Use `SqlCommand.ExecuteReader`
XML stream	Use `SqlCommand.ExecuteXmlReader`

If you know the data type of the value, you can use more strongly typed column accessors. Both SQL Server providers have two kinds of strongly typed accessors. `IDataReader.GetDecimal(n)` is an example; this returns the value of the first column of the current row as a .NET Framework `System.Decimal` data type. If you want full SQL Server type fidelity, it is better to use `SqlDataReader`'s SQL Server–specific accessors, such as `IDataReader.GetSqlDecimal(n)`; these return instances of structures from the `System.Data.SqlTypes` namespace. These types are isomorphic with SQL Server data types; examples of their use and reasons why they are preferable to the .NET Framework base data types when used inside the server are covered in Chapter 3. Listing 4-9 shows an example of using each type.

LISTING 4-9: Getting column values from a SqlDataReader

```
SqlConnection conn = new SqlConnection("context connection=true");
conn.Open();
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "select * from authors";
cmd.CommandType = CommandType.Text;

SqlDataReader rdr = cmd.ExecuteReader();
while (rdr.Read() == true)
{
  string s;
  // 1. Use ordinals or names
  //    explicit casting, if you know the right type
  s = (string)rdr[0];
  s = (string)rdr["au_id"];

  // 2. Use GetValue (must cast)
  s = (string)rdr.GetValue(0);

  // 3. Strong typed accessors
  s = rdr.GetString(0);

  // 4. Accessors for SqlTypes
  SqlString s2 = rdr.GetSqlString(0);
}
```

Although you can process results obtained inside .NET Framework procedural code, you can also pass these items back to the client. This is accomplished through the `SqlPipe` class, which is described later in the chapter. Note that each of the classes returns rows, which must be processed sequentially; these results cannot be updated in place.

Transactions

Multiple SQL operations within a stored procedure or user-defined function can be executed individually or composed within a single transaction. Composing multistatement procedural code inside a transaction ensures that a set of operations has ACID properties. *ACID* is an acronym for the following:

- *Atomicity*—All the operations in a transaction will succeed, or none of them will.
- *Consistency*—The transaction transforms the database from one consistent state to another.
- *Isolation*—Each transaction has its own view of the database state.
- *Durability*—These behaviors are guaranteed even if the database or host operating system fails—for example, because of a power failure.

You can use transactions in two general ways within the `SqlClient` managed provider: by starting a transaction by using the `SqlConnection`'s `BeginTransaction` method or by using declarative transactions using `System.Transaction.TransactionScope`. The `TransactionScope` is part of a new library in .NET Framework 2.0: the `System.Transactions` library. Listing 4-10 shows a simple example of each method.

LISTING 4-10: SqlClient can use two different coding styles for transactions

```
// Example 1: start transaction using the API
SqlConnection conn = new SqlConnection("context connection=true");
conn.Open();
SqlTransaction tx = conn.BeginTransaction();
// do some work
tx.Commit();
conn.Dispose();
```

```
// Example 2: start transaction using Transaction Scope
using System.Data.SqlClient;
using System.Transactions;

using (TransactionScope ts = new TransactionScope())
{
  SqlConnection conn = new SqlConnection("context connection=true");
  // connection auto-enlisted in transaction on Open()
  conn.Open();
  // transactional commands here
  conn.Close();
  ts.Complete();
} // transaction commits when TransactionScope.Dispose called implicitly
```

If you've done any ADO.NET coding before, you've probably run into the `BeginTransaction` method. This method encapsulates issuing a `BEGIN TRANSACTION` statement in T-SQL. The `TransactionScope` requires a bit more explanation.

The `System.Transactions` library is meant to provide a representation of the concept of a transaction in the managed world. It is also a lightweight way to access MSDTC, the distributed transaction coordinator. It can be used as a replacement for the automatic transaction functionality in COM+ exposed by the `System.EnterpriseServices` library, but it does not require the components that use it to be registered in the COM+ catalog. `System.EnterpriseServices` cannot be used in .NET Framework procedural code that runs in SQL Server. To use automatic transactions with `System.Transactions`, simply instantiate a `TransactionScope` object with a `using` statement, and any connections that are opened inside the `using` block will be enlisted in the transaction automatically. The transaction will be committed or rolled back when you exit the `using` block and the `TransactionScope`'s `Dispose` method is called. Notice that the default behavior when `Dispose` is called is to roll back the transaction. To commit the transaction, you need to call the `TransactionScope`'s `Complete` method.

In SQL Server 2005, using the `TransactionScope` starts a local, not a distributed, transaction. This is the behavior whether `TransactionScope` is used with client-side code or SQLCLR procedures unless there is already a transaction started when the SQLCLR procedure is invoked. This phenomenon is illustrated below:

```
-- Calling a SQLCLR procedure that uses TransactionScope

EXECUTE MySQLCLRProcThatUsesTransactionScope -- local transaction
GO

BEGIN TRANSACTION
-- other T-SQL statements
EXECUTE MySQLCLRProcThatUsesTransactionScope -- distributed transaction
COMMIT
GO
```

The transaction actually begins when `Open` is called on the `SqlConnection`, not when the `TransactionScope` instance is created. If more than one `SqlConnection` is opened inside a `TransactionScope`, both connections are enlisted in a distributed transaction when the second connection is opened. The transaction on the first connection actually changes from a local transaction to a distributed transaction. Recall that you can have only a single instance of the context connection, so opening a second connection really means opening a connection using `SqlClient` and a network library. Most often, you'll be doing this specifically to start a distributed transaction with another database. Because of the network traffic involved and the nature of the two-phase commit protocol used by distributed transactions, a distributed transaction will be much higher overhead than a local transaction.

`BeginTransaction` and `TransactionScope` work identically in the simple case. But some database programmers like to make each procedure usable and transactional when used stand-alone or when called when a transaction already exists. To accomplish this, you would put transaction code in each procedure. When one procedure with transaction code calls another procedure with transaction code, this is called *composing transactions.* SQL Server supports nesting of transactions and named savepoints, but not autonomous (true nested) transactions. So using a T-SQL procedure `X` as an example,

```
CREATE PROCEDURE X
AS
BEGIN TRAN
-- work here
COMMIT
```

calling it stand-alone (`EXECUTE X`) means that the work is in a transaction. Calling it from procedure `Y`

```
CREATE PROCEDURE Y
AS
BEGIN TRANSACTION
-- other work here
EXECUTE X
COMMIT
```

doesn't start an autonomous transaction (a second transaction with a different scope); the `BEGIN TRANSACTION` in `X` merely increases a T-SQL variable

`@@TRANCOUNT` by one. Two error messages are produced when you roll back in procedure `X` while it's being called by procedure `Y`:

```
Msg 266, Level 16, State 2, Procedure Y, Line 0
Transaction count after EXECUTE indicates that a COMMIT or ROLLBACK
TRANSACTION statement is missing. Previous count = 1, current count = 0.
Msg 3902, Level 16, State 1, Procedure X, Line 5
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

I'd like to emulate this behavior in SQLCLR—that is, have a procedure that acts like `X` and that can be used stand-alone or composed. I can do something akin to T-SQL (and get the interesting rollback behavior with a slightly different error number) using the `BeginTransaction` method on the context `SqlConnection`. Using a `TransactionScope` has a different behavior, however. If I have a SQLCLR proc that looks like this (condensed version),

```
public static void X {
  using (TransactionScope ts = new TransactionScope())
  using (
    SqlConnection conn = new SqlConnection("Context connection=true"))
  {
  conn.Open();
  ts.Complete();
  }
}
```

and if SQLCLR `X` is used stand-alone, all is well, and the `TransactionScope` code gets a local transaction. If SQLCLR `X` is called from procedure `Y` (above), `SqlConnection`'s `Open` starts a distributed transaction. Apparently, it *has* to be this way, at least for now, because of how `TransactionScope` works. Local transactions don't expose the events that `TransactionScope` needs to compose transactions.

If you *want* a distributed transaction composed with your outer transaction (your `SqlConnection` is calling to another instance, for example), use `TransactionScope`; if you *don't* want one, use `SqlConnection`'s `BeginTransaction`. It won't act any different from T-SQL (except you do get a different error number) if you roll back inside an "inner" transaction. But you get a nesting local transaction with `BeginTransaction`.

Listing 4-11 shows an example of using a distributed transaction with `TransactionScope`.

LISTING 4-11: A distributed transaction using TransactionScope

```
public static void DoDistributed() {
  string ConnStr =
    "server=server2;integrated security=sspi;database=pubs";
  using (TransactionScope ts = new TransactionScope())
  using (SqlConnection conn1 =
         new SqlConnection("Context connection=true"))
  using (SqlConnection conn2 =
         new SqlConnection(ConnStr))
  {
  conn1.Open();
  conn2.Open();
  // do work on connection 1
  // do work on connection 2
  // ask to commit the distributed transaction
  ts.Complete();
  }
}
```

TransactionScope Exotica

You can use options of the `TransactionScope` class to compose multiple transactions in interesting ways. You can start multiple transactions, for example (but not on the context connection), by using a different `TransactionScopeOption`. Listing 4-12 will begin a local transaction using the context connection and then begin an autonomous transaction using a connection to the same server.

LISTING 4-12: Producing the equivalent of an autonomous transaction

```
public static void DoPseudoAutonomous() {
  string ConnStr =
    "server=sameserver;integrated security=sspi;database=samedb";
  using (TransactionScope ts1 = new TransactionScope())
  using (SqlConnection conn1 =
         new SqlConnection("context connection=true"))
  {
  conn1.Open();
  // do work on connection 1, then
  {
      using (TransactionScope ts2 =
             new TransactionScope(TransactionScopeOption.RequiresNew))
      using (SqlConnection conn2 = new SqlConnection(ConnStr))
      {
         conn2.Open();
         // do work on connection 2
         ts2.Complete();
      }
  // ask to commit transaction1
  ts1.Complete();
  }
}
```

This code works because it uses a second connection to the same server to start a second transaction. This second connection is separate from the first one, not an autonomous transaction on the same connection. The result is the same as you would get from an autonomous transaction; you just need two connections (the context connection and a second connection) to accomplish it.

Attempting to use any `TransactionScopeOption` other than the default `TransactionRequired` fails if there already is an existing transaction (as we saw before, when `BEGIN TRANSACTION` was called in T-SQL before `EXECUTE` on the SQLCLR procedure) and you attempt to use context connection, as shown in Listing 4-13. You'll get a message saying `"no autonomous transaction"`.

LISTING 4-13: Attempting to use autonomous transactions on a single connection fails

```
-- Calling a SQLCLR procedure that uses TransactionScope
-- with an option other than TransactionRequired

EXECUTE DoPseudoAutonomous -- works
GO

BEGIN TRANSACTION
-- other T-SQL statements
EXECUTE DoPseudoAutonomous -- fails, "no autonomous transaction"
COMMIT
GO
```

This is because SQL Server doesn't support autonomous transactions on a single connection.

Best Practices

With all these options and different behaviors, what's the best and easier thing to do to ensure that your local transactions always work correctly in SQLCLR procedures? At this point, because SQL Server 2005 doesn't support autonomous transactions on the same connection, SqlConnection's BeginTransaction method is the best choice for local transactions. In addition, you need to use the Transaction.Current static properties in System.Transactions.dll to determine whether a transaction already exists—that is, whether the caller has already started a transaction. Listing 4-14 shows a strategy that works well whether or not you compose transactions.

LISTING 4-14: A generalized strategy for nesting transactions

```
// Works whether caller has transaction or not
public static int ComposeTx()
{
  int returnCode = 0;
  // determine if we have transaction
  bool noCallerTx = (Transaction.Current == null);
  SqlTransaction tx = null;

  SqlConnection conn = new SqlConnection("context connection=true");
  conn.Open();

  if (noCallerTx)
    tx = conn.BeginTransaction();

  try {
    // do the procedure's work here
    SqlCommand workcmd = new SqlCommand(
      "INSERT jobs VALUES('New job', 10, 10)", conn);
    if (tx != null)
      workcmd.Transaction = tx;
    int rowsAffected = workcmd.ExecuteNonQuery();

    if (noCallerTx)
      tx.Commit();
  }

  catch (Exception ex) {
    if (noCallerTx) {
      tx.Rollback();
      // raise error - covered later in chapter
    }
    else {
      // signal an error to the caller with return code
      returnCode = 50010;
    }
  }
  conn.Dispose();
  return returnCode;
}
```

For distributed transactions as well as the pseudoautonomous transactions described earlier, you must use `TransactionScope` or a separate second connection back to the server using `SqlClient`. If you don't mind the behavior that nesting transactions with `TransactionScope` force a distributed transaction and the extra overhead caused by MSDTC, you can use `TransactionScope` all the time. Finally, if you know your procedure won't be called with an existing transaction, you can use either `BeginTransaction` or `TransactionScope`. Refraining from nesting transactions inside nested procedures may be a good strategy until this gets sorted out.

Pipe

In the section on results earlier in this chapter, we mentioned that you have a choice of processing results in your procedural code as part of its logic or returning the results to the caller. Consuming `SqlDataReader`s or the stream of XML in procedural code makes them unavailable to the caller; you cannot process a cursorless mode result more than once. The code for in-process consumption of a `SqlDataReader` is identical to `SqlClient`; you call `Read()` until no more rows remain. To pass a resultset back to the client, you need to use a special class, `SqlPipe`.

The `SqlPipe` class represents a channel back to the client; this is a TDS (Tabular Data Stream) output stream if the TDS protocol is used for client communication. You obtain a `SqlPipe` by using the static `SqlContext.Pipe property`. Rowsets, single rows, and messages can be written to the pipe. Although you can get a `SqlDataReader` and return it to the client through the `SqlPipe`, this is less efficient than just using a new special method for the `SqlPipe` class: `ExecuteAndSend`. This method executes a `SqlCommand` and points it directly to the `SqlPipe`. Listing 4-15 shows an example.

LISTING 4-15: Using SqlPipe to return rows to the client

```
public static void getAuthorsByState(SqlString state)
{
  SqlConnection conn = new SqlConnection("context connection=true");
  conn.Open();
  SqlCommand cmd = conn.CreateCommand();
  cmd.CommandText = "select * from authors where state = @state";
  cmd.Parameters.Add("@state", SqlDbType.VarChar);
  cmd.Parameters[0].Value = state;
  SqlPipe pipe = SqlContext.Pipe;
  pipe.ExecuteAndSend(cmd);
}
```

In addition to returning an entire set of results through the pipe, `SqlPipe`'s `Send` method lets you send an instance of the `SqlDataRecord` class. You can also batch the send operations however you like. An interesting feature of using `SqlPipe` is that the result is streamed to the caller immediately, as fast as you are able to send it, taking into consideration that the client stack may do row buffering. This may improve performance at the client because you can process rows as fast as they are sent out the pipe. Note that you can combine executing a command and sending the results back through `SqlPipe` in a single operation with the `ExecuteAndSend` convenience method, using a `SqlCommand` as a method input parameter.

`SqlPipe` also contains methods for sending scalar values as messages and affects how errors are exposed. We'll talk about error handling practices next. The entire set of methods exposed by `SqlPipe` is shown in Table 4-5.

There is also a boolean property on the `SqlPipe` class, `IsSendingResults`, that enables you to find out whether the `SqlPipe` is busy. Because multiple active resultsets are not supported when you're inside SQL Server,

TABLE 4-5: Methods of the SqlPipe Class

Method	What It Does
`ExecuteAndSend(SqlCommand)`	Executes command, returns results through `SqlPipe`
`Send(String)`	Sends a message as a string
`Send(SqlDataReader)`	Sends results through `SqlDataReader`
`Send(SqlDataRecord)`	Sends results through `SqlDataRecord`
`SendResultsStart (SqlDataRecord)`	Starts sending results
`SendResultsRow (SqlDataRecord)`	Sends a single row after calling `SendResultsStart`
`SendResultsEnd()`	Indicates finished sending rows

attempting to execute another method that uses the pipe while it's busy will procedure an error. The only exception to this rule is that `SendResultsStart`, `SendResultsRow`, and `SendResultsEnd` are used together to send results one row at a time.

`SqlPipe` is available for use only inside a SQLCLR stored procedure. Attempting to get the `SqlContext.Pipe` value inside a user-defined function returns a null instance. This is because sending rowsets is not permitted in a user-defined function. Within a stored procedure, however, you can not only send rowsets through the `SqlPipe` by executing a command that returns a rowset, but also synthesize your own. Synthesizing rowsets involves the use of two server-specific classes we haven't seen before: `SqlDataRecord` and `SqlMetaData`.

(chapter continues...)