Chapter 3

An Overview of SQL CLR

—Andrew Brust

In this chapter:

Getting Started: Enabling CLR Integration ........................................... 46
Visual Studio/SQL Server Integration .................................................. 48
Your First SQL CLR Stored Procedure .............................................. 53
CLR Stored Procedures and Server-Side Data Access ......................... 55
Deployment ....................................................................................... 59
CLR Functions .................................................................................. 65
CLR Triggers ...................................................................................... 69
CLR Aggregates ............................................................................... 74
CLR Types ......................................................................................... 78
Security ............................................................................................ 84
Examining and Managing CLR Types in a Database ......................... 85
Best Practices for SQL CLR Usage .................................................... 91
Summary .......................................................................................... 92

The banner headline for Microsoft SQL Server 2005 is its integration of the Microsoft .NET common language runtime (CLR). This architectural enhancement means that SQL Server can use certain .NET classes as basic data types and can accommodate the use of .NET languages for the creation of stored procedures, triggers, and functions, and even user-defined aggregates.

Note Throughout this chapter, we will refer to the CLR integration in SQL Server as SQL CLR features, functionality, or integration, and we will refer to stored procedures, triggers, functions, aggregates, and user-defined types as the five basic SQL CLR entities.

Let’s face facts: Transact SQL (T-SQL) is essentially a hack. Back when SQL Server was first developed, Microsoft and Sybase took SQL—a declarative, set-based language—and added variable declaration, conditional branching, looping, and parameterized subroutine logic to make it into a quasi-procedural language. Although extremely clever and useful, T-SQL lacked, and still lacks, many of the niceties of a full-fledged procedural programming language.
Part I: Design Fundamentals and Core Technologies

This shortcoming has forced some people to write T-SQL stored procedures that are overly complex and difficult to read. It has forced others to put logic in their middle-tier code that they would prefer to implement on the database. And it’s even forced some people to abandon stored procedures altogether and use dynamic SQL in their applications, a practice that we do not endorse. Because of these workarounds to address T-SQL’s procedural limitations, CLR integration is a welcome new feature in SQL Server, and it has caught the market’s attention.

Meanwhile, T-SQL—and, one might argue, SQL itself—is vastly superior to procedural languages for querying and manipulating data. Its set-based syntax and implementation simply transcend the approach of procedurally iterating through rows of data. This is no less true in SQL Server 2005 than in previous versions of the product, and T-SQL has been greatly enhanced in this release (as detailed in Chapter 2), making it more valuable still.

So this places database application developers at a crossroads. We must simultaneously learn how the SQL CLR features work and develop a sophisticated, judicious sense of when to use T-SQL instead of the SQL CLR feature set. We must resist the temptation to completely “.NET-ify” our databases but learn to take advantage of the SQL CLR feature set where and when prudent. This chapter aims to help you learn to use SQL CLR features and to develop an instinct for their appropriate application.

In this chapter, you will learn:

■ How to enable (or disable) SQL CLR integration on your SQL Server
■ How SQL Server accommodates CLR code, through the loading of .NET assemblies
■ How to use SQL Server 2005 and Visual Studio 2005 together to write SQL CLR code and deploy it, simply and quickly
■ How to deploy SQL CLR code independently of Visual Studio, using T-SQL commands, with or without the help of SQL Server Management Studio
■ How to create simple CLR stored procedures, triggers, functions, aggregates, and user-defined types, use them in your databases, and utilize them from T-SQL
■ How both the standard SQL Server client provider and the new server-side library can be combined to implement SQL CLR functionality
■ How SQL CLR security works and how to configure security permissions for your assemblies
■ When to use SQL CLR functionality, and when to opt to use T-SQL instead

Getting Started: Enabling CLR Integration

Before you can learn how to use SQL CLR features, you need to know how to enable them. As with many new products in the Microsoft Windows Server System family, many advanced features of SQL Server 2005 are disabled by default. The reasoning behind this is sound: Each
additional feature that is enabled provides extra “surface area” for attacks on security or integrity of the product, and the added exposure is inexcusable if the feature goes unused.

The SQL CLR features of SQL Server 2005 are sophisticated and can be very useful, but they are also, technically, nonessential. It is possible to build high-performance databases and server-side programming logic without SQL CLR integration, so it is turned off by default.

Don’t be discouraged, though: Turning on the feature is easy. Microsoft provides both a user-friendly GUI tool (aptly named the SQL Server Surface Area Configuration tool) and a system stored procedure for enabling or disabling SQL CLR integration. We’ll cover both approaches.

To use the Surface Area Configuration tool, simply start it from the Configuration Tools subgroup in the Microsoft SQL Server 2005 programs group on the Windows start menu. Figure 3-1 shows the tool as it appears upon startup.

![Surface Area Configuration Tool](image)

**Figure 3-1** The SQL Server 2005 Surface Area Configuration tool

To configure CLR integration, click the Surface Area Configuration For Features link at the bottom of the form. After a short pause, the Surface Area Configuration For Features dialog box appears; a tree view-style list of features appears on the left, and the Ad Hoc Remote Queries feature is preselected. Click the CLR Integration node immediately below it, and you will see an Enable CLR Integration check box on the right of the form. (This is shown in Figure 3-2.) To enable SQL CLR features, make sure that the check box is checked, and click OK to close the Surface Area Configuration For Features window. (You can also clear the check box to disable SQL CLR integration.) Close the Surface Area Configuration tool by clicking its close box in the upper-right corner of the window.
Part I: Design Fundamentals and Core Technologies

Figure 3-2 The Surface Area Configuration For Features window

If you’d prefer a command-line method for enabling or disabling SQL CLR functionality, open up SQL Server Management Studio and connect to the server you’d like to configure. Then, from a query window, type the following commands, and click the Execute button on the Management Studio SQL Editor toolbar.

```sql
sp_configure 'clr enabled', 1
GO
RECONFIGURE
GO
```

That’s all there is to it! To disable SQL CLR integration, just use a value of 0, instead of 1, as the second parameter value in the `sp_configure` call.

Tip  Don’t forget that this will work from any tool that can connect to SQL Server, not just Management Studio. In fact, you could issue the previous command text from your own code using the ADO.NET `SqlCommand` object’s `ExecuteNonQuery` method as long as your code can connect to your server and your server can authenticate as a user in the sysadmin server role.

With SQL CLR integration enabled, you’re ready to get started writing SQL CLR code. Before we dive in, we need to discuss Visual Studio/SQL Server integration and when to use it.

**Visual Studio/SQL Server Integration**

Visual Studio 2005 and SQL Server 2005 integrate tightly in a number of ways. It’s important to realize, however, that the use of Visual Studio integration is completely optional and the use of T-SQL is a sufficient substitute. T-SQL has been enhanced with new DDL commands for
maintaining CLR assemblies, types, and aggregates, and its existing commands for stored procedures, triggers, and functions have been enhanced to recognize code within deployed assemblies. Visual Studio can execute those commands on your behalf. It can also make writing individual SQL CLR classes and functions easier.

Ultimately, we think all developers should be aware of both Visual Studio–assisted and more manual coding and deployment methods. You might decide to use one method most of the time, but in some situations you’ll probably need the other, so we want to prepare you. As we cover each major area of SQL CLR programming, we will discuss deployment from both points of view. We’ll cover some general points about Visual Studio integration now, and then we’ll move on to cover SQL CLR development.

**SQL Server Projects in Visual Studio**

The combination of Visual Studio 2005 and SQL Server 2005 on the same development machine provides a special SQL Server Project type in Visual Studio and, within projects of that type, defined templates for the five basic SQL CLR entities. These templates inject specific code attributes and function stubs that allow you to create SQL CLR code easily. The attributes are used by Visual Studio to deploy your assembly and its stored procedures, triggers, and so on to your database. Some of them are also used by SQL Server to acknowledge and properly use your functions, user-defined types (UDTs), and aggregates.

To test out the new project type and templates, start Visual Studio 2005 and create a new project by using the File/New/Project... main menu option, the New Project toolbar button, the Ctrl+Shift+N keyboard accelerator, or the Create Project... hyperlink on the Visual Studio Start Page. In the New Project dialog box (Figure 3-3), select Database from the Project types tree view on the left (the Database node appears under the parent node for your programming language of choice; in Figure 3-3, the language is C#), and click the SQL Server Project icon in the Templates list on the right. Enter your own project name if you’d like, and click OK.

![Figure 3-3](image) The Visual Studio 2005 New Project dialog box with the SQL Server project type selected
Next, the Add Database Reference dialog box appears (Figure 3-4).

![Add Database Reference dialog box](image)

**Figure 3-4** The Add Database Reference dialog box

Because Visual Studio provides automated deployment of your SQL CLR code, it must associate your project with a specific server and database via a database reference (connection). Any database connections that have already been defined in the Server Explorer window appear in this window, as does an Add New Reference button that allows you to define a new connection, if necessary. Pick an existing connection or define a new one, and then click OK. The project opens.

**Note** If no data connections have already been defined in the Server Explorer window, the New Database Reference dialog box will appear in place of the Add Database Reference dialog box. In the New Database Reference dialog box, you may specify server, login, and database details for a new database connection that will be used by your project as its database reference and added to the Server Explorer as a new data connection.

You can easily add preconfigured classes for the five basic SQL CLR entities to your project. You can do this in a number of ways: directly from the Project menu or from the Add submenu on the Server Explorer's project node shortcut menu (Figure 3-5).

You can also add the preconfigured classes from the Add New Item dialog box (Figure 3-6), which is available from the Project/Add New Item... option on the main menu, or the Add/New Item... option on the Solution Explorer project node's shortcut menu.
Automated Deployment

Once opened, SQL Server projects add a Deploy option to the Visual Studio Build menu. In addition, the Play (Start Debugging) button and the Start Debugging, Start Without Debugging, and Step Over options on the Debug menu (and their keyboard shortcuts F5, Ctrl+F5, and F10, respectively) all deploy the project assembly in addition to performing their listed function.
Visual Studio can do a lot of deployment work for you. But as you’ll learn, you can do so on your own and, in certain circumstances, have more precise control over the deployment process when you do so.

**SQL CLR Code Attributes**

A number of .NET code attributes are provided for SQL CLR developers; these are contained in the `Microsoft.SqlServer.Server` namespace. Many of them are inserted in your code when you use the various templates in the SQL Server project type, as is a `using` statement for the `Microsoft.SqlServer.Server` namespace itself. If you choose to develop code without these templates, you must add the appropriate attributes, and optionally the `using` statement, yourself. Although all these attributes are provided in the same namespace, some are used exclusively by Visual Studio and others are used by both Visual Studio and SQL Server.

Covering all SQL CLR attributes and their parameters would itself require an entire chapter, so our coverage will be intentionally selective. Specifically, we will provide coverage of the `SqlProcedure`, `SqlFunction`, `SqlTrigger`, `SqlUserDefinedAggregate`, and `SqlUserDefinedType` attributes. We will not cover the `SqlFacet` and `SqlMethod` attributes.

Just as certain attributes are not covered here, we cover only some of the parameters accepted by the attributes that we do cover. And in some cases, we cover only certain of the possible values that can be passed to these attributes. For example, `SqlFunction` accepts several parameters but the only ones we will cover are `Name`, `FillRowMethodName`, and `TableDefinition`. For `SqlUserDefinedAggregate` and `SqlUserDefinedType`, we will cover only a single value setting for the `Format` parameter, and will not cover the several other parameters those two attributes accept.

The coverage we provide will be more than sufficient for you to implement basic, intermediate, and certain advanced functionality with all the basic five SQL CLR entities. The attributes and parameters that we won’t cover are useful mostly for optimizing your SQL CLR code, and they are well documented in SQL Server Books Online and articles on MSDN.

**About the Sample Code**

The sample .NET code for this chapter is provided in two versions. The primary material is supplied as a Visual Studio SQL Server project, accessible by opening the solution file Chapter03.sln in the Chapter03 subfolder of this chapter’s VS sample code folder. We also supply the code as a standard Class Library project, accessible by opening the solution file Chapter03Manual.sln in the Chapter03Manual subfolder. The code in each project is virtually identical, although the Class Library project does not autodeploy when the various Build and Debug options are invoked in Visual Studio 2005. As we cover each SQL CLR feature, we’ll discuss how automated deployment takes place from the SQL Server project and how command-driven deployment should be performed for the Class Library project.
We’ll also discuss executing test scripts from within Visual Studio for the SQL Server project and from SQL Server Management Studio for the Class Library project. As a companion to those discussions, we also provide a Management Studio project, accessible by opening Chapter03.ssmsln in this chapter’s SSMS folder. This project consists of a number of SQL scripts used for testing the sample SQL CLR code and a script for cleaning up everything in the database created by the sample code and tests. The project also contains a script file called CreateObjects.sql, which deploys the Class Library assembly and the SQL CLR entities within it.

Your First SQL CLR Stored Procedure

Although SQL CLR programming can get quite complex and involved, it offers in reality a simple model that any .NET developer can use with high productivity in relatively short order. That’s because the crux of SQL CLR functionality is nothing more than the ability of SQL Server 2005 to load .NET assemblies into your database and then to allow you to use the procedures, functions, and types within the assembly as you define your columns, views, stored procedures, triggers, and functions.

To give you a good understanding of SQL CLR integration, we must go through its features and techniques carefully. Before doing so, however, let’s quickly go through an end-to-end scenario for creating and executing a SQL CLR stored procedure. This will make it easier for you to understand the individual features as we describe them.

Strictly speaking, any .NET class library assembly (in certain cases using appropriate .NET code attributes in its classes and functions) can be loaded into your database with a simple T-SQL command. To see how easily this works, start up Management Studio and open a query window using a connection to the AdventureWorks sample database. In the sample code folder for this chapter, confirm that the file Chapter03.dll is located in the VS\Chapter03Manual\Chapter03\bin\Debug subfolder. If the parent folder were C:\ProgrammingSQL2005\Chapter03, you would load the assembly into the AdventureWorks database with the following T-SQL command:

```
CREATE ASSEMBLY Chapter03
FROM 'C:\ProgrammingSQL2005\Chapter03\VS\Chapter03Manual\Chapter03\bin\Debug\Chapter03.dll'
```

There are other syntax options for the CREATE ASSEMBLY command, but for now we’ll focus on the previous limited usage.

Functions in an assembly that reside within a class and perform local computational tasks and/or certain types of data access can be easily exposed as SQL Server stored procedures, triggers, or functions. As with conventional stored procedures, triggers, and functions, all it takes is a simple T-SQL CREATE PROCEDURE, CREATE TRIGGER, or CREATE FUNCTION
Part I: Design Fundamentals and Core Technologies

command to make this happen. We’ll go through each of these options in this chapter, but let’s cut to the chase and create a simple CLR stored procedure right now.

You can view the source code for the Chapter03 assembly by opening the solution file VS\Chapter03Manual\Chapter03Manual.sln in this chapter’s sample code folder. Within the project, the file Sprocs.cs contains the following code:

```csharp
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public partial class Sprocs
{
    public static void spContactsQuick()
    {
        SqlContext.Pipe.ExecuteAndSend(new SqlCommand("Select * from Person.Contact");
    }
};
```

The code within the procedure is designed to connect to the database in which its assembly has been loaded (AdventureWorks), perform a `SELECT *` against the Person.Contact table, and use special server-side objects to send the data back to the client application. To make this CLR code available via SQL Server as a stored procedure, also called `spContactsQuick`, you simply execute the following command from the Management Studio query window you opened previously.

```sql
CREATE PROCEDURE spContactsQuick
AS EXTERNAL NAME
Chapter03.Sprocs.spContactsQuick
```

**Important** Be sure to enter the `Sprocs.spContactsQuick` portion of the command verbatim. This phrase is case-sensitive.

To test the SQL CLR stored procedure, run it from a Management Studio query window as you would any conventional stored procedure:

```sql
EXEC spContactsQuick
```

Or simply:

```sql
spContactsQuick
```

Management Studio should respond by displaying the contents of the Person.Contact table in the Results tab of the query window.

As you can see from this rather trivial example, writing a CLR stored procedure can be very easy and is a lot like writing client-side or middle-tier code that performs data access using
ADO.NET. The biggest differences involve the provision of a database connection and the fact that the data must be “piped” back to the client rather than loaded into a SqlDataReader and returned, manipulated, or displayed through a UI. In addition, the presence of the SqlContext object differentiates SQL CLR code from conventional .NET data access code. We’ll cover the use of the SqlContext object and its Pipe property in the next section.

The bits of T-SQL and C# code just shown certainly don’t tell the whole SQL CLR story. The use of the ExecuteAndSend method allowed us to skip over a number of otherwise important concepts. There are three ways to deploy assemblies, and you’ve seen only a simplified version of one of those ways. Security considerations must be taken into account, and we haven’t even begun to look at triggers, functions, aggregates, or UDTs. So although the example showed how easy SQL CLR programming can be, we’ll now take our time and show you the nooks and crannies.

**CLR Stored Procedures and Server-Side Data Access**

Our previous “quick and dirty” sample looked at CLR stored procedure development, but we need to cover that topic more thoroughly now. We’ve already covered the mechanics of writing and deploying a stored procedure, but let’s back up a bit and try and understand how CLR stored procedures work from a conceptual standpoint.

SQL CLR stored procedure code runs in an instance of the .NET CLR that is hosted by SQL Server itself; it is not called as an external process, as COM-based extended stored procedures (XPs) would be. Because SQL CLR code runs in the context of the server, it treats objects in the database as native, local objects, more or less. Likewise, it must treat the client that calls it as remote. This contextual environment is, in effect, the opposite of that under which client and middle-tier ADO.NET code runs. This takes a little getting used to, but once you’ve mastered thinking about things this way, SQL CLR code becomes easy to write and understand.

Meanwhile, as .NET has no intrinsic way of accessing local objects on the server or transmitting data and messages to the client, you must use a special set of classes to perform these tasks. These classes are contained in the Microsoft.SqlServer.Server namespace.

### Note

As an aside, it is interesting and important to note that the Microsoft.SqlServer.Server namespace is actually supplied by the System.Data Framework assembly. This means that you don’t need to worry about adding a reference to your project to use this namespace. The namespace’s location within System.Data also further emphasizes the tight integration between .NET and SQL Server.

If you’d like, you can think of Microsoft.SqlServer.Server as a helper library for System.Data.SqlClient. It supplies the SQL CLR code attributes we already mentioned, a few enumerations, an exception class, an interface, and five classes: SqlContext, SqlPipe, SqlTriggerContext, SqlMetaData, and SqlDataRecord. We’ll cover SqlMetaData and SqlDataRecord at the end of this
section, and we’ll cover `SqlTriggerContext` when we discuss CLR triggers. We’ll cover the `SqlContext` and `SqlPipe` objects right now.

At a high level, the `SqlContext` object, which is static, provides a handle to the server-side context in which your code runs. It also has a channel to the client through which you can return data and text: its `Pipe` property, which in turn provides access to a properly initiated `SqlPipe` object.

A `SqlPipe` object can send data and messages to the calling client through several methods: `Send`, `SendResultsStart`, `SendResultsRow`, `SendResultsEnd`, and `ExecuteAndSend`. In the previous code sample, we used the `SqlPipe` object’s `ExecuteAndSend` method to implicitly open a connection, call `ExecuteReader` on an `SqlCommand` object that uses that connection, and transmit the contents of the resulting `SqlDataReader` back to the client. Although the implicit work done by `ExecuteAndSend` might have been convenient, it’s important to avoid such shortcuts in our detailed discussion on SQL CLR programming.

In general, SQL CLR stored procedure code that queries tables in the database must open a connection to that database, use the `SqlCommand` object’s `ExecuteReader` method to query the data, and then use one or a combination of the `Send` methods to send it back. The `Send` methods do not accept `DataSet` objects; they accept only `SqlDataReader` objects, strings, and/or special `SqlDataRecord` objects. Listing 3-1, which shows the implementation of the function `spContacts` from `spTest.cs` in the sample project, is a representative example of how this is done.

Listing 3-1  `spContacts` from `spTest.cs`

```csharp
[SqlProcedure]
public static void spContacts()
{
    SqlConnection conn = new SqlConnection("context connection=true");
    SqlCommand cm = new SqlCommand("Select * from Person.Contact", conn);
    conn.Open();
    SqlDataReader dr = cm.ExecuteReader();
    SqlContext.Pipe.Send("Starting data dump");
    SqlContext.Pipe.Send(dr);
    SqlContext.Pipe.Send("Data dump complete");
    dr.Close();
    conn.Close();
}
```

For this code to work, we need to use both the `Microsoft.SqlServer.Server` and `System.Data.SqlClient` namespaces (and if you look in the sample project rather than Listing 3-1, you’ll see that we have). This is because any conventional ADO.NET objects we might use, such as `SqlConnection`, `SqlCommand`, and `SqlDataReader`, are supplied to us from `System.Data.SqlClient`, just as they would be in a conventional client application or middle-tier assembly. As already discussed, we need the `Microsoft.SqlServer.Server` namespace in order to use objects such as `SqlContext` and
Chapter 3: An Overview of SQL CLR


Note Readers who worked with early beta versions of SQL Server 2005 might recall a System.Data.SqlServer library, which in effect supplied all conventional and server-side ADO.NET objects necessary to write SQL CLR code. This hybrid library was eliminated and replaced with the dual-library approach later in the beta process.

Although server-side code uses SqlClient objects, it does so in a specialized way. For example, notice that the context connection=true connection string passed to the SqlConnection object’s constructor. This essentially instructs ADO.NET to open a new connection to the database in which the CLR assembly resides. Notice also the second call to the SqlContext.Pipe object’s Send method. Here, the SqlDataReader parameter overload of the SqlPipe object’s Send method is used to push the contents of the SqlDataReader back to the client. You can think of this method as performing a while (dr.Read()) loop through the SqlDataReader and echoing out the values of each column for each iteration of the loop, but instead of having to do that work yourself, the Send method does it for you.

Before and after the SqlDataReader is piped, we use the String parameter overload of the Send method to send status messages to the client. When this stored procedure is run in Management Studio, the piped text appears on the Results tab of the query window when you use the Management Studio Results To Text option and on the Messages tab when you use the Results To Grid option.

The rest of the listing contains typical ADO.NET code, all of it using objects from the SqlClient provider. And that illustrates well the overall theme of SQL CLR programming: Do what you’d normally do from the client or middle tier, and use a few special helper objects to work within the context of SQL Server as you do so.

Piping Data with SqlDataReader and SqlMetaData

We mentioned that the SqlPipe object’s Send method can accept an object of type SqlDataReader, and we mentioned previously that Microsoft.SqlServer.Server provides this object as well as an object called SqlMetaData. You can use these two objects together in a CLR stored procedure to return a result set one row at a time, instead of having to supply the SqlPipe object’s Send method with an SqlDataReader. This allows (but does not require) you to inspect the data before sending it back to the client. Sending SqlDataReader objects prevents inspection of the data within the stored procedure because SqlDataReader objects are forward-only result set structures. Using the ExecuteAndSend method and an SqlCommand object has the same limitation.

The SqlDataReader object permits .NET code to create an individual record/row to be returned to the calling client. Its constructor accepts an array of SqlMetaData objects, which in turn describe the metadata for each field/column in the record/row.
Listing 3-2, which shows the implementation of function spContactCount from spTest.cs in the sample project, illustrates how to use SqlPipe.Send together with SqlDataRecord and SqlMetaData objects to return a single-column, single-row result set from a stored procedure.

### Listing 3-2  *spContactCount* from *spTest.cs*

```csharp
[SqlProcedure()]
public static void spContactCount()
{
    SqlConnection conn = new SqlConnection("context connection=true");
    SqlCommand cm = new SqlCommand("Select Count(*) from Person.Contact", conn);
    SqlDataReader drc = new SqlDataReader(new SqlMetaData("ContactCount", SqlDbType.Int));
    conn.Open();
    drc.SetInt32(0, (Int32)cm.ExecuteScalar());
    SqlContext.Pipe.Send(drc);
    conn.Close();
}
```

The code declares variable *drc* as a SqlDataReader object and passes its constructor a single SqlMetaData object. (Passing a single object rather than an array is permissible if the SqlDataReader object will only have a single field/column.) The SqlMetaData object describes a column called *ContactCount* of type SqlDbType.Int.

The rest of the code is rather straightforward. First, a context connection and command are opened and a SELECT COUNT(*) query is performed against the AdventureWorks Person.Contact table. Because the query returns a single scalar value, it is run using the SqlCommand object’s ExecuteScalar method. Next, the value returned by ExecuteScalar is casted into an integer and that value is loaded into field/column 0 (the only one) of the SqlDataReader object using its SetInt32 method. The SqlDataReader is then piped back to the client using the SqlContext object’s Send method.

The SQL Server Stored Procedure template inserts a using statement for this namespace. If you are creating SQL CLR code without using this template, you should add the using statement yourself.

### Note

The SqlDbType enumeration is contained within the System.Data.SqlTypes namespace. The SQL Server Stored Procedure template inserts a using statement for this namespace. If you are creating SQL CLR code without using this template, you should add the using statement yourself.

If we wanted to send back multiple SqlDataReader objects, we would send the first one using the SqlContext object’s SendResultsStart method and then send all subsequent SqlDataReader objects using the SendResultsRow method. We would call the SendResultEnd method after all SqlDataReader objects had been sent.

Once the stored procedure has been deployed (the techniques for which we will discuss shortly), you can execute it from SQL Server Management Studio as you would any other stored procedure. Although the result is a single value, it is presented as a column and the
column name ContactCount is shown on the Results tab of the query window. Keep in mind that this `COUNT(*)` query result could have been returned without using the `SqlMetaData` and `SqlDataRecord` objects; the sample is provided to demonstrate the use of these objects as an alternative to piping `SqlDataReader` objects and text to the client.

**CLR Stored Procedure Usage Guidelines**

It’s important to understand how to perform data access and retrieval in CLR stored procedures. As a .NET developer, you already know how to do more computational tasks within your code, so our samples illustrate server-side data access more than anything else. As proof-of-concept code, these samples are completely adequate.

Meanwhile, you should avoid writing CLR stored procedures that merely perform simple “CRUD” (Create, Retrieve, Update, and Delete) operations. Such tasks are better left to conventional T-SQL stored procedures, which typically perform these operations more efficiently than ADO.NET can. CLR stored procedures work well when you need to perform computation on your data and you need the expressiveness of a .NET language to do so (where such expressiveness is missing from T-SQL).

For example, implementing a “fuzzy search” using business logic embedded in .NET assemblies to determine which data has an affinity to other data is a good use of SQL CLR stored procedures. Regular-expression-based data validation in an update or insert stored procedure is another good application of SQL CLR integration. As a general rule, straight data access should be left to T-SQL. “Higher-valued” computations are good candidates for SQL CLR integration. We’ll revisit the SQL CLR usage question at various points in this chapter.

**Deployment**

Before you can test your SQL CLR code, you must deploy the assembly containing it and register the individual functions that you want recognized as stored procedures. A number of deployment methods are at your disposal; we will pause to cover them now, before discussing testing of your stored procedures and the other four basic SQL CLR entities.

**Deploying Your Assembly**

As mentioned earlier, Visual Studio deploys the SQL Server project version of the sample code when you build, start, or step through the project or use the Build/Deploy function on Visual Studio’s main menu. If you’re working with the SQL Server project version of the samples, go ahead and use the Deploy option or one of the Start or Build options in Visual Studio now.

For deploying the Class Library project version, assuming `C:\Programming\SQL2005\Chapter03` as this chapter’s sample code parent directory, you can execute the following T-SQL command...
Part I: Design Fundamentals and Core Technologies

from within Management Studio:

```sql
CREATE ASSEMBLY Chapter03
AUTHORIZATION dbo
FROM 'C:\ProgrammingSQL2005\Chapter03\VS\Chapter03Manual\Chapter03\bin\Debug\Chapter03.dll'
WITH PERMISSION_SET = SAFE
GO
```

The `AUTHORIZATION` clause allows you to specify a name or role to which ownership of the assembly is assigned. The default authorization is that of the current user, and because you are most likely logged in as dbo for AdventureWorks, in this case the clause is unnecessary (which is why we omitted it from our previous example).

The meaning and effect of the `WITH PERMISSION_SET` clause are discussed at the end of this chapter. For now, just note that this clause allows you to specify the security permissions with which your assembly runs. As with the `AUTHORIZATION` clause, in this case the `WITH PERMISSION_SET` clause is technically unnecessary because `SAFE` is the default `PERMISSION_SET` value used when a `CREATE ASSEMBLY` command is executed.

If your assembly has dependencies on other assemblies, SQL Server looks to see if those assemblies have already been loaded into the database and, if so, confirms that their ownership is the same as that of the specified assembly. If the dependent assemblies have not yet been loaded into the database, SQL Server looks for them in the same folder as the specified assembly. If it finds all dependent assemblies in that location, it loads them and assigns them the same ownership as the primary assembly. If it does not find the dependent assemblies in that folder, the `CREATE ASSEMBLY` command will fail.

You can supply a string expression instead of a literal in the `FROM` clause, allowing for some interesting data-driven possibilities. For example, you could fetch an assembly path reference from a table in your database. It is also possible to supply a bitstream in the `FROM` clause instead of a file specification. You do this by specifying a varbinary literal value or expression (or a comma-delimited list of varbinary values or expressions, when dependent assemblies must be specified) that contains the actual binary content of your assembly (or assemblies). This allows the creation of a database, including any CLR assemblies it contains, to be completely scripted, without requiring distribution of actual assembly files. The binary stream can be embedded in the script itself or, using an expression, it can be fetched from a table in a database.

**More Info** See SQL Server Books Online for more information on this option.

In addition to using Visual Studio deployment and the T-SQL `CREATE ASSEMBLY` command, you can upload the assembly into your database interactively from Management Studio. Simply right-click the `servername\AdventureWorks\Programmability\Assemblies` node in the Object Explorer (where `servername` is the name of your server) and choose New Assembly... from the shortcut menu. The New Assembly dialog box, shown in Figure 3-7, appears.
Figure 3-7  The Management Studio New Assembly dialog box

Type the assembly path and file name in the Path To Assembly text box, or use the Browse... button to specify it interactively. You can specify AUTHORIZATION and WITH PERMISSION_SET details in the Assembly Owner text box (using the ellipsis button, if necessary) and the Permission Set combo box, respectively.

Regardless of the deployment method you use, once your assembly has been added to your database, it becomes an integral part of that database and its underlying MDF file. This means if your database is backed up and restored, or xcopy deployed, any assemblies within it move along with the data itself and need not be manually added as a subsequent step.

Deploying Your Stored Procedures

In the SQL Server project version of the sample code, deployment of all the stored procedures is handled by Visual Studio when the assembly itself is deployed. This is due to the application of the SqlProcedure attribute to the functions in class StoredProcedures (found in file spTest.cs). The SqlProcedure attribute accepts an optional Name parameter, the value of which is the actual callable stored procedure name. If you do not supply a value for the Name parameter, the name of the .NET function is used as the stored procedure name.

The SqlProcedure attribute is used only by Visual Studio in SQL Server projects. Therefore, it has been removed from the source code in the Class Library project. Deploying the stored procedures from that version of the source code requires issuing a CREATE PROCEDURE T-SQL command using the new EXTERNAL NAME clause to specify the assembly, fully qualified class name specifier, and function name. For example, to load the Class Library version of spContacts, you would issue the following command.
The preceding command specifies that function `spContacts`, in class `StoredProcedures`, in the loaded assembly with T-SQL name `Chapter03`, should be registered as a CLR stored procedure callable under the name `spContacts`.

**Note** All necessary `CREATE PROCEDURE` commands for the Class Library project version of the sample code are contained in the `CreateObjects.sql` script in the Management Studio project supplied with the sample code. You will need to run that script in order to execute the various SQL CLR entities implemented in the Class Library project.

Note that had the CLR stored procedure been written in Visual Basic .NET rather than C#, the class name specifier would change to `Chapter03.StoredProcedures`. This would necessitate a change to the deployment T-SQL code as follows:

```
CREATE PROCEDURE spContacts
AS EXTERNAL NAME Chapter03.[Chapter03.StoredProcedures].spContacts
```

In Visual Basic projects, the default namespace for a project itself defaults to the project name, as does the assembly name. The class within the project must be referenced using the default namespace as a prefix. Because the class specifier is a multipart dot-separated name, it must be enclosed within square brackets so that SQL Server can identify it as a single indivisible name. Because C# projects handle the default namespace setting a little differently, the namespace prefix is not used in the class specifier for C# assemblies.

One last point before we discuss how to test your now-deployed CLR stored procedures. It is important to realize that the class specifier and function name in the `EXTERNAL NAME` clause are case-sensitive and that this is true even for assemblies developed in Visual Basic .NET. Although this point perplexed us quite a bit at first, it does make sense in hindsight. SQL Server searches for your subs functions within your assemblies, not within your source code. In other words, it’s looking within Microsoft Intermediate Language (MSIL) code, not Visual Basic .NET or C# source code. Because MSIL is case-sensitive (it has to be, to support case-sensitive languages like C#), SQL Server must be as well as it searches for a specific class and sub/function.

The fact that SQL Server is not case sensitive by default (even though it once was) and that Visual Basic .NET is not a case-sensitive language is of no import! If you attempt to register a sub/function and you receive an error that it cannot be found within the assembly, double-check that the case usage in your command matches that of your source code.
Chapter 3: An Overview of SQL CLR

Testing Your Stored Procedures

With your assembly and stored procedures now deployed, you’re ready to run and test them. Typically, you should do this from Management Studio; however, Visual Studio SQL Server projects allow you to test your SQL CLR code from Visual Studio itself. When you create a Visual Studio SQL Server project, a folder called Test Scripts is created as a subdirectory in your source code directory. Within that subdirectory, Visual Studio creates a script file called Test.sql. If you look at that file, you will see that it contains commented instructions as well as commented sample T-SQL code for testing stored procedures, functions, and UDTs. It also contains an uncommented generic `SELECT` command that echoes a text literal to the caller.

Visual Studio connects to your database and runs this script immediately after your assembly is deployed, and the output from the script appears in Visual Studio’s Output window. This allows you to execute any number of T-SQL commands directly from Visual Studio without having to switch to another tool. Although this approach is much less interactive than a Management Studio query window, it allows you to run quick tests against your code. It is especially useful for regression testing—that is, confirming that a new version of your assembly does not break older, critical functionality.

The file extension of the script must be `.sql`, but otherwise the name of the file is inconsequential. You can have multiple script files in the Test Scripts folder. To add a new one, right-click the Test Scripts folder node or the project node in the Solution Explorer window and select the Add Test Script option from the shortcut menu. Only one script can be active at one time, and as soon as you have more than one script, you must specify which one is active. To make a script active, simply right-click its node in the Solution Explorer window and select the Set As Default Debug Script option from its shortcut menu. When you do so, the node is displayed in bold. You may run or debug a script even if it is not the active script. To do so, right-click its node in the Solution Explorer window and select the Debug Script option from its shortcut menu.

**Warning** At press time, there appears to be an anomaly in the working of the test script facility and the Output window in Visual Studio 2005. Simply put, if your test script executes a query (whether it be a T-SQL `SELECT` command or a call to a stored procedure) that returns a column of type uniqueidentifier (GUID), the query’s result set will not appear in the Output window, and execution of the test script might hang Visual Studio. For this reason, you should avoid calling the sample code CLR stored procedures `spContactsQuick` and `spContacts` (both of which perform a `SELECT * FROM Person.Contact` query and thus retrieve the rowguid column, which is of type uniqueidentifier) from your test script and instead test these procedures from SQL Server Management Studio, where the anomaly does not occur. You can safely call `spContactCount`, which simply performs a `SELECT COUNT(*) FROM Person.Contact` query, from your Visual Studio test script. Alternatively, you can modify `spContactsQuick` and/or `spContacts` to select specific columns from the Person.Contact table, making sure that `rowguid` is not one of them.
If you’re working with the Class Library version of the sample code, you must test the stored procedures from Management Studio or another SQL Server query tool. Even if you are working with the SQL Server project version, you’ll find that testing your SQL CLR code in Management Studio provides a richer experience and more flexibility.

The script file TestStoredProcs.sql in the Management Studio project supplied with the sample code will run both of our CLR stored procedures (spContactCount and spContacts). Open the file in Management Studio, and click the Execute button on the SQL Editor toolbar, choose the Query/Execute option on the main menu, or press F5. (You can also right-click the query window and select Execute from the shortcut menu.)

When the script runs, you should see the single-valued result of the spContactCount stored procedure appear first, as shown in Figure 3-8. Note that the column name ContactCount appears on the Results tab and recall that this is a direct result of your using the SqlMetaData object in the CLR code. Below the spContactCount result, you will see the results from the spContacts stored procedure come in. Because the Person.Contact table has almost 20,000 rows, these results might take some time to flow in.

![Figure 3-8 TestStoredProcs.sql script code and results](image)

Even while the results are coming in, the “Starting data dump” status message should be visible on the Messages tab (or on the Results tab if you’re using Management Studio’s Results To Text option). Once all rows have been fetched, you should see the “Data dump complete” message appear as well. If you get impatient and want to abort the query before all rows have been fetched, you can use the Cancel Executing Query button on the SQL Editor toolbar or the Query/Cancel Executing Query option on the main menu; you can also use the Alt+Break keyboard shortcut.
We have yet to cover CLR functions, triggers, aggregates, and UDTs, but you have already learned most of the skills you need to develop SQL CLR code. You have learned how to create Visual Studio SQL Server projects and use its autodeployment and test script features. You have also learned how to develop SQL CLR code in standard Class Library projects and to use T-SQL commands and Management Studio to deploy the code for you. You’ve learned about the subtle differences between deploying C# code and Visual Basic .NET code, and we’ve covered the case-sensitive requirements of T-SQL-based deployment.

With all this under your belt, we can cover the remaining four basic SQL CLR entities relatively quickly.

**CLR Functions**

Let’s take everything we’ve discussed about CLR stored procedures and deployment and apply it to CLR functions. As any programmer knows, a function is a lot like a procedure, except that it returns a value (or an object). Mainstream .NET functions typically return .NET types. SQL CLR functions, on the other hand, must return a SqlType. So to start with, we need to make sure our classes that implement SQL CLR functions import/use the System.Data.SqlTypes namespace. The SQL Server Project template for User Defined Functions contains the appropriate using code by default; you must add the code manually to standard Class Library code.

Once the namespace is imported, you can write the functions themselves. In Visual Studio SQL Server Projects, they should be decorated with the SqlFunction attribute; this attribute accepts an optional name parameter that works identically to its SqlProcedure counterpart. In our sample code, we will not supply a value for this parameter. SqlFunction is used by Visual Studio SQL Server projects for deployment of your SQL CLR functions, but for scalar-valued functions in Class Library projects it is optional, so it appears in the Class Library sample code only for our table-valued function (described later).

Listing 3-3, which shows the code for function fnHelloWorld from fnTest.cs in the sample project, implements a simple “Hello World” function that returns a value of type SqlString.

**Listing 3-3 fnHelloWorld from fnTest.cs**

```csharp
[SqlFunction()]
public static SqlString fnHelloWorld()
{
    return new SqlString("Hello World");
}
```

Notice that SqlType objects require explicit instantiation and constructor value passing; you cannot simply declare and assign values to them. The code in Listing 3-3 instantiates a SqlString object inline within the return statement to avoid variable declaration.
A function that returns a hardcoded value is of little practical use. Typically, functions are passed values and perform calculations on them, and they are often used from within T-SQL statements, in effect as extensions to the functions built into the T-SQL language itself.

Listing 3-4, which shows the code for function `fnToCelsius` in fnTest.cs in the sample project, implements a Fahrenheit-to-Celsius conversion function.

Listing 3-4  `fnToCelsius` from fnTest.cs

```csharp
[SqlFunction()]
public static SqlDecimal fnToCelsius(SqlInt16 Fahrenheit)
{
    return new SqlDecimal((((Int16)Fahrenheit) - 32) / 1.8);
}
```

The function accepts a Fahrenheit temperature (as a `SqlInt16`), converts it to Celsius, and returns it (as a `SqlDecimal`). Notice that the code casts the input parameter from a `SqlInt16` to a .NET `Int16`, applies a Fahrenheit-to-Celsius conversion formula, and passes the result to the constructor of a new `SqlDecimal` object.

Deployment of these functions is automatic in the Visual Studio SQL Server project version of our sample code. For the Class Library version, use the T-SQL `CREATE FUNCTION` command in a similar fashion to our use of the `CREATE PROCEDURE` command in the previous section, but include a data type specification for the return value. For example, to deploy the `fnHelloWorld` function, you would use this command:

```sql
CREATE FUNCTION fnHelloWorld()
RETURNS NVARCHAR(4000) WITH EXECUTE AS CALLER
AS EXTERNAL NAME Chapter03.UserDefinedFunctions.fnHelloWorld
```

Notice the use of data type `NVARCHAR(4000)` to correspond with the `SqlString` type used in the function’s implementation. The `WITH EXECUTE AS CALLER` clause specifies that the SQL CLR function should execute under the caller’s identity.

Tip  You can enter the `CREATE FUNCTION` command yourself, but all such necessary commands for the sample code SQL CLR functions are contained in the CreateObjects.sql script file in the Management Studio project supplied with the sample code.

You can test these functions using the Visual Studio SQL Server project test script or in Management Studio. Use the following query in your test script or a Management Studio query window to test the two functions. (You can also run the TestScalarFunctions.sql script file in the Management Studio sample project.)
T-SQL functions can return result sets as well as scalar values. Such functions are called *table-valued functions* (TVFs). Writing SQL CLR TVFs is possible, although you do so differently than you would CLR scalar-valued functions or CLR stored procedures. CLR TVFs must return a type that implements the .NET interface `IEnumerable`, and they must declare a "FillRow" method that interprets that type and converts an instance of the type to a table row.

Listing 3-5, which shows the code for functions `fnPortfolioTable` and `FillTickerRow` in `fnTest.cs` in the sample project, implements a TVF called `fnPortfolioTable`.

**Listing 3-5  fnPortfolioTable and FillTickerRow from fnTest.cs**

```csharp
[SqlFunction(FillRowMethodName="FillTickerRow", TableDefinition="TickerSymbol nvarchar(5), Value decimal")]
public static System.Collections.IEnumerable fnPortfolioTable(SqlString TickersPacked)
{
    string[] TickerSymbols;
    object[] RowArr = new object[2];
    object[] CompoundArray = new object[3];
    char[] parms = new char[1];
    parms[0] = ';';
    TickerSymbols = TickersPacked.Value.Split(parms);
    RowArr[0] = TickerSymbols[0];
    RowArr[1] = 1;
    CompoundArray[0] = RowArr;
    RowArr = new object[2];
    RowArr[0] = TickerSymbols[1];
    RowArr[1] = 2;
    CompoundArray[1] = RowArr;
    RowArr = new object[2];
    RowArr[0] = TickerSymbols[2];
    RowArr[1] = 3;
    CompoundArray[2] = RowArr;

    return CompoundArray;
}

public static void FillTickerRow(object row, ref SqlString TickerSymbol, ref SqlDecimal Value)
{
    object[] rowarr = (object[]) row;
    TickerSymbol = new SqlString((string)rowarr[0]);
    Value = new SqlDecimal(decimal.Parse(rowarr[1].ToString()));
}
```
Rather than implementing its own IEnumerable-compatible type, fnPortfolioTable uses an array. This is perfectly legal because arrays implement IEnumerable. Function fnPortfolioTable accepts a semicolon-delimited list of stock ticker symbols and returns a table with each ticker symbol appearing in a separate row as column TickerSymbol and a value for the ticker as column Value. The structure of the returned table is declared in the TableDefinition parameter of the SqlFunction attribute in SQL Server projects and in the CREATE FUNCTION T-SQL command for Class Library projects. The assigned values are hardcoded, and only three rows are returned, regardless of how many ticker symbols are passed in. As with our other samples, this one is more useful as a teaching tool than as a practical application of TVFs.

Arrays are the name of the game here. First the String.Split method is used to crack the delimited ticker list into an array of single ticker strings. Then the TVF structures the data so that each element in the return value array (CompoundArray) is itself a two-element array storing a single ticker symbol and its value. The function code itself needs only to return CompoundArray. Next, the FillTickerRow function (named in the FillRowMethodName parameter of the SqlFunction attribute) takes each two-element array and converts its members to individual scalars that correspond positionally to the columns in the TableDefinition argument of the SqlFunction attribute.

Because the FillRowMethodName parameter of the SqlFunction attribute is required by SQL Server, we have decorated the Class Library version of function fnPortfolioTable with that attribute, supplying a value for that one parameter. In the SQL Server project version, we also supply a value for the TableDefinition parameter to enable autodeployment of the TVF.

As with the other functions, deployment of this function is performed by Visual Studio in the SQL Server project sample code. For the Class Library version, you can deploy the function using the following T-SQL command (also contained in the CreateObjects.sql script file):

```
CREATE FUNCTION fnPortfolioTable(@TickersPacked [NVARCHAR](4000))
RETURNS TABLE ( 
    TickerSymbol NVARCHAR(5),
    VALUE DECIMAL
) 
WITH EXECUTE AS CALLER
AS EXTERNAL NAME Chapter03.UserDefinedFunctions.fnPortfolioTable
```

As with fnHelloWorld, we have mapped the SqlDbType data type to an NVARCHAR(4000), this time for one of the input parameters. Because fnPortfolioTable is a TVF, its return type is declared as TABLE, with inline specifications for the table's definition.

Use the following query in your Visual Studio test script or a Management Studio query window to test the TVF (or run the TestTableValuedFunction.sql script file in the Management Studio sample project):

```
SELECT * FROM fnPortfolioTable('IBM;MSFT;SUN')
```
The following data should be returned:

<table>
<thead>
<tr>
<th>TickerSymbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>1</td>
</tr>
<tr>
<td>MSFT</td>
<td>2</td>
</tr>
<tr>
<td>SUN</td>
<td>3</td>
</tr>
</tbody>
</table>

**CLR Triggers**

T-SQL triggers are really just stored procedures that are called by SQL Server at specific times and query values in the “inserted” and “deleted” pseudo-tables. SQL CLR triggers are similar to SQL CLR stored procedures, and they can be created for all data manipulation language (DML) actions (updates, inserts, and deletes).

SQL Server 2005 introduces the concept of data definition language (DDL) triggers, which handle actions such as CREATE TABLE and ALTER PROCEDURE. Like DML triggers, DDL triggers can be implemented in T-SQL or SQL CLR code. We will cover SQL CLR DML and DDL triggers in this section.

SQL CLR DML triggers, like their T-SQL counterparts, have access to the “inserted” and “deleted” pseudo-tables and must be declared as handling one or more specific events for a specific table or, under certain circumstances, a specific view. Also, they can make use of the SqlTriggerContext object (through the SqlContext object’s TriggerContext property) to determine which particular event (update, insert, or delete) caused them to fire and which columns were updated.

Once you latch on to these concepts, writing SQL CLR DML triggers is really quite simple. Listing 3-6, which shows the code for function trgUpdateContact from trgTest.cs in the sample project, shows the SQL CLR code for DML trigger trgUpdateContact, which is designed to function as a FOR UPDATE trigger on the Person.Contact table in the AdventureWorks database.

**Listing 3-6  trgUpdateContact from trgTest.cs**

```csharp
// [SqlTrigger(Target="Person.Contact", Event="for UPDATE")]
public static void trgUpdateContact()
{
    SqlTriggerContext TriggerContext = SqlContext.TriggerContext;
    String OldName = String.Empty;
    String NewName = String.Empty;
    String OldDate = String.Empty;
    String NewDate = String.Empty;
    SqlConnection conn = new SqlConnection("context connection=true");
    SqlCommand cmOld = new SqlCommand("SELECT FirstName, ModifiedDate from DELETED", conn);
    SqlCommand cmNew = new SqlCommand("SELECT FirstName, ModifiedDate from INSERTED", conn);
    conn.Open();
    SqlDataReader drOld = cmOld.ExecuteReader();
    if (drOld.Read())
```
This CLR DML trigger queries the “deleted” and “inserted” tables and echoes back the “before and after” values for the FirstName and ModifiedDate columns when a row is updated. It does so not by piping back SqlDataReader objects but by fetching values from them and echoing back the values as text using the SqlPipe object’s Send method. The trigger code also uses the TriggerContext.IsUpdatedColumn method to echo back a list of all columns in the Person.Contact table and whether each was updated.

To deploy the trigger automatically, you would normally configure a SqlTrigger attribute and apply it to the .NET function that implements the trigger. Because DML triggers are applied to a target object (a table or a view) and an event (for example, “for update” or “instead of insert”), the SqlTrigger attribute has parameters for each of these pieces of information and you must supply values for both. The SqlTrigger attribute deploys only a single copy of the trigger, but you can use T-SQL to deploy the same code as a separate trigger for a different event and/or table. Each separate deployment of the same code is assigned a unique trigger name.

Unfortunately, a bug in Visual Studio prevents the SqlTrigger attribute from being used for target objects not in the dbo schema. (For example, our table, Person.Contact, is in the Person schema rather than the dbo schema.) This is because the value for the Target parameter is surrounded by square brackets when Visual Studio generates its T-SQL code (generating, for example, [Person.Contact], which will cause an error). It is for this reason that the SqlTrigger attribute code is commented out in Listing 3-6. A workaround to this problem is available.
through the use of pre-deployment and post-deployment scripts, which we will discuss shortly.

**Important** Although you might be tempted to work around the Visual Studio schema bug by supplying a Target value of Person.[Contact] instead of Person.Contact, rest assured that this will not work. You may initiate a trace in SQL Server Profiler to observe the erroneous T-SQL generated by Visual Studio in either scenario.

Although Listing 3-6 does not demonstrate it, you can create a single piece of code that functions as both the update and insert trigger for a given table. You can then use the TriggerContext object’s TriggerAction property to determine exactly what event caused the trigger to fire, and you can execute slightly different code accordingly. Should you wish to deploy such a CLR trigger using the SqlTrigger attribute, you would set its Event parameter to “FOR UPDATE, INSERT”.

The T-SQL command to register a .NET function as a SQL CLR trigger for the update event only is as follows:

```sql
CREATE TRIGGER trgUpdateContact
ON Person.Contact
FOR UPDATE
AS EXTERNAL NAME Chapter03.Triggers.trgUpdateContact
```

**Note** All necessary CREATE TRIGGER commands for the Class Library project version of the sample code are contained in the CreateObjects.sql script in the Management Studio project supplied with the sample code.

Beyond using such T-SQL code in Management Studio, there is a way to execute this T-SQL command from Visual Studio, and thus work around the SqlTrigger non-dbo schema bug. An essentially undocumented feature of Visual Studio SQL Server projects is that they allow you to create two special T-SQL scripts that will run immediately before and immediately after the deployment of your assembly. To use this feature, simply create two scripts, named PreDeployScript.sql and PostDeployScript.sql, in the root folder (not the Test Scripts folder) of your project. Although not case-sensitive, the names must match verbatim.

**Tip** You can create the PreDeployScript.sql and PostDeployScript.sql scripts outside of Visual Studio and then add them to your project using Visual Studio’s Add Existing Item… feature. You can also add them directly by right-clicking the project node or Test Scripts folder node in the Solution Explorer, choosing the Add Test Script option from the shortcut menu, renaming the new scripts, and dragging them out of the Test Scripts folder into the root folder of your project.
To use this feature to work around the SqlTrigger non-dbo schema bug, insert the preceding CREATE TRIGGER code in your PostDeployScript.sql file and insert the following T-SQL code into your PreDeployScript.sql:

```sql
IF EXISTS (SELECT * FROM sys.triggers WHERE object_id = OBJECT_ID(N'Person.trgUpdateContact'))
DROP TRIGGER Person.trgUpdateContact
```

Regardless of deployment technique, you can use the following query in your Visual Studio test script or a Management Studio query window to test the trigger (this T-SQL code can be found in the TestTriggers.sql script file in the Management Studio project):

```sql
UPDATE Person.Contact
SET FirstName = 'Gustavoo'
WHERE ContactId = 1
```

When you run the preceding query, you will notice that the trigger is actually run twice. This is because the AdventureWorks Person.Contact table already has a T-SQL update trigger, called uContact. Because uContact itself performs an update on the ModifiedDate column of Person.Contact, it implicitly invokes a second execution of trgUpdateContact. By looking at the output of trgUpdateContact, you can confirm that the FirstName column is updated on the first execution (by the test query) and the ModifiedDate column is modified on the second execution (by trigger uContact). The two executions' output might appear out of order, but the values of ModifiedDate will make the actual sequence clear.

If you place the TriggerContext object's TriggerAction property in a comparison statement, IntelliSense will show you that there is a wide array of enumerated constants that the property can be equal to, and that a majority of these values correspond to DDL triggers. This demonstrates clearly that SQL CLR code can be used for DDL and DML triggers alike.

In the case of DDL triggers, a wide array of environmental information might be desirable to determine exactly what event caused the trigger to fire, what system process ID (SPID) invoked it, what time the event fired, and other information specific to the event type such as the T-SQL command that caused the event. The SqlTriggerContext object's EventData property can be queried to fetch this information. The EventData property is of type SqlXml; therefore it, in turn, has a CreateReader method and a Value property that you can use to fetch the XML-formatted event data as an XmlReader object or a string, respectively.

The code in Listing 3-7, taken from function trgCreateTable in trgTest.cs in the sample project, shows the SQL CLR code for the DDL trigger trgCreateTable registered to fire for any CREATE TABLE command executed on the AdventureWorks database.
Listing 3-7  trgCreateTable from trgTest.cs

```csharp
[SqlTrigger(Target = "DATABASE", Event = "FOR CREATE_TABLE")]
public static void trgCreateTable()
{
    SqlTriggerContext TriggerContext = SqlContext.TriggerContext;
    if (!(TriggerContext.EventData == null))
    {
    }
}
```

The code interrogates the Value property of `SqlContext.TriggerContext.EventData`, casts it to a string, and pipes that string back to the client. Note that the `SqlTrigger` attribute is not commented out in this case because a schema prefix is not used in the `Target` parameter value. Thus, you can use attribute-based deployment in the SQL Server project or the following command for the Class Library version:

```sql
CREATE TRIGGER trgCreateTable
ON DATABASE
FOR CREATE_TABLE
AS EXTERNAL NAME Chapter03.Triggers.trgCreateTable
```

Use the following T-SQL DDL command in your Visual Studio test script or a Management Studio query window to test the DDL trigger. (You can find this code in the TestTriggers.sql script file in the sample Management Studio project.)

```sql
CREATE TABLE Test (low INT, high INT)
DROP TABLE Test
```

Your result should appear similar to the following:

```
<Event_INSTANCE>
  <EventType>CREATE_TABLE</EventType>
  <PostTime>2006-04-29T16:37:50.690</PostTime>
  <SPID>54</SPID>
  <ServerName>CGH-T42AB</ServerName>
  <LoginName>CGH-T42AB\AndrewB</LoginName>
  <UserName>dbo</UserName>
  <DatabaseName>AdventureWorks</DatabaseName>
  <SchemaName>dbo</SchemaName>
  <ObjectName>Test</ObjectName>
  <ObjectType>TABLE</ObjectType>
  <TSQLCommand>
    <CommandText>CREATE TABLE Test (low INT, high INT)</CommandText>
  </TSQLCommand>
</EVENT_INSTANCE>
```
CLR Aggregates

T-SQL has a number of built-in aggregates, such as \textit{SUM()}, \textit{AVG()}, and \textit{MAX()}, but that set of built-in functions is not always sufficient. Luckily, the SQL CLR features in SQL Server 2005 allow us to implement user-defined aggregates in .NET code and use them from T-SQL. User-defined aggregates can be implemented only in SQL CLR code; they have no T-SQL equivalent. Because aggregates tend to perform computation only, they provide an excellent use case for SQL CLR code. As it turns out, they are also quite easy to build.

At first, aggregates feel and look like functions because they accept and return values. In fact, if you use an aggregate in a non-data-querying T-SQL call (for example, \textit{SELECT SUM(8)}), you are in fact treating the aggregate as if it were a function. The thing to remember is that the argument passed to an aggregate is typically a column, and so each discrete value for that column, for whichever \textit{WHERE}, \textit{HAVING}, \textit{ORDER BY}, and/or \textit{GROUP BY} scope applies, gets passed into the aggregate. It is the aggregate’s job to update a variable, which eventually will be the return value, as each discrete value is passed to it.

CLR aggregates require you to apply the \texttt{SqlUserDefinedAggregate} attribute to them. The \texttt{SqlUserDefinedAggregate} attribute accepts a number of parameters, but all of them are optional except \texttt{Format}. In our example, we will use the value \texttt{Format.Native} for the \texttt{Format} parameter. For more advanced scenarios, you might want to study SQL Server Books Online to acquaint yourself with the other parameters this attribute accepts. Sticking with \texttt{Format.Native} for the \texttt{Format} parameter is sufficient for many scenarios.

Unlike the \texttt{SqlProcedure}, \texttt{SqlFunction}, and \texttt{SqlTrigger} attributes, the \texttt{SqlUserDefinedAggregate} attribute is required by SQL Server for your class to be eligible for use as an aggregate. Visual Studio SQL Server projects do use this attribute for deployment, and the attribute is included in the aggregate template, but it also must be used in generic Class Library project code in order for T-SQL registration of the aggregate to succeed.

Aggregate classes must have four methods: \texttt{Init}, \texttt{Accumulate}, \texttt{Merge}, and \texttt{Terminate}. The \texttt{Accumulate} method accepts a SQL type, the \texttt{Terminate} method returns one, and the \texttt{Merge} method accepts an object typed as the aggregate class itself.

The \texttt{Accumulate} method handles the processing of a discrete value into the aggregate value, and the \texttt{Terminate} method returns the final aggregated value after all discrete values have been processed. The \texttt{Init} method provides startup code, typically initializing a class-level private variable that will be used by the \texttt{Accumulate} method. The \texttt{Merge} method is called in a specific multi-threading scenario, which we will describe later on.
Just to be perfectly clear, your aggregate class will not implement an interface to supply these methods; you must create them to meet what we might term the “conventions” that are expected of SQL CLR aggregate classes (as opposed to a “contract” with which they must comply). When you develop your code in a Visual Studio 2005 SQL Server project, the Aggregate template includes stubs for these four methods as well as the proper application of the SqlUserDefinedAggregate attribute.

Creating your own aggregates is fairly straightforward, but thinking through aggregation logic can be a bit confusing at first. Imagine you want to create a special aggregate called Bakers-Dozen that increments its accumulated value by 1 for every 12 units accumulated (much as a baker, in simpler times, would throw in a free 13th donut when you ordered 12). By using what you now know about CLR aggregates and combining that with integer division, you can implement a BakersDozen aggregate quite easily. Listing 3-8, the code from struct BakersDozen in aggTest.cs in the sample project, contains the entire implementation of the aggregate BakersDozen.

Listing 3-8  struct BakersDozen from aggTest.cs

```csharp
[Serializable]
public struct BakersDozen
{
    private SqlInt32 DonutCount;

    public void Init()
    {
        DonutCount = 0;
    }

    public void Accumulate(SqlInt32 Value)
    {
        DonutCount += Value + ((Int32)Value) / 12;
    }

    public void Merge(BakersDozen Group)
    {
        DonutCount += Group.DonutCount;
    }

    public SqlInt32 Terminate()
    {
        return DonutCount;
    }
}
```

The code here is fairly straightforward. The private variable DonutCount is used to track the BakersDozen-adjusted sum of items ordered, adding the actual items-ordered value and incrementing the running total by the integer quotient of the ordered value divided by 12. By this
Part I: Design Fundamentals and Core Technologies

logic, bonus items are added only when an individual value equals or exceeds a multiple of 12. Twelve includes a full dozen, and so would 13. Twenty-four includes two dozen, and so would 27. Two individual orders of 6 items each would not generate any bonus items because a minimum of 12 items must be ordered in a line item to qualify for a bonus.

To deploy the aggregate, use attribute-based deployment in the SQL Server project or the following command for the Class Library version:

```sql
CREATE AGGREGATE BakersDozen
(@input int)
RETURNS int
EXTERNAL NAME Chapter03.BakersDozen
```

Notice that no method name is specified because the aggregate is implemented by an entire class rather than an individual function. Notice also that the return value data type must be declared as the data type of the values this aggregate function will process. The @input parameter acts as a placeholder, and its name is inconsequential. Note that aggregates can be built on SQL CLR types (covered in the next section) as well as SQL scalar types.

**Note** The preceding `CREATE AGGREGATE` command for the Class Library project version of the sample code is contained in the CreateObjects.sql script in the Management Studio project supplied with the sample code.

To see the aggregate work, first run the CreateTblAggregateTest.sql script file in the Management Studio sample project to create a table called AggregateTest with columns OrderItemId, OrderId, and ItemsOrdered and several rows of data, as shown here:

```sql
CREATE TABLE tblAggregateTest(
    [OrderItemId] [int] IDENTITY(1,1) NOT NULL,
    [OrderId] [int] NULL,
    [ItemsOrdered] [int] NOT NULL
)
GO
INSERT INTO tblAggregateTest VALUES (1,2)
INSERT INTO tblAggregateTest VALUES (1,4)
INSERT INTO tblAggregateTest VALUES (2,1)
INSERT INTO tblAggregateTest VALUES (2,12)
INSERT INTO tblAggregateTest VALUES (3,3)
INSERT INTO tblAggregateTest VALUES (3,2)
```
With such a table built, use the following T-SQL DDL command in your Visual Studio test script or a Management Studio query window to test the aggregate function:

```sql
SELECT
    OrderId,
    SUM(ItemsOrdered) AS SUM,
    dbo.BakersDozen(ItemsOrdered) AS BakersDozen
FROM tblAggregateTest
GROUP BY OrderId
```

For each distinct value in the OrderId column, this query effectively uses our CLR code under the following algorithm:

- Call `Init()`.
- Call `Accumulate` once for each row with the same `OrderId` value, passing it that row's value of the `ItemsOrdered` column.
- Call `Terminate` upon a change in the `OrderId` value to retrieve the aggregated value that the query will pipe to the client.

The results should be as follows:

<table>
<thead>
<tr>
<th>OrderId</th>
<th>SUM</th>
<th>BakersDozen</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

By including the built-in T-SQL aggregate `SUM` in our query, we can see how many bonus items were added. In this case, for OrderId 2, a single bonus item was added, due to one row in the table with the following values:

<table>
<thead>
<tr>
<th>OrderItemId</th>
<th>OrderId</th>
<th>ItemsOrdered</th>
</tr>
</thead>
<tbody>
<tr>
<td>421</td>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

All the other rows contain `ItemsOrdered` values of less than 12, so no bonus items were added for them.

Because SQL Server sometimes segments the work required to satisfy a query over multiple threads, the query processor might need to execute your aggregate function multiple times for a single query and then merge the results together. For your aggregate to work properly in this scenario, you must implement a `Merge` method.

The `Merge` method takes the result of one thread's aggregation and merges it into the current thread's aggregation. The calculation required to do this could be complicated for some aggregates; in our case, we simply added the `DonutCount` value from the secondary thread's aggregate (accessible via the `Group` input parameter) to our own. There is no need to add bonus
items because they would have been added in the individual Accumulate calls on the secondary thread. Simple addition is all that’s required. An aggregate that calculated some type of average, or tracked the largest value in the data series supplied, for example, would require more complex merge code.

Don’t forget that aggregates can be passed scalar values and can be used from T-SQL without referencing a table. Your aggregate must accommodate this scenario, even if it seems impractical. In the case of BakersDozen, single scalar values are easily handled. To see for yourself, try executing the following table-less T-SQL query:

```
SELECT dbo.BakersDozen(13)
```

You will see that it returns the value 14.

**Note** The TestAggregate.sql script file in the Management Studio project contains both aggregate-testing queries.

Aggregates are an excellent use of SQL CLR programming. Because they are passed data values to be processed, they typically perform only computational tasks and no data access of their own. They consist of compiled CLR code, so they perform well, and unlike stored procedures, triggers, and functions, they cannot be implemented at all in T-SQL. That said, you must still make your aggregate code, especially in the Accumulate method, as “lean and mean” as possible. Injecting your own code into the query processor’s stream of work is an honor, a privilege, and a significant responsibility. Take that responsibility seriously, and make sure that your code is as low-impact as possible.

**CLR Types**

The last SQL CLR feature for us to explore is user-defined types (UDTs). This feature is perhaps the most interesting, yet also the most controversial. It’s interesting because, technically, it allows for storage of objects in the database. It’s controversial because it’s prone to abuse. CLR types were not implemented to allow developers to create object-oriented databases; they were created to allow multi-value or multi-behavior data types to be stored, retrieved, and easily manipulated.

CLR types have an 8 KB size limit. They also have certain indexing limitations, and their entire value must be updated when any of their individual property/field values is updated.

CLR type methods must be static. You cannot, therefore, call methods from T-SQL as instance methods; instead, you must use a special *TypeName::MethodName()* syntax. You can implement properties as you would in any conventional class and read from them or write to them from T-SQL using a standard variable.property/column.property dot-separated syntax.

Listing 3-9, the code from struct *typPoint* in typTest.cs in the sample project, shows the implementation of *typPoint*, a CLR type that can be used to store Cartesian coordinates in the database.

**Listing 3-9  struct *typPoint* from typTest.cs**

```csharp
[Serializable]
[SqlUserDefinedType(Format.Native)]
public struct typPoint : INullable
{
    private bool m_Null;
    private double m_x;
    private double m_y;

    public override string ToString()
    {
        if (this.IsNull)
            return "NULL";
        else
            return this.m_x + ":" + this.m_y;
    }

    public bool IsNull
    {
        get
        {
            return m_Null;
        }
    }

    public static typPoint Null
    {
        get
        {
            typPoint pt = new typPoint();
            pt.m_Null = true;
            return pt;
        }
    }

    public static typPoint Parse(SqlString s)
    {
        if (s.IsNull)
            return Null;
        else
```
Through the class's X and Y properties, you can process coordinates in a single database column or variable. You can assign coordinate values to an instance of the type as a colon-delimited string (for example, 3:4, by using the Parse method [implicitly]); you can read them back in the same format by using the ToString method. Once a value has been assigned, you can individually read or modify its X or Y portion by using the separate X and Y properties.

The class implements the INullable interface and its IsNull property. The Sum method demonstrates how to expose a static member and allow it to access instance properties by accepting an instance of the CLR type of which it is a member.

Notice that the class is a struct and that the Serializable and SqlUserDefinedType attributes have been applied to it. As with the SqlUserDefinedAggregate attribute, SqlUserDefinedType is required by SQL Server and appears in the Class Library sample code as well as the SQL Server project version. As with the SqlUserDefinedAggregate, we simply assign a value of Format.Native to the Format parameter and leave the other parameters unused.

More Info You might want to study SQL Server Books Online for information on using other parameters for this attribute.
Listing 3-10, the code from struct typBakersDozen in typTest.cs in the sample project, re-implements the BakersDozen logic we used in our aggregate example, this time in a UDT.

**Listing 3-10**  struct typBakersDozen from typTest.cs

```csharp
[Serializable]
[SqlUserDefinedType(Format.Native)]
public struct typBakersDozen : INullable
{
    private bool m_Null;
    private double m_RealQty;

    public override string ToString()
    {
        return (m_RealQty + (long)m_RealQty / 12).ToString();
    }

    public bool IsNull
    {
        get
        {
            return m_Null;
        }
    }

    public static typBakersDozen Null
    {
        get
        {
            typBakersDozen h = new typBakersDozen();
            h.m_Null = true;
            return h;
        }
    }

    public static typBakersDozen Parse(SqlString s)
    {
        if (s.IsNull)
            return Null;
        else
        {
            typBakersDozen u = new typBakersDozen();
            u.RealQty = double.Parse((string)s);
            return u;
        }
    }

    public static typBakersDozen ParseDouble(SqlDouble d)
    {
        if (d.IsNull)
            return Null;
        else
        
```
The `RealQty` and `AdjustedQty` properties allow the ordered quantity to be assigned a value and the adjusted quantity to be automatically calculated, or vice versa. The real quantity is the default "input" value, the adjusted quantity is the default "output" value of the type, and the `Parse` and `ToString` methods work accordingly. If the `AdjustedQty` property is assigned a value that is an even multiple of 12 (which would be invalid), that value is assigned to the `RealQty` property, forcing the `AdjustedQty` to be set to its passed value plus its integer quotient when divided by 12.

To deploy the UDTs, use attribute-based deployment for the SQL Server project. The script file `CreateObjects.sql` in the Management Studio project supplied with the sample code contains the T-SQL code necessary to deploy the Class Library versions of the UDTs. Here's the command that deploys `typPoint`:

```sql
CREATE TYPE typPoint
EXTERNAL NAME Chapter03.typPoint
```

The script file `TestTypPoint.sql` in the Management Studio project contains T-SQL code that tests `typPoint`. Run it and examine the results for an intimate understanding of how to work
with the type. The script file CreateTblPoint.sql creates a table with a column that is typed based on typPoint. Run it, and then run the script file TestTblPoint.sql to see how to manipulate tables that use SQL CLR UDTs.

The script file TestTypBakersDozen.sql contains T-SQL code that tests typBakersDozen. The ParseDouble method demonstrates how to implement a non-SqlString parse method. We named it ParseDouble because the Parse method itself cannot be overloaded. You must call ParseDouble explicitly as follows:

```sql
DECLARE @t AS dbo.typBakersDozen
SET @t = typBakersDozen::ParseDouble(12)
```

This is equivalent to using the default Parse method (implicitly) and assigning the string 12 as follows:

```sql
DECLARE @t AS dbo.typBakersDozen
SET @t = '12'
```

Notice that typBakersDozen essentially stores a value for the real quantity, and its properties are really just functions that accept or express that value in its native form or as an adjusted quantity. There is no backing variable for the AdjustedQty property, the get block of the AdjustedQty property merely applies a formula to the backing variable for RealQty and returns the result.

So typBakersDozen is not really an object with distinct properties, as typPoint is (with its admittedly simple ones). Because it merely implements a SqlDouble and adds some specialized functionality to it, typBakersDozen is a more appropriate implementation of CLR UDTs than is typPoint. In this vein, other good candidates for CLR UDTs include date-related types (for example, a type that stores a single underlying value for an annual quarter and accepts/presents its value as a calendar quarter or a fiscal quarter) and currency types (especially those with fixed rates of exchange between their native and converted values).

In general, you can think of CLR UDTs as “super scalars”—classes that wrap a scalar value and provide services and conversion functions for manipulating that scalar value and converting it among different interpretive formats or numbering systems. Do not think of SQL CLR UDTs as object-relational entities. This might seem counterintuitive, but consider the use of (de)serialization and the XML data type as more appropriate vehicles for storing objects in the database.

We have now investigated all five SQL CLR entities. Before we finish up, we need to discuss assembly security and ongoing maintenance of SQL CLR objects in your databases.
Security

Depending on the deployment method, you have numerous ways to specify what security level to grant a CLR assembly. All of them demand that you specify one of three permission sets:

- **Safe** Assembly can perform local data access and computational tasks only.
- **External Access** Assembly can perform local data access and computational tasks and also access the network, the file system, the registry, and environment variables. Although **External Access** is less restrictive than **Safe**, it still safeguards server stability.
- **Unsafe** Assembly has unrestricted permissions and can even call unmanaged code. This setting can significantly compromise SQL Server security; only members of the sysadmin role can create (load) unsafe assemblies.

When you deploy an assembly from Visual Studio, its security level is set to Safe by default. To change it, you can select the project node in the Solution Explorer window and set the Permission Level property in the Properties window by selecting Safe, External, or Unsafe from the combo box provided (Figure 3-9).

![Figure 3-9 The Permission Level property and its options in the Visual Studio 2005 Properties window](image)

Alternatively, you can right-click the project node in the Solution Explorer window and select Properties from the shortcut menu. You can also double-click the Properties node (or the My Project node in Visual Basic projects). Either action opens up the project properties designer. Select the designer’s Database tab and then select a permission set from the Permission Level combo box (Figure 3-10). (The same three options are available here as in the Properties window.)

To specify an assembly’s permission set using T-SQL, simply specify **SAFE**, **EXTERNAL_ACCESS**, or **UNSAFE** within the “**WITH PERMISSION_SET**” clause of the **CREATE ASSEMBLY** statement covered earlier in this chapter. Recall that our example used the default **SAFE** setting in this clause.

Finally, in the Management Studio New Assembly dialog box (shown earlier in Figure 3-7), you can select Safe, External Access, or Unsafe from the Permission Set combo box.
Examining and Managing CLR Types in a Database

Once deployed, your SQL CLR stored procedures, functions, triggers, aggregates, and user-defined types and their dependencies might become difficult to keep track of in your head. Luckily, you can easily perform discovery on deployed CLR entities using the Management Studio UI. All CLR objects in a database can be found in Management Studio’s Object Explorer window. To find them within the Object Explorer window’s tree view, first navigate to the \servername\Databases\databasename node (where servername and databasename are the names of your server and database, respectively). Refer to Table 3-1 for the subnodes of this node that contain each CLR entity.

Table 3-1 Finding CLR Objects in Object Explorer

<table>
<thead>
<tr>
<th>To view...</th>
<th>Look in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent node for SQL CLR stored procedures, DDL triggers, functions, aggregates, and UDTs</td>
<td>Programmability (see Figure 3-11)</td>
</tr>
<tr>
<td>Assemblies</td>
<td>Programmability\Assemblies (see Figure 3-12)</td>
</tr>
<tr>
<td>Stored procedures</td>
<td>Programmability\Stored Procedures (see Figure 3-13)</td>
</tr>
<tr>
<td>Functions</td>
<td>Programmability\Functions\Scalar-Valued Functions and Programmability\Functions\Table-Valued Functions (see Figure 3-14)</td>
</tr>
<tr>
<td>Aggregates</td>
<td>Programmability\Functions\Aggregate Functions (see Figure 3-14)</td>
</tr>
<tr>
<td>DML triggers</td>
<td>Tables\tablename\Triggers, where tablename is the name of the database table, including schema name, on which the trigger is defined (see Figure 3-15)</td>
</tr>
</tbody>
</table>
Table 3-1  Finding CLR Objects in Object Explorer

<table>
<thead>
<tr>
<th>To view...</th>
<th>Look in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDL triggers</td>
<td>Programmability\Database Triggers (see Figure 3-16) (also \servername\Server Objects\Triggers, where servername is the name of your server)</td>
</tr>
<tr>
<td>User-defined types</td>
<td>Programmability\Types\User-Defined Types (see Figure 3-17)</td>
</tr>
</tbody>
</table>

Figure 3-11  The SQL Server Management Studio Object Explorer window, with Programmability node highlighted

Figure 3-12  The Object Explorer window, with Assemblies node highlighted

Figure 3-13  The Object Explorer window, with CLR stored procedures highlighted
Figure 3-14  The Object Explorer window, with CLR table-valued, scalar-valued, and aggregate functions highlighted

Figure 3-15  The Object Explorer window, with CLR DML trigger highlighted
Bear in mind that you might need to use the Refresh shortcut menu option on the nodes listed in the table to see your CLR objects. If you’ve deployed or deleted any SQL CLR objects (as discussed shortly) since opening the Object Explorer’s connection to your database, the tree view will be out of date and will have to be refreshed. Notice that the tree view icons for CLR stored procedures and CLR DML triggers differ slightly from their T-SQL counterparts; they have a small yellow padlock on the lower-right corner.

Once you’ve located a CLR entity in the Object Explorer window, you can right-click its tree view node and generate CREATE, DROP, and in some cases ALTER scripts for it by selecting
the Script object type As option from the shortcut menu (where object type is the SQL CLR object type selected). The script text can be inserted into a new query window, a file, or the clipboard.

For stored procedures, you can also generate EXECUTE scripts or, by selecting Execute Stored Procedure from the shortcut menu, execute it interactively and generate the corresponding script via Management Studio’s Execute Procedure dialog box. This dialog box explicitly prompts you for all input parameters defined for the stored procedure.

In addition to generating scripts for your CLR entities, you can view their dependencies (either objects that are dependent on them or objects on which they depend). Just right-click the object and choose the View Dependencies option from the shortcut menu.

To remove your CLR objects, either in preparation for loading a new version of your assembly or to delete the objects permanently, you have several options. For Visual Studio SQL Server projects, redeploying your assembly causes Visual Studio to drop it and any SQL CLR objects within it that were previously deployed by Visual Studio. This means that new versions can be deployed from Visual Studio without any preparatory steps.

For Class Library projects, you must issue T-SQL DROP commands for each of your SQL CLR objects and then for the assembly itself. You must drop any dependent objects before you drop the SQL CLR entity. For example, you must drop tblPoint before dropping typPoint. You can write these DROP scripts by hand or generate them by using the Script object type As/DROP To shortcut menu options in the Management Studio Object Explorer window.

You can also use the Delete shortcut menu option on any SQL CLR object in the Management Studio Object Explorer window to drop an object. This option brings up the Delete Object dialog box (Figure 3-18).

![Figure 3-18](The Management Studio Delete Object dialog box)
The script file Cleanup.sql in the Management Studio project provided with the sample code contains all the necessary DROP commands, in the proper order, for removing all traces of our Visual Studio SQL Server project or Class Library project from the AdventureWorks database. For the SQL Server project, run this script only if you want to permanently remove these objects. For the Class Library project, run it before you deploy an updated version of your assembly or if you want to permanently remove these objects.

SQL CLR objects, with the exception of DDL triggers, can also be viewed in Visual Studio 2005’s Server Explorer window, as shown in Figure 3-19.

You’ll find most of the objects under their appropriate parent nodes within the data connection parent node: CLR stored procedures appear under the Stored Procedures node; scalar and table-valued functions, as well as aggregates, appear under the Functions node; the Types and Assemblies nodes contain their namesake objects; and DML triggers appear under the node of the table to which they belong.

You may also drill down on a particular assembly node and view a list of all its SQL CLR objects, as well as the source code files that make it up (see Figure 3-20).

For assemblies created from Visual Studio 2005 SQL Server projects, you may double-click on any SQL CLR object in the Server Explorer window to view its source code. (You may also do this by selecting the Open option from the SQL CLR object node’s shortcut menu or the Data/Open option from Visual Studio’s main menu while the node is selected.) If the assembly’s project is open when you open the object’s source, the code will be editable; if the project is not open, the source will be read-only.
Figure 3-20  The Server Explorer window, with the Assemblies node and its child nodes highlighted

Caution  Because trgUpdateContact, our SQL CLR DML trigger, was deployed via T-SQL in the PostDeployScript.sql script and not via the SqlTrigger attribute, its source cannot be browsed through the Server Explorer window connection’s Tables\Contact (Person)\trgUpdateContact node or its Assemblies\Chapter03\trgUpdateContact (Person) node. You can, however, view its source through the Assemblies\Chapter03\trgTest.cs node.

Best Practices for SQL CLR Usage

Before we close this chapter, we’d like to summarize certain best practices for the appropriate use of SQL CLR programming.

The CLR integration in SQL Server 2005 is a powerful technology. In some cases, it allows you to do things you can’t do practically in T-SQL (such as apply complex business logic in stored procedures or triggers), and in other cases it allows you to do things you can’t do at all in T-SQL (such as create your own aggregate functions).

The fact remains, however, that set-based data selection and modification is much better handled by the declarative constructs in T-SQL than in the procedural constructs of .NET and the ADO.NET object model. SQL CLR functionality should be reserved for specific situations when the power of .NET as a calculation engine is required.

In general, functions and aggregates are great uses of SQL CLR integration. UDTs, if used as “super scalars” rather than objects per se, make good use of SQL CLR integration as well.

For stored procedures and triggers, we recommend that you start with the assumption that these should be written in T-SQL and write them using SQL CLR code only if a case can be made that they cannot be reasonably written otherwise. And before you make such a case,
consider that SQL CLR functions, aggregates, and UDTs can be used from within T-SQL stored procedures and triggers.

**Summary**

In this chapter, you’ve been exposed to the “mechanics” of developing the five basic SQL CLR entities and using them from T-SQL. You’ve seen how to take advantage of SQL Server 2005/Visual Studio 2005 integration as well as how to develop SQL CLR code in conventional Class Library assemblies and deploy them using T-SQL and SQL Server Management Studio. You’ve also been exposed to most of the SQL CLR .NET code attributes and their use in SQL Server projects and standard Class Library projects. You’ve gotten a sense of how to use Management Studio and the Visual Studio 2005 Server Explorer window as management tools for your SQL CLR objects, and we’ve discussed scenarios in which using SQL CLR integration is a good choice as well as scenarios in which T-SQL is the better choice.

This first part of the book essentially covers building databases, so in this chapter we intentionally kept our focus on using SQL CLR objects from T-SQL and Management Studio. In the previous chapter, we highlighted a number of enhancements to T-SQL that you can use in your database development process. The second part of the book covers developing applications that use your databases. In Chapter 8, we’ll look at how to consume your SQL CLR objects in .NET applications by using ADO.NET, including within strongly typed `DataSet` objects, and optionally using Windows Forms and ASP.NET data binding. And in Chapter 9, we’ll show you how to perform end-to-end debugging of client-side and SQL CLR code, as well as T-SQL code, in Visual Studio 2005. The sum total of the material from Chapters 3, 8, and 9 provides a rich resource for diving into SQL CLR development.