SQL Server™ 2005: CLR Integration

Table of Contents

SQL Server™ 2005: SQL CLR Integration

Objectives

After completing this lab, you will be able to work with SQL Server Projects, work with the in-process managed provider, leverage the .NET Framework in database development, understand managed code permissions, create triggers using managed code, create user-defined types and user-defined aggregates, compare and Contrast T-SQL to managed code, and explore new features in ADO.NET.

NOTE: This lab focuses on the concepts in this module and as a result may not comply with Microsoft® security recommendations.

NOTE: The SQL Server 2005 labs are based on beta builds of the product. The intent of these labs is to provide you with a general feel for some of the planned features in the next release of SQL Server. As with all software development projects, the final version may differ from beta builds in both features and user interface. For the latest details on SQL Server 2005, please visit http://www.microsoft.com/sql/2005/.

Scenario

Prerequisites

Estimated Time to Complete This Lab

90 Minutes

Lab Setup

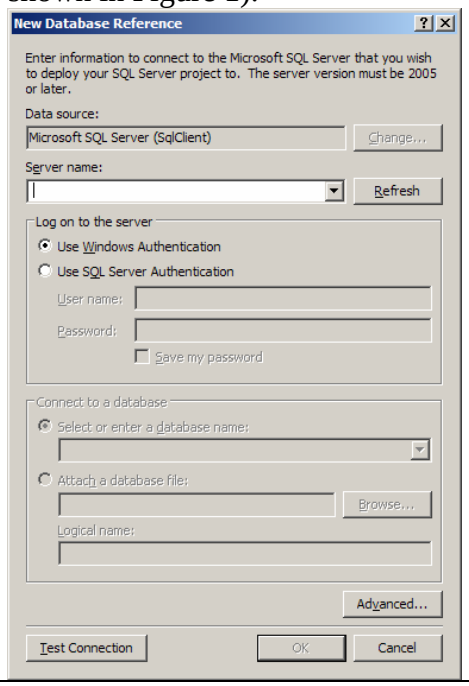
Scenario

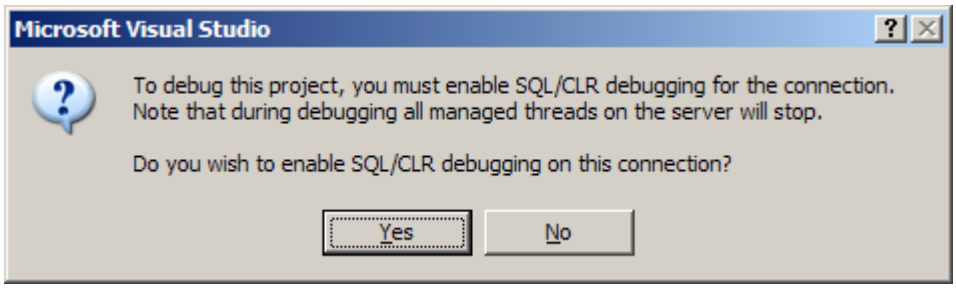
Tasks	Detailed Steps
Task 1: Log In	Log in using the **Administrator** user account. The password is **Pass@word1**.

Exercise 1
Working with SQL Server Projects

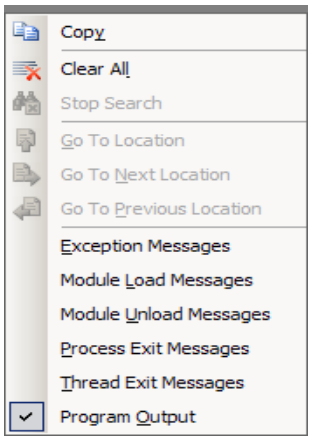
In this exercise, you will learn how to work with Microsoft Visual Studio® 2005 and Microsoft SQL Server 2005 in tandem by creating a SQL Server project and a basic user-defined function whose implementation is in managed code. Then you will build a Microsoft Windows® Form application that displays the result returned from the user-defined function. The goal of this exercise is to lay a foundation for subsequent exercises by introducing how Visual Studio 2005 and Microsoft SQL Server 2005 work together.

NOTE: Long code snippets are available in the file **C:\SQL Labs\Lab Projects\SQL CLR\SQL CLR Lab Code.txt** for your convenience.

Tasks	Detailed Steps
Task 1: Creating a SQL Server Project	1. Start Visual Studio 2005. 2. Select the **File | New | Project** menu command. Visual Studio displays the New Project dialog. 1. In the Project Types pane, indicate that you want to create a **Visual Basic** project. Expand the node for Visual Basic and select the **Database** node. 2. In the Templates pane, select **SQL Server Project**. 3. In the Name field, enter **SqlServerProject**. 4. In the Location field, enter **C:\SQL Labs\User Projects**. 5. Uncheck the **Create directory for solution** option. 6. Click **OK**. Visual Studio displays the **New Database Reference** dialog box (as shown in Figure 1). 

Tasks	Detailed Steps
	Figure 1: The New Database Reference dialog box A SQL Server project is associated with a specific database. This dialog box lets you select the database to which you will be deploying managed code. Database References are shown in the Solution Explorer and become part of the Visual Studio 2005 environment. As you open and close different projects, Database References are saved so they can be reused.
Task 2: Creating a Database Reference	1. In the server name field, enter "localhost". 2. Select **Use Windows NT Integrated security**. 3. In the database field, enter "SQLCLRIntegration". 4. Click the **Test Connection** button. 5. Click **OK** to dismiss the Add Database Reference dialog box. Upon clicking OK, a dialog box like Figure 2 will be displayed:  Figure 2: Debugging Warning 6. Click **Yes** so that debugging will be enabled.
Task 3: Using a User-Defined Function Template	1. In the Solution Explorer, right-click **SqlServerProject** and select **Add | New Item** from the context menu**.** 2. In the Add New Item dialog box, click **User-Defined Function**. 3. Click **Add** to accept the default file name. In SQL Server, a user-defined function is similar to a function in other programming languages. Unlike a stored procedure, which can return only an integer or result set, a user-defined function can return a variety of data types. You also can treat a user-defined function similarly to a table or view, issuing SELECT statements directly against the function. In prior versions of SQL Server, you could only create a user-defined function using T-SQL. With Microsoft SQL Server 2005, you can now create user-defined functions using managed code. Visual Studio includes a pre-defined template to use in your SQL Server Projects to make the process easier. 4. In the **Solution Explorer**, open the TestScripts folder, right-click **test.sql** and select **Open**. Notice that when you used the User-Defined Function template, Visual Studio automatically added an additional file to your project named **test.sql**. You can use the script in test.sql for debugging your code. You modify this script to call your code so that it can be debugged.

Tasks	Detailed Steps
	5. In the Solution Explorer, right-click Function1.vb and select **View Code**. You'll notice that an example function has already been defined. You can remove it or leave it. 6. In the Code Editor, before the End Class statement, type the following code: `<Microsoft.SqlServer.Server.SqlFunction()> _` `Public Shared Function GetTodaysDate As Date` `  Return Date.Today` `End Function` 7. In the **Solution Explorer**, right-click **test.sql** and select **Open**. 8. Replace the line **select 'put your test script here'** (located at the bottom of the file) with the following text: `select dbo.GetTodaysDate()`
Task 4: Using a User-Defined Function Template	1. In the **Solution Explorer**, right-click **SqlServerProject** and select **Deploy**. Before an assembly can be used by a database, it must first be deployed to the database server. Visual Studio 2005 makes deploying an assembly to Microsoft SQL Server 2005 easy to do. Just by right-clicking the project name in the Solution Explorer and selecting the Deploy menu item, Visual Studio 2005 automatically builds and deploys your assembly to SQL Server 2005. The assembly is also deployed and debugged simply by pressing **F5**. 2. Once the deployment process has completed, press **F5**. However, there are two problems. The first is that the code just appears to execute and return. In order to debug, place a breakpoint in the Function1.vb file where your function is defined. 3. Next, in order to get managed code to run, you need to enable the CLR for the database. From the Windows task bar, select **Start | All Programs | Microsoft SQL Server 2005 CTP | SQL Server Management Studio**. 4. When the **Connect to Server** dialog box opens, verify that **SQL Server** is selected as the **Server type,** verify that **Server name** is same name as the host machine or set it to **localhost**, and verify that **Windows Authentication** is selected as the authentication method. 5. Click **Connect**. 6. Open a new query window by via **File | New | New Database Engine Query.** When prompted, click Connect in the Connect to Database Engine dialog box. 7. In the editor, enter the following T-SQL: `sp_configure 'clr enabled', 1` 8. Press **F5** to execute the query. Then enter the following: `reconfigure;`

Tasks	Detailed Steps
	9. Highlight the new line and press **F5**. 10. Close SQL Server Management Studio (there's no need to save anything if prompted). 11. Now, back in Visual Studio, Press **F5**. 12. Once you hit your breakpoint, press **F5** to continue. 13. In the **Output** window, in the **Show output from** drop-down list, select **Debug**. You can see that the output from the user-defined function is displayed in the Output window, allowing you to fully test your managed routines without leaving Visual Studio 2005. Note that you can make it easier to see your output by right-clicking in the Output window and de-selecting all of the output options except for Program Output (see Figure 3). If you re-run your project, you see much less information displayed.  Figure 3: Controlling output window information
Task 5: Accessing a User-Defined Function from a Windows Application **NOTE:** If the Toolbox window isn't visible, select **View | Toolbox** to display it. **NOTE:** You might need to expand the Common Controls group to see the Button control on the Toolbox.	1. Select the **File | Add | New Project** menu command. 2. If prompted to save the existing project, select the C:\SQL Labs\User Projects folder. 3. In the Project Types pane, select **Visual Basic**. 4. In the Templates list, select **Windows Application**. 5. In the **Name** field, enter: **SqlServerProjectWinApp**. In the Location field, enter **C:\SQL Labs\User Projects**. 6. Click **OK** to dismiss the dialog box. 7. Right-click on **SqlServerProjectWinApp** in the Solution Explorer and select **Project | Add Reference**. Scroll the list of components until you find **System.Data**. Select it and then click OK to accept the new reference. 8. From the **Toolbox**, drag a **Button** control to the design surface of **Form1**. 9. Double-click **Button1**. 10. Scroll to the top of the code file, and enter the following Imports statement: `Imports System.Data.SqlClient` 11. In the **Code Editor**, type the following code (or paste it from the helper file)

Tasks	Detailed Steps
	within the Button1_Click event handler: `Dim connectionString As String = "Data Source=localhost;" & _` `"Initial Catalog=SQLCLRIntegration;" & _` `"Integrated Security=True"` `Using conn As New SqlConnection(connectionString)` `conn.Open()` `Dim cmd As New SqlCommand( _` `"SELECT dbo.GetTodaysDate()", conn)` `Dim result As Object = cmd.ExecuteScalar()` `Dim d As Date = CDate(result)` `MsgBox(d.ToString("d"))` `End Using` This code opens a connection to the database and executes a simple command to display today's date in a message box. Note that the code required to call a managed code user-defined function is the same as the code to call a T-SQL user-defined function. 12. In the Solution Explorer, right-click **SqlServerProjectWinApp** and select **Set as StartUp Project**. 13. Select the **Debug | Start Without Debugging** menu command (or press **Ctrl+F5**). 14. Click **Button1**. You can see that the user defined function, implemented as managed code, is accessed from the client exactly like a user-defined function written using T-SQL. 15. Click **OK**. 16. Close **Form1**. 17. Select the **File | Close Solution** menu command. This exercise provided a brief introduction to working with Visual Studio 2005 and SQL Server 2005. You also learned how to create SQL Server projects and how to write user-defined functions with managed code.

Exercise 2
Leveraging the .NET Framework in Database Development

In this exercise, you will learn how to leverage the .NET Framework when doing database development. You will learn how to create a user-defined function that uses regular expressions to validate input. In the past, this would have been difficult to accomplish with only T-SQL. Validating data passed to a procedure or function is a common database operation, and this exercise shows how easily this task can be accomplished using managed code.

Tasks	Detailed Steps
Task 1: Creating a User-Defined Function	1. Select the **File | New | Project** menu command. 2. Create a new **Microsoft Visual C#®** project, selecting the **SQL Server Project** template. 3. In the **Name** field, enter **UserDefinedFunction**. 4. In the Location field, enter **C:\SQL Labs\User Projects**. (Uncheck the **Create directory for solution** option if it's checked.) 5. Click **OK**. 6. In the **Add Database Reference** dialog box, select the existing reference to the SQLCLRIntegration database. 7. Click **OK**. 8. In the Solution Explorer, right-click **UserDefinedFunction** and select **Add | New Item** from the context menu. 9. In the **Add New Item** dialog, click **User-Defined Function**. 10. Click **Add**. 11. In the **Code Editor**, change the existing function declaration line to the following code: `public static bool IsValidZipCode(string ZipCode)` You will now create a user-defined function that validates a ZIP code. Note that the function is decorated with a [SqlFunction] attribute. This tells the compiler that the function is a SQL Server user-defined function. 12. Delete the following two lines of code: `// Put your code here` `return "Hello";` 13. Inside **IsValidZipCode**, type the following line of code (or paste it from the helper file): `return System.Text.RegularExpressions.Regex.IsMatch(` `ZipCode, @"^\s*(\d{5}|(\d{5}-\d{4}))\s*$");` Regular expression support is just one area of additional functionality that the .NET Framework provides. In fact, as a database developer, you now will have access to literally thousands of pre-built classes that you can reuse in your database objects.

Tasks	Detailed Steps
	14. Select the **File | Save All** menu item to save the project. Save the project in the C:\SQL Labs\User Projects folder.
Task 2: Calling a User-Defined Function from a Windows Application	1. Select the **File | Add | New Project** menu command. 2. Ensure that **Visual C#** is still selected in the Project Types pane. 3. In the **Templates** list, select **Windows Application**. You will now create a Windows Forms application that tests the IsValidZipCode user-defined function. 4. In the **Name** field, enter **UserDefinedFunctionWinApp**. 5. In the **Location** field, enter **C:\SQL Labs\User Projects**. 6. Click **OK**. 7. From the **Toolbox**, drag a **TextBox** control to the design surface of **Form1**. The user will enter ZIP codes into the TextBox control. 8. From the **Toolbox**, drag a **Button** control to the design surface of **Form1**. 9. Double-click **button1**. 10. Scroll to the top of the code file, and under the existing using statements, add the following: `using System.Data.SqlClient;` 11. In the **Code Editor**, type the following code within the button1_Click procedure: `string connectionString = "Data Source=localhost;" + "Initial Catalog=SQLCLRIntegration;" + "Integrated Security=True";` `using (SqlConnection conn = new SqlConnection(connectionString)) { conn.Open(); SqlCommand cmd = new SqlCommand("SELECT dbo.IsValidZipCode(@ZipCode)", conn); cmd.CommandType = CommandType.Text; cmd.Parameters.Add("@ZipCode", SqlDbType.VarChar); cmd.Parameters["@ZipCode"].Value = textBox1.Text; Object result = cmd.ExecuteScalar(); MessageBox.Show(result.ToString()); }` This code opens a connection to the database, and executes a simple command to call the user-defined function, passing in a value submitted by the user. This value will then be compared to a regular expression to insure that it is a properly formatted ZIP code. 12. In the Solution Explorer, right-click **UserDefinedFunctionWinApp** and

Tasks	Detailed Steps
	select **Set as StartUp Project.** 13. Select the **File | Save All** menu command. 14. Select the **Debug | Start Without Debugging** menu command (or press **Ctrl+F5**). 15. In the text box enter **"**555-5555**"**. To test IsValidZipCode, first enter data that is invalid. 16. Click **Button1**. The application calls the user-defined function and displays a message box indicating that the data is not a valid ZIP code. 17. Click **OK**. 18. In the text box, enter **"**98052**"**. 19. Click **button1**. Because "98052" is a valid ZIP code, the code displays True. 20. Click **OK**. 21. Close **Form1**. 22. Select the **File | Close Solution** menu command. In this exercise, you learned how to create a user-defined function in managed code using Visual Studio 2005. By leveraging the .NET Framework Base Class Library, you saw how you can use managed code to create powerful database objects.

Exercise 3
Understanding Managed Code Permissions

In this exercise, you will learn how various permission sets can be applied to assemblies running in SQL Server 2005. You will create a stored procedure that uses the .NET Framework to write to an external file. By default, managed code within SQL Server 2005 is constrained so that it cannot access external resources. Therefore, appropriate permission must be granted to the assembly in order for it to write to the file. You will also learn how to load and unload assemblies manually using DDL statements within SQL Server Management Studio.

Tasks	Detailed Steps
Task 1: Creating a Stored Procedure that Writes to a File NOTE: Be careful to not delete the SqlProcedure attribute preceding the declaration in Step 11.	1. Select the **File | New | Project** menu command. 2. Using Microsoft Visual Basic®, create a new SQL Server project. 3. In the **Name** field, enter **Permissions**. 4. In the **Location** field, enter **C:\SQL Labs\User Projects**. 5. Click **OK**. 6. In the **Add Database Reference** dialog box, select the existing reference to the SQLCLRIntegration database**.** 7. Click **OK**. 8. In the Solution Explorer, right-click **Permissions** and select **Add | New Item** from the context menu**.** 9. In the **Add New Item** dialog box, select **Stored Procedure**. 10. Click **Add**. 11. In the **Code Editor**, change the existing subroutine declaration line to the following: `Public Shared Sub WriteToFile( _` `  ByVal fileName As String, ByVal message As String)` You will now create the body of the stored procedure that writes to the file. The procedure accepts two parameters: the file to write to, and the text to write. 12. Inside the **WriteToFile** subroutine, type the following lines of code: `Using sw As New System.IO.StreamWriter(fileName, True)` `  sw.WriteLine(message)` `End Using` This routine leverages the inherent functionality present in the .NET Framework Base Class Library to write to a file. 13. In the **Solution Explorer**, right-click **Permissions** and select **Build**. 14. Select **the File | Save All** menu items, and save the project.
Task 2: Loading an Assembly Manually	1. From the Windows task bar, select **Start | All Programs | Microsoft SQL Server 2005 CTP | SQL Server Management Studio**. 2. When the **Connect to Server** dialog box opens, verify that **Database Engine**

Tasks	Detailed Steps
using DDL Statements **NOTE:** In Step 6, make sure that there are no line breaks in the folder name when typing the code. If you wish you can copy & paste the above code from the helper file.	is selected as the **Server type**, verify that **Server name** is same name as the host machine (or localhost), and verify that **Windows Authentication** is selected as the authentication method. 3. Click **Connect**. 4. Open a new query window by via **File | New | New Database Engine Query** (follow the same procedures as above to connect to the server) and enter the following T-SQL: `USE SQLCLRIntegration` `GO` 5. Press **F5**. 6. Type the following lines of code into the query window: `CREATE ASSEMBLY Permissions` `FROM 'C:\SQL Labs\User Projects\Permissions\bin\Permissions.dll'` 7. Select the above lines of code and press **F5**. Running this DDL statement loads the Permissions.dll assembly into Microsoft SQL Server 2005. Until now, you have been using Visual Studio 2005 to deploy all assemblies. This exercise demonstrates how to load an assembly manually using DDL for two reasons. First, loading the assembly automatically doesn't allow you to configure the permissions for an assembly, and the ability to configure a permission set will be needed later in the exercise. Second, this gives you a great opportunity to see what is going on under the hood. 8. Type the following lines of code (or paste from the helper file): `CREATE PROCEDURE WriteToFile` `@FILENAME NVARCHAR(256),` `@MESSAGE NVARCHAR(4000)` `AS` `EXTERNAL NAME Permissions.[Permissions.StoredProcedures].` `WriteToFile` 9. Select the above lines of code and press **F5**. Running this DDL statement creates an entry point for the managed stored procedure in the assembly you just loaded. 10. Type the following line of code: `EXEC WriteToFile 'c:\test.txt', 'This is a test'` 11. Select the above line of code and press **F5**. Running this T-SQL causes an error indicating that you don't have permission to write to the file you specified. The error text will look something like this (the actual text may be slightly different): `Msg 6522, Level 16, State 1, Procedure WriteToFile, Line 0`

Tasks	Detailed Steps
	A .NET Framework error occurred during execution of user defined routine or aggregate 'WriteToFile': System.Security.SecurityException: Request for the permission of type 'System.Security.Permissions.FileIOPermission, mscorlib, Version=2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934e089' failed. System.Security.SecurityException: at .... (Deleted for brevity.) This error is expected. To ensure the security of the server, SQL Server 2005 only grants a limited set of permissions to an assembly. The file system is a secured resource and by default, a stored procedure is not allowed to access secured resources. If an assembly needs additional permissions, it must be specifically granted.
Task 5: Unloading an Assembly and Stored Procedure Manually Using DDL Statements	1. Type the following two lines of code: `DROP PROCEDURE WriteToFile` `DROP ASSEMBLY Permissions` 2. Select the above lines of code and press **F5**. This unloads the stored procedure and assembly.
Task 6: Loading an Assembly with a Specific Permission Set **NOTE:** In Step 1, make sure that there are no line breaks in the folder name when typing the code.	1. Type the following lines of code: `CREATE ASSEMBLY Permissions` `FROM 'C:\SQL Labs\User Projects\Permissions\bin\Permissions.dll'` `WITH PERMISSION_SET = EXTERNAL_ACCESS` An important distinction between this CREATE ASSEMBLY statement and the previous one is that this time you're specifying the permission set to be used by the assembly. There are three different permission sets: SAFE, EXTERNAL_ACCESS, and UNSAFE. SAFE is the default permission set and works for the majority of scenarios. When code in an assembly runs under the SAFE permission set, it can only do computation and data access within the server via the in-process managed provider. EXTERNAL_ACCESS is a code permission set that addresses scenarios where the code needs to access resources outside the server, such as the files, network, registry and environment variables. UNSAFE code permission is for those situations where an assembly is not verifiably safe or requires additional access to restricted resources, such as the Microsoft Win32® API. 2. Highlight the above lines of code so that they are selected and press **F5**. Running this DDL statement loads the Permissions.dll assembly into SQL Server 2005 with the EXTERNAL_ACCESS permission set.

Tasks	Detailed Steps
	3. Type the following lines of code again (or highlight the ones you had typed previously): `CREATE PROCEDURE WriteToFile` `@FILENAME NVARCHAR(256),` `@MESSAGE NVARCHAR(4000)` `AS` `EXTERNAL NAME Permissions.[Permissions.StoredProcedures].` `WriteToFile` 4. Select the above lines of code and press **F5**. 5. Type the following line of code: `EXEC WriteToFile 'c:\test.txt', 'This is a test'` 6. Highlight the above line of code so that it is selected and press **F5**. This time, the command completes successfully. 7. In **SQL Server Management Studio**, select the **File | Exit** menu command. If prompted to save your changes, click **No**.
Task 7: Viewing the File	1. Open **Windows Explorer** and find the file you just created. 2. Double-click the file to load it into NotePad. You can see that the message was written from the managed code into this file. 3. Close **Notepad**.

In this exercise, you began working with SQL Server Management Studio, part of SQL Server 2005. You learned that .NET assemblies are loaded into SQL Server 2005 using a given permission set. By default, assemblies are loaded with limited privileges and cannot access external resources such files. To give an assembly access to external resources, it must be loaded with a different additional permission set such as EXTERNAL_ACCESS. You also learned how to manually load and unload assemblies and stored procedures using DDL.

Exercise 4
Creating User-Defined Types and User-Defined Aggregates

In this exercise, you will learn how to extend the type system by creating user-defined types as well as how you can extend the list of installed aggregate functions by creating new user-defined aggregates whose implementation is in managed code.

Tasks	Detailed Steps
Task 1: Using the User-Defined Type Template	1. In Visual Studio 2005, select the **File | New | Project** menu command. 2. Create a new **Visual Basic** project, using the **SQL Server Project** template. 3. In the **Name** field, enter **UserDefinedTypes**. 4. In the **Location** field, enter **C:\SQL Labs\User Projects**. 5. Click **OK**. 6. In the **Add Database Reference** dialog box, select the existing reference to the SQLCLRIntegration database**.** 7. Click **OK**. 8. In the **Solution Explorer**, right-click **UserDefinedTypes** and select **Add | New Item.** 9. In the **Add New Item** dialog box, click **User-Defined Type**. 10. In the **Name** field, enter **Seat**. 11. Click **Add**. 12. In the Code Editor, navigate to the structure definition. `<Serializable()> _` `<Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)> _` `Public Structure Seat` `  Implements INullable` The first attribute is the Serializable attribute, which allows the class to be serialized. The second attribute, SqlUserDefinedType, instructs that the structure will be serialized using the server's native format (in contrast to a user-defined format). Finally, the structure implements INullable. 13. In the **Code Editor**, modify the SqlUserDefinedType attribute so that it uses a user-defined format with a maximum byte size of 512. When you're done, your code should look like the following code: `<Serializable()> _` `<SqlUserDefinedType(Format.UserDefined, MaxByteSize:=512)> _` `Public Structure Seat` `  Implements INullable` This needs to be done because the structure you're defining will have string fields that are not serializable using the native format. 14. In the **Code Editor**, after the **Public Structure Seat** statement, scroll through the following code:

Tasks	Detailed Steps

```
<Serializable()> _
<SqlUserDefinedType(Format.UserDefined, MaxByteSize:=512)> _
Public Structure Seat
    Implements INullable

    Public Overrides Function ToString() As String
        ' Put your code here
        Return ""
    End Function

    Public ReadOnly Property IsNull() As Boolean _
        Implements INullable.IsNull
        Get
            ' Put your code here
            Return m_Null
        End Get
    End Property

    Public Shared ReadOnly Property Null() As Seat
        Get
            Dim h As Seat = New Seat
            h.m_Null = True
            Return h
        End Get
    End Property

    Public Shared Function Parse(ByVal s As SqlString) As Seat
        If s.IsNull Then
            Return Null
        End If

        Dim u As Seat = New Seat
        ' Put your code here
        Return u
    End Function

    ' This is a place-holder method
    Public Function Method1() As String
        ' Put your code here
        Return "Hello"
    End Function

    ' This is a place-holder static method
    Public Shared Function Method2() As SqlString
        ' Put your code here
        Return New SqlString("Hello")
    End Function

    ' This is a place-holder field member
    Public var1 As Integer
    ' Private member
    Private m_Null As Boolean
End Structure
```

Tasks	Detailed Steps
	This is boilerplate code supplied for user-defined types, and some of it will need to be changed. First, a User-Defined Type (UDT) must implement the INullable interface. This interface contains a single property that you must implement, named **IsNull**. In addition to implementing the INullable interface, the ToString, Parse, and Null members must be supplied. These routines allow the user of the UDT to query some basic information about the UDT, obtain a string representation of the current instance, and support a conversion operation using the Convert operator via the Parse function. 15. In the Seat structure, in the (Declarations) section, before the ToString method, type the following code: `Private m_size As String` `Private m_type As String` The remainder of the code for a UDT is implementation-specific. The first line declares a field that indicates whether the type is null. The second and third lines declare variables to hold the values of the structure's custom properties. 16. In the **Code Editor**, in the **IsNull** routine, delete the following code: `' Put your code here` `Return True` These lines of code will be replaced with implementation-specific code. 17. In the **Code Editor**, in the **IsNull** routine, before the **End Get** statement, type the following code: `Return m_Null` This statement returns to the caller whether or not the type is null. Without this line, your user-defined type will not work. In the **Code Editor**, before the **End Structure** statement, type the following code (or copy & paste from the helper file): `Public Property Size() As String` `    Get` `        Return m_size` `    End Get` `    Set(ByVal Value As String)` `        m_size = Value` `    End Set` `End Property` `Public Property Type() As String` `    Get` `        Return m_type` `    End Get` `    Set(ByVal Value As String)`

Tasks	Detailed Steps

```
        If Value = "Racing" Or Value = "Cruising" Or _
           Value = "Banana" Then
            m_type = Value
        Else
            Throw New ArgumentException("Type must be " & _
               "either Racing, Cruising, or Banana")
        End If
    End Set
End Property
```

The Size and Type properties allow the user to specify the size and type of the seat. The type of seat is constrained to one of three possible values. Just above the code you entered, remove the sample methods **Method1** and **Method2** as well as the **var1** field (listed below for reference).

```
    ' This is a place-holder method
    Public Function Method1() As String
        ' Put your code here
        Return "Hello"
    End Function

    ' This is a place-holder static method
    Public Shared Function Method2() As SqlString
        ' Put your code here
        Return New SqlString("Hello")
    End Function

    ' This is a place-holder field member
    Public var1 As Integer
```

18. At this point, if you build and then try and deploy the assembly, you will get a deployment failure. In the Visual Studio 2005 Output window, the following error will be displayed:

```
Error: Type "UserDefinedTypes.UserDefinedTypes.Seat" is marked
for user-defined serialization, but does not implement the
"System.Data.Microsoft.SqlServer.Server.IBinarySerialize"
interface.   UserDefinedTypes
```

19. In order to eliminate this error, you need to implement a second interface, IBinarySerialize. At the top of the structure definition, add the following text after **Implements INullable** and press Enter:

```
, IBinarySerialize
```

The code editor automatically stubs out entry points for the Interface (scroll to the bottom of the file) as follows:

```
Public Sub Read(ByVal r As System.IO.BinaryReader) _
  Implements System.Data.Sql.IBinarySerialize.Read
```

Tasks	Detailed Steps

```
End Sub

Public Sub Write(ByVal w As System.IO.BinaryWriter) _
  Implements System.Data.Sql.IBinarySerialize.Write

End Sub
```

20. Add the following code to the Read method.

```
Me.m_size = r.ReadString()
Me.m_type = r.ReadString()
Me.m_Null = r.ReadBoolean()
```

This bit of code reloads the serialized values into the UDT's fields from the passed in binary stream reader.

21. Add the following code to the Write method.

```
If m_size Is Nothing Then m_size = String.Empty
If m_type Is Nothing Then m_type = String.Empty

w.Write(m_size)
w.Write(m_type)
w.Write(m_Null)
```

This code checks both of the string fields to see if they have not been initialized. If they haven't, they're initialized to empty. Then all three fields are written out using the passed-in binary stream writer.

Your completed code for the Seat structure should look very similar to the following code listing:

```
<Serializable()> _
<SqlUserDefinedType(Format.UserDefined, MaxByteSize:=512)> _
Public Structure Seat
    Implements INullable, IBinarySerialize

    Private m_size As String
    Private m_type As String

    Public Overrides Function ToString() As String
        ' Put your code here
        Return ""
    End Function

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

    Public Shared ReadOnly Property Null() As Seat
        Get
```

Tasks	Detailed Steps

```
            Dim h As Seat = New Seat
            h.m_Null = True
            Return h
        End Get
    End Property

    Public Shared Function Parse(ByVal s As SqlString) As Seat
        If s.IsNull Then
            Return Null
        End If

        Dim u As Seat = New Seat
        ' Put your code here
        Return u
    End Function

    ' Private member
    Private m_Null As Boolean

' Continued on the following page

    Public Property Size() As String
        Get
            Return m_size
        End Get
        Set(ByVal Value As String)
            m_size = Value
        End Set
    End Property

    Public Property Type() As String
        Get
            Return m_type
        End Get
        Set(ByVal Value As String)
            If Value = "Racing" Or Value = "Cruising" Or _
               Value = "Banana" Then
                m_type = Value
            Else
              Throw New ArgumentException("Type must be " & _
                   "either Racing, Cruising, or Banana")
            End If
        End Set
    End Property

   Public Sub Read(ByVal r As System.IO.BinaryReader) _
   Implements Microsoft.SqlServer.Server.IBinarySerialize.Read
        Me.m_size = r.ReadString()
        Me.m_type = r.ReadString()
        Me.m_Null = r.ReadBoolean()
    End Sub

   Public Sub Write(ByVal w As System.IO.BinaryWriter) _
  Implements Microsoft.SqlServer.Server.IBinarySerialize.Write
```

Tasks	Detailed Steps
	`` If m_size Is Nothing Then m_size = String.Empty If m_type Is Nothing Then m_type = String.Empty w.Write(m_size) w.Write(m_type) w.Write(m_Null) End Sub End Structure ``` 22. In the **Solution Explorer**, right-click **UserDefinedTypes** and select **Deploy.** This step deploys your user-defined type to SQL Server.
Task 2: Testing a User-Defined Type	1. If it's not already running, start **SQL Server Management Studio,** and connect if requested**.** 2. Create a new Database Engine query as in the previous example. 3. In the **Code Editor**, type the following line of code: ``` USE SQLCLRIntegration ``` 4. Press **F5**. 5. In **Registered Servers**, select **localhost** (or the name of your computer)**.** 6. In **Object Explorer**, expand **localhost | Databases | SQLCLRIntegration | Programmability | Types | User-Defined Types | dbo.Seat.** As you can see, the user-defined type is now part of the database. 7. In the **query editor window**, type the following lines of code: ``` CREATE TABLE BikeSeats (SeatID INT NOT NULL, SeatInfo [Seat] NOT NULL) ``` This shows you how to create a table using your user-defined type that was written in managed code. Notice that the CREATE TABLE statement above uses the Seat UDT for the data type of the SeatInfo column. 8. Select the above lines of code and press **F5**. 9. In the **query editor window**, type the following lines of code: ``` DECLARE @s [Seat] SET @s = CONVERT([Seat], '') SET @s.Size = '10' SET @s.Type = 'Racing' INSERT BikeSeats VALUES(1, @s) ``` This creates an instance of the Seat data type and sets its Size and Type

Tasks	Detailed Steps
	properties. Then it uses an INSERT statement to add a new record to the BikeSeats table. This shows you how to create a table using your user-defined type that was written in managed code. 10. Select the above lines of code and press **F5**. 11. In the **Code Editor**, type the following lines of code: `SELECT SeatInfo.Size AS Size, SeatInfo.Type AS Type` `FROM BikeSeats` This SELECT query retrieves the record that was just inserted into the BikeSeats table. Select the above lines of code and press **F5**. You should see the racing bike seat of size 10 in the output window. Keep SQL Server Management Studio open; you will use a bit later.
Task 3: Testing the User-Defined Aggregate Template	An aggregate function performs a calculation on a set of values and returns a single value. T-SQL already comes with several built-in aggregate functions such as COUNT, AVG and SUM. With CLR integration, you are now able to create your own aggregates using managed code. For this exercise, you will create a user-defined aggregate that performs string concatenation. 1. Switch back to Visual Studio 2005. 2. In the **Solution Explorer**, right-click **UserDefinedTypes** and select **Add | New Item.** 3. In the **Add New Item** dialog box, click **Aggregate**. 4. Change the name to "StringAggregate" and then click **Add**. 5. In the Code Editor, navigate to the structure definition. `<Serializable()> _` `<Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Native)> _` `Public Structure StringAggregate` The first attribute is the Serializable attribute, which allows the class to be serialized. The second attribute, SqlUserDefinedAggregate, allows the structure to be serialized using the server's native format (in contrast to a user-defined format). 6. In the **Code Editor**, modify the SqlUserDefinedAggregate attribute so that it uses a user-defined format with a maximum byte size of 8000. When you're done, your code should look like the following: `<Serializable()> _` `<SqlUserDefinedAggregate(Format.UserDefined, _` `MaxByteSize:=8000)> _` `Public Structure StringAggregate` This needs to be done because the structure you're defining has a field that is not

Tasks	Detailed Steps
	serializable using the native format. 7. In the **Code Editor**, at the top of the code window, type the following code: `Imports System.Text` The aggregate you are creating will use the StringBuilder class of the System.Text namespace. Importing System.Text reduces typing. 8. In the **StringAggregate** structure, before **Public Sub Init()**, type the following code: `Private sb As StringBuilder` `Private firstConcat As Boolean` `Private m_isNull As Boolean` The StringBuilder object sb will be used to perform fast string concatenations. The Boolean flag firstConcat indicates whether a concatenation is the first concatenation. 9. In the **Init** procedure, before **End Sub**, type the following code: `sb = New StringBuilder` `firstConcat = True` This initializes the StringBuilder object when the aggregate is invoked and sets the firstConcat field to true. 10. In the **Accumulate** procedure, before **End Sub**, type the following code: `If firstConcat Then` `  firstConcat = False` `Else` `  sb.Append(", ")` `End If` `sb.Append(value)` These lines of code perform the actual string concatenation. Accumulate is called for each row in the result set from which values will be aggregated. 11. In the **Merge** procedure, before **End Sub**, type the following code: `Accumulate(value.ToString())` The Merge method is used by the query processor to merge another instance of this aggregate class with another instance as part of partial computations. 12. In the **Terminate** procedure, delete the following lines of code: `' Put your code here` `Return New SqlString("")`

Tasks	Detailed Steps

13. In the **Terminate()** routine, before **End Sub**, type the following code:

```
Return sb.ToString()
```

This line of code converts the StringBuilder object to a regular String object and returns that String back to the calling code. Terminate is invoked when the entire result set has been processed.

14. In the **StringAggregate** class, before **End Structure**, type the following code:

```
Public Overrides Function ToString() As String
  Return sb.ToString()
End Function
```

Again, this code converts the StringBuilder object to a regular String object and returns that String back to the calling code.
Just above the code you entered, remove the sample **var1** field and comment (listed below for reference).

```
' This is a place-holder field member
Public var1 As Integer
```

15. As with the previously built user-defined type, this aggregate needs to implement INullable. After the line of code that defines the aggregate, type the following code and press Enter.

```
Implements INullable
```

The code editor automatically stubs out entry points for the Interface (scroll to the bottom of the file) as follows:

```
Public ReadOnly Property IsNull() As Boolean _
  Implements System.Data.SqlTypes.INullable.IsNull
  Get

  End Get
End Property
```

16. Add the following line of code to return the current null state of the type using the previously defined field:

```
Return m_isNull
```

17. The aggregate needs two additional shared methods to comply with the rules of aggregate types. You will add a read-only Null property and a Parse method (similar to those used earlier in the Seat type). Type the following code or copy it from the lab's helper file:

Tasks	Detailed Steps

```
Public Shared ReadOnly Property Null() As StringAggregate
  Get
    Dim a As New StringAggregate
    Return a
  End Get
End Property

Public Shared Function Parse(ByVal s As SqlString) _
  As StringAggregate

  If s.IsNull Or s.Value.ToLower().Equals("null") Then
    Return Null
  End If

  Dim a As New StringAggregate
  Return a
End Function
```

18. As with the previously built user-defined type, this aggregate needs to implement IBinarySerialize.

```
, IBinarySerialize
```

The code editor automatically stubs out entry points for the Interface (scroll to the bottom of the file) as follows (line continuation characters have been added for formatting purposes):

```
Public Sub Read(ByVal r As System.IO.BinaryReader) _
  Implements Microsoft.SqlServer.Server.IBinarySerialize.Read

End Sub

Public Sub Write(ByVal w As System.IO.BinaryWriter) _
  Implements Microsoft.SqlServer.Server.IBinarySerialize.Write

End Sub
```

19. Add the following code to the Read method.

```
sb = New StringBuilder
Me.firstConcat = r.ReadBoolean()
Me.m_isNull = r.ReadBoolean()
If Not Me.firstConcat Then
   Dim sTemp As String = r.ReadString()
   sb.Append(sTemp)
End If
```

This bit of code reloads the serialized values into the UDT's fields and creates a new instance of the StringBuilder, reloading it with the previously stored state (if any).

Tasks	Detailed Steps
	20. Add the following code to the Write method. `w.Write(Me.firstConcat)` `w.Write(Me.m_isNull)` `If sb.Length > 0 Then` `  w.Write(sb.ToString())` `End If` `sb = Nothing` All of the aggregate's fields are written out. Because it's the StringBuilder's data that is interesting, not the StringBuilder instance itself, its contents are serialized (if any exists). All of this is written out using the passed-in binary stream writer. The completed StringAggregate structure should now look very similar to the following code: `<Serializable()> _` `<SqlUserDefinedAggregate(Format.UserDefined, _` `MaxByteSize:=8000)> _` `Public Structure StringAggregate` `  Implements INullable, IBinarySerialize` `  Private sb As StringBuilder` `  Private firstConcat As Boolean` `  Private m_isNull As Boolean` `  Public Sub Init()` `    ' Put your code here` `    sb = New StringBuilder` `    firstConcat = True` `  End Sub` `  Public Sub Accumulate(ByVal value As SqlString)` `    ' Put your code here` `    If firstConcat Then` `      firstConcat = False` `    Else` `      sb.Append(", ")` `    End If` `    sb.Append(value)` `  End Sub` `  Public Sub Merge(ByVal value As StringAggregate)` `    ' Put your code here` `    Accumulate(value.ToString())` `  End Sub` `  Public Function Terminate() As SqlString` `    Return sb.ToString()` `  End Function`

Tasks	Detailed Steps

```
  Public Overrides Function ToString() As String
    Return sb.ToString()
  End Function

  Public ReadOnly Property IsNull() As Boolean _
     Implements System.Data.SqlTypes.INullable.IsNull
    Get
      Return m_isNull
    End Get
  End Property

  Public Shared ReadOnly Property Null() As StringAggregate
    Get
      Dim a As New StringAggregate
      Return a
    End Get
  End Property

  Public Shared Function Parse(ByVal s As SqlString) _
    As StringAggregate

    If s.IsNull Or s.Value.ToLower().Equals("null") Then
      Return Null
    End If

    Dim a As New StringAggregate
    Return a
  End Function

Public Sub Read(ByVal r As System.IO.BinaryReader) _
   Implements Microsoft.SqlServer.Server.IBinarySerialize.Read
    sb = New StringBuilder
    Me.firstConcat = r.ReadBoolean()
    Me.m_isNull = r.ReadBoolean()
    If Not Me.firstConcat Then
      Dim sTemp As String = r.ReadString()
      sb.Append(sTemp)
    End If
End Sub

Public Sub Write(ByVal w As System.IO.BinaryWriter) _
  Implements Microsoft.SqlServer.Server.IBinarySerialize.Write
    w.Write(Me.firstConcat)
    w.Write(Me.m_isNull)
    If sb.Length > 0 Then
      w.Write(sb.ToString())
    End If
    sb = Nothing
End Sub

End Structure
```

21. In the Solution Explorer, right-click UserDefinedTypes and select Rebuild.

Tasks	Detailed Steps
Task 4: Deploying a User-Defined Function	1. Switch to **SQL Server Management Studio**. Before proceeding, you drop the user-defined type and UserDefinedTypes assembly. 2. Type the following line of code: `DROP TABLE BikeSeats` 3. Select the above line of code, and press **F5**. 4. Type the following line of code: `DROP TYPE Seat` 5. Select the above line of code and press **F5**. 6. Switch to Visual Studio 2005. 7. In the **Solution Explorer**, right-click **UserDefinedTypes**, and select **Deploy**.
Task 5: Testing the User-Defined Aggregate	1. In SQL Server Management Studio, in the existing query, type the following lines of code: `CREATE TABLE BookAuthors` `(` `BookID int NOT NULL,` `AuthorName nvarchar(200) NOT NULL` `)` 2. Select the above lines of code and press **F5**. 3. Type the following lines of code: `INSERT BookAuthors VALUES(1, 'Johnson')` `INSERT BookAuthors VALUES(2, 'Taylor')` `INSERT BookAuthors VALUES(3, 'Steven')` `INSERT BookAuthors VALUES(2, 'Mayler')` `INSERT BookAuthors VALUES(3, 'Roberts')` `INSERT BookAuthors VALUES(3, 'Michaels')` 4. Select the above lines of code and press **F5**. This inserts some sample data into the newly created BookAuthors table. Next, you will test the user-defined aggregate in a SELECT query. 5. Type the following lines of code: `SELECT BookID, dbo.StringAggregate(AuthorName) As Authors` `FROM BookAuthors` `GROUP BY BookID` 6. Select the above lines of code and press **F5**.

Tasks	Detailed Steps
	The results should be: `1 Johnson` `2 Taylor, Mayler` `3 Roberts, Michaels, Steven` 7. Select the **File | Exit** menu command. If prompted to save changes, click **No**. 8. In **Visual Studio 2005**, select the **File | Close Solution** menu command. This exercise demonstrated extending the type system by creating a user-defined type and how to create a user-defined aggregate using managed code.

Exercise 5
Comparing and Contrasting T-SQL and Managed Code

In this exercise, you will directly compare and contrast T-SQL and managed code. You will see how managed code can in some situations result in less code, better performance, and more power. As an example, you will write the same routine twice, the first time in traditional T-SQL and the second in managed code.

Tasks	Detailed Steps
Task 1: Creating the User-Defined function Using T-SQL	1. Start **SQL Server Management Studio,** if it's not already running. 2. Connect to the server as before. 3. Using the **New Query** button on the toolbar, create a new Database Engine Query. Connect, if requested. 4. In the **Code Editor**, type the following line of code: `USE SQLCLRIntegration` 5. Press **F5**. 6. Type the following lines of code: `CREATE FUNCTION Factorial1 (@Number FLOAT)` `RETURNS FLOAT` `AS` `BEGIN` `  DECLARE @returnValue FLOAT` `  IF @Number <= 1` `  SELECT @returnValue = 1` `  ELSE` `  SELECT @returnValue = @Number * dbo.Factorial1(@Number - 1)` `  RETURN @returnValue` `END` `GO` This user-defined type is written in traditional T-SQL to implement a recursive factorial algorithm. A factorial is the product of all the positive integers multiplied together, starting with a given number and working down to 2 in whole number increments. For example, the factorial of 3 is 6 (3 * 2 * 1). The factorial of 4 is 24 (4 * 3 * 2 * 1). Recursive algorithms have many uses, such as iterating through collections of collections, sorting, or applying mathematical formulas such as greatest common divisors, Fibonacci series, Hilbert curves, and Sierpinksi curves. Factorial was chosen because it is one of the simpler recursive algorithms. 7. Select the above lines of code and press **F5**. Running this DDL statement creates the Factorial1 user-defined

Tasks	Detailed Steps
	function. 8. Type the following line of code: `SELECT dbo.Factorial1(4)` 9. Select the above line of code and press **F5**. The return value should be 24.
Task 2: Creating the User-Defined Function Using Managed Code	1. Switch to Visual Studio 2005. 2. Select the **File | New | Project** menu command. 3. Create new Visual Basic project, using the **SQL Server Project** template. 4. In the **Name** field, enter "TSQLvsCLR". 5. In the Location field, enter "C:\SQL Labs\User Projects" 6. Click **OK**. 7. In the **Add Database Reference** dialog box, select the existing reference to the SQLCLRIntegration database**.** 8. Click **OK**. **9.** In the **Solution Explorer**, right-click **TSQLvsCLR** and select **Add | New Item.** 10. In the **Add New Item** dialog box, click **User-Defined Function**. 11. Type "Factorial2" in the name field and click **Add**. 12. In the code editor, delete the entire Factorial2 sample function that's supplied as part of the template. Make sure you delete the SqlFunction attribute, as well. 13. In the code editor, just before **End Class**, type the following lines of code (or cut and paste from the helper file): `<SqlFunction()> _` `Public Shared Function Factorial2(ByVal Number As Integer) _` `  As Double` `    If Number <= 1 Then` `        Return 1` `    Else` `        Return Number * Factorial2(Number - 1)` `    End If` `End Function` One of the advantages of managed code over T-SQL is that managed code can often perform the same tasks with fewer lines of code, as you can see in this case. In addition, as this is compiled code, it may execute faster. In the **Solution Explorer**, right-click **TSQLvsCLR** and select **Deploy**.
Task 3: Pushing T-SQL to its Limits	1. Switch to SQL Server Management Studio. 2. Type the following line of code (or find your previous typed version):

Tasks	Detailed Steps

```
SELECT dbo.Factorial2(4)
```

3. Select the above line of code and press **F5**.

The return value should be 24.

4. Type the following line of code:

```
SELECT dbo.Factorial1(33)
```

5. Select the above line of code and press **F5**.

You will get the following error:

```
Msg 217, Level 16, State 1, Line 10
Maximum stored procedure, function, trigger, or view nesting
level exceeded
(limit 32).
```

This error is expected. User-defined functions and stored procedures written in T-SQL have a limitation of 32 levels of nesting. Because of this limitation, it is often not practical to write recursive routines in T-SQL.

6. Type the following line of code:

```
SELECT dbo.Factorial2(33)
```

7. Select the above lines of code, and press **F5**.

Notice that this time, there is no error. This is because user-defined functions and stored procedures written in managed code have no limit to the levels of nesting (aside from memory constraints).

8. Select the **File | Exit** menu command. If prompted to save your changes, click **No**.
9. In Visual Studio 2005, select the **File | Save All** (save your project to the C:\SQL Labs\User Projects folder) and the **File | Close Project** menu commands.

In this exercise, you were able to compare and contrast how the same routine can be written in T-SQL and managed code. By implementing a recursive factorial algorithm, you learned that using managed code can allow you to write more powerful routines with fewer lines of code that delivers better performance.

Exercise 6
Exploring New Features in ADO.NET

In this exercise, you will explore some of the new functionality in ADO.NET. First, you will see how multiple active result sets (MARS) allow you to issue multiple commands simultaneously through the same SQL connection. Then you will see how ADO.NET now supports truely asynchronous operations.

Tasks	Detailed Steps
Task 1: Examining the Starting MARS Project	1. Load Visual Studio 2005, if it's not already running. 2. In Visual Studio 2005, select the **File | Open | Project/Solution** menu command. 3. Navigate to **C:\SQL Labs\Lab Projects\SQL CLR Lab\Mars**. 4. Open the Mars.sln solution in Visual Studio 2005. 5. In the **Solution Explorer**, double-click **Program.cs**. 6. Examine the following code: `SqlCommand cmdCustomers = new SqlCommand(` `  "SELECT * FROM Sales.Customer", cnn);` `SqlDataReader drCustomers = cmdCustomers.ExecuteReader();` This code retrieves all the customers from the AdventureWorks database. Next, the application retrieves order information for all "preferred" customers. The application is able to determine if a customer is preferred by running a simple algorithm on the customer's account number. When a preferred customer is found, the total order amount for that customer is retrieved. What makes this interesting is that the order information is retrieved using the same connection, and while drCustomers is in use. Prior to MARS, you would have needed to open a second connection to the database. 7. Examine the following code: `SqlCommand cmdOrders = new SqlCommand(` `  "SELECT SUM(TotalDue) FROM Sales.SalesOrderHeader " +` `  "WHERE customerid = @customerid", cn);` `cmdOrders.Parameters.Add("@customerid", SqlDbType.Int);` This final section of code sets up the SqlCommand that will be used to retrieve order information for each preferred customer.
Task 2: Retrieving Order Information for Preferred Customers	1. Enter the following code (or copy and paste from the lab helper file) after the **TODO: Retrieve order totals for preferred customers** comment: `while (drCustomers.Read())`

Tasks	Detailed Steps
	`` { // Pick some arbitrary group of customers. In this case, // if their account number is divisible by 47, they're in. acctNum = ((string)drCustomers["AccountNumber"]); parsedAcctNum = Convert.ToInt32(acctNum.Substring(2)); if ((parsedAcctNum % 47) == 0) { cmdOrders.Parameters["@customerid"].Value = drCustomers["customerid"].ToString(); Console.WriteLine(acctNum + " " + cmdOrders.ExecuteScalar().ToString()); } } ``` This code loops through each customer in the drCustomers SqlDataReader. The account number for the customer is then retrieved from drCustomers and parsed to remove the leading two characters. An "if" statement is used to perform a simple algorithm that determines if the current row represents a preferred customer or not. If so, the customer ID is specified as a parameter for a query which retrieves total order information. This command is executed on the same connection used by drCustomers, and it is executed before drCustomers closes. This would not be possible without MARS. Once the information is retrieved, it is output to the console.
Task 3: Testing the MARS Functionality	1. Press **F5**. Note that the order totals are output for the customers. 2. Press **Enter**. 3. Select the **File | Close Solution** menu command. Next you will see how ADO.NET supports asynchronous operations.
Task 4: Examining the Async Project	1. Select the **File | Open | Project/Solution** menu command. 2. Navigate to **C:\SQL Labs\Lab Projects\SQL CLR Lab\Async**. 3. Load the Async.sln solution in Visual Studio 2005. Don't save changes to the previous solution, when prompted. 4. In the **Solution Explorer**, double-click **Class1.cs**. 5. Examine the following code: ``` SqlConnection cnn = new SqlConnection("Data Source=localhost;" + "Initial Catalog=AdventureWorks;" + "Integrated Security = SSPI;" + "Asynchronous Processing=true"); cnn.Open(); SqlCommand cmd = new SqlCommand("IF EXISTS (SELECT * FROM dbo.sysobjects " + ```

Tasks	Detailed Steps
	`"WHERE id = object_id(N'[dbo].[AsyncLab]') " +` `"AND OBJECTPROPERTY(id, N'IsUserTable') = 1) " +` `"DROP TABLE [dbo].[AsyncLab];" +` `"SELECT * INTO AsyncLab FROM Sales.SalesOrderDetail;" +` `"SELECT COUNT(*) FROM AsyncLab", cnn);` The SqlCommand is populated with a statement that will take a noticeable amount of time to execute. This command will be executed asynchronously so that the application can perform other operations while the statement completes.
Task 5: Adding Asynchronous Processing	1. Add the following code (which is in the helper file) after the **TODO: Process asynchronously** comment: `IAsyncResult iar = cmd.BeginExecuteReader();` `while (!iar.IsCompleted)` `{` `Thread.Sleep(100);` `Console.Write(".");` `}` `SqlDataReader dr = cmd.EndExecuteReader(iar);` `dr.Read();` `Console.WriteLine("\nProcessing complete, " +` `dr[0].ToString() +` `" rows copied");` You can see that ADO.NET now supports the asynchronous design pattern used throughout the base class library. When BeginExecuteReader is called, the query is executed in the background. You can be notified of completion a couple of ways. You can pass a delegate for a callback method specified in the call to BeginExecuteReader. This method is invoked when the query completes. You can also poll the IsCompleted property of the IAsyncResult. The latter approach is used here. Once the query completes, the SqlDataReader can be populated by calling EndExecuteReader, passing the IAsyncResult as an argument.
Task 6: Testing the Asynchronous Functionality	1. Select the **Debug | Start** menu command (or press **F5**). Note that periods are output while the SQL statement executes, and the total number of rows copied is displayed when the query completes. 2. Press **Enter**. 3. Close the solution without saving changes.

This lab showed you how Microsoft SQL Server 2005 and Visual Studio 2005 combine to provide a new level of power and flexibility when developing database and data-centric applications. You learned that the .NET Common Language Runtime is now hosted in SQL Server 2005 so that you can use stored procedures, user-defined functions, user-defined types, triggers, and aggregates using managed code with languages such as Visual Basic .NET and Visual C#. Although managed code does not replace T-SQL, the ability to use managed code is an important new tool in the database developer's toolbox.