Wrox Programmer to Programmer™

Professional

Microsoft® SQL Server® 2008 Integration Services

Brian Knight, Erik Veerman, Grant Dickinson, Douglas Hinson, Darren Herbold

WROX

Updates, source code, and Wrox technical support at **www.wrox.com**

9

Scripting in SSIS

With the introduction of C#, and the embedding of the new Visual Studio Tools for Applications into SSIS, you can't think of using the Script Task and Script Component as scripting anymore; now it's all-out programming. In the early days of DTS-based SQL Server ETL processing, the ActiveX Script Task allowed you to embed programmatic logic and became the Swiss-Army knife of package development. Typically, you'd code logic into these Script Tasks to control the execution and logic flow within a package or to perform some specialized business validation.

Scripting in SSIS has completely evolved from these simple ActiveX roots. You've still got a Swiss-Army knife hidden in here, but there is a separation of functionality from previous uses of ActiveX scripting into three new concepts: the Scripting Task, the Scripting Component, and Expressions. Expressions are completely new to SSIS and replace the old methodology of manipulating variables or properties within the package model. The other two Scripting Components provide access into a new scripting development environment using Microsoft Visual Studio Tools for Applications (VSTA). This change finally allows SSIS developers to script logic into packages using Visual Basic 2008 *or* Visual C# 2008 .NET code.

In this chapter, you learn all about these new scripting options and learn how to exploit them in your package development tasks to control execution flow, perform custom transformations, manage variables, and provide runtime feedback.

Scripting?

If you think of scripting as having to compile at runtime and unstructured or unmanaged coding languages, then this is not scripting. If you think scripting means small bits of code in specialized places to execute specific tasks, then yes, it's scripting. The current scripting abilities have come a long way from their predecessors in DTS, and even the earlier versions of SSIS. Whether you're a grizzled veteran of DTS, or if SSIS is your first exposure to SQL Server ETL development, it is helpful to understand the historical landscape of the Scripting Components, and why there is now a separation by functional usage. We'll examine this and open up the new scripting IDE

environment to walk though the mechanics of applying programmatic logic into these components — including how to add your own classes and compiled assemblies.

The ActiveX Task in the past was useful no doubt, but it was primitive. The task allowed coding in only the two ActiveX-based scripting languages, VB and J script, with no IntelliSense help. However, if you look around at historical DTS package development (or if you've had to convert any to SSIS) you'll find some significant creative work going on in these tasks. Digging into what developers were doing, the functional activities can be broken up into these categories:

- Retrieving or setting the value of package variables
- Retrieving or setting properties within the package
- Applying business logic to validate or format data streams
- Controlling workflow in a package
- Performing miscellaneous tasks not supported by existing package components

Retrieving and setting the value of variables and package properties was so prevalent in DTS that the SSIS team decided to create a completely different feature to allow this to be less of a programmatic task. The expressions editor, in SSIS, allows package components to be easily altered by setting component properties to an expression, or variable that represents an expression. This concept is a maturation and replacement of the Dynamic Property Component that was a part of DTS development. See Chapter 6 for information on how to use expressions and variables; they are out of scope for this chapter.

The Data Pump Task in DTS was limited. Delimited data or supported sources could be mapped, but outside of these narrow constraints, you were on your own. Because of this, you can find DTS packages that performed all the ETL steps written purely in ActiveX script. In ActiveX Script Tasks, you could connect to Data Sources via ADO, parse and manipulate the data, and push it into destination sources. In SSIS, this need was replaced and expanded with the Data Flow Container that allows this type of activity to be visually represented. However, to perform the ad-hoc data messaging functionality, scripting is still needed, so the Script Component was added. The primary role of the Scripting Component is to extend the Data Flow capabilities and allow programmatic data manipulation within the context of the data stream. However, it can do more as you'll learn later.

To continue to enable the numerous miscellaneous tasks that are needed in ETL development, the ActiveX Script Task has been replaced with the Script Task, which can be used only in the Control Flow design surface. In this task, all the various things that you could do with the ActiveX Task can be replicated, but within the managed code framework of .NET.

Today, you'll also still find the ActiveX Script Task in the BIDS environment. However, this is only to support backward compatibility and even then for only some of the more simple uses. You can get some of the functionality that used to work in DTS, but not everything converts over. For one, setting package properties, which was prevalent in DTS development, is no longer allowed in SSIS. If you are currently using this task, we recommend you get it into one of these new Scripting Components as soon as possible, since no one knows how long the ActiveX Task will be supported.

The initial versions of SSIS Script Components stopped half-way down a path of complete replacement of the older Visual Basic for Applications (VBA) implementation that had previously been used for scripting to a .NET environment. One downside for many was that only the VB language was supported.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

The latest versions of these Script Components host the new Visual Studio Tools for Applications (VSTA) environment, which is the replacement for the Visual Basic for Applications (VBA) implementation. VSTA is essentially a scaled-down version of Visual Studio that can be added to an application that allows coding extensions using managed code and .NET languages. Even though SSIS packages are built inside of VS, when you are in the context of a Script Task or Component, you are actually coding in the VSTA environment that is, in fact a mini-project within the package. The new VSTA IDE provides IntelliSense, full edit-and-continue capabilities, as well as the ability to code in either VB or C#. The IDE is available when you edit script and looks like Visual Studio. You can now even access some of the .NET assemblies and even use web references. Earlier versions of SSIS required the creation of a proxy class to use a Web service.

Getting Started in SSIS Scripting

The new Script Task and Script Component, combined with the addition of VSTA to the BIDS environment, has really opened up the possibilities when it comes to script-based ETL development in SSIS. However, you may find it confusing at first to know when to use which component and what things can be done in each. Although two have the word "script" in the names, they have different usages, and even coding tasks such as variable retrieval are different in each. You need to know when to use which component and how to do similar tasks in each. First, to keep them all straight, here's a matrix to explain when to use each component:

Component	When to Use
ActiveX Script Task	Use only if you are in the middle of converting a DTS package to SSIS. This component should not be used in new development.
Script Task	Use this task when you need to program logic that either controls package execution or performs a task of retrieving or setting variables within a package during runtime.
Script Component	Use this component when pumping data using the Data Flow Task. Here you can apply programmatic logic to massage data in the pipeline.
Expressions	Use expressions to set task and component properties or variables during runtime. See Chapter 6 for more detail.

To get a good look at the scripting model, we'll walk through a TextBox "Hello World" coding project in SSIS. Although this is not a typical example of ETL programming, we'll use this as a start to understanding the scripting paradigm in SSIS with a basic Script Task. Then we'll look at specific applications of each Scripting Component.

Selecting the Scripting Language

One of the new capabilities of SSIS using the VSTA environment is the addition of the C# scripting language to the existing VB coding option. To see where you can make this choice, drop a Script Task onto the Control Flow design surface. Right-click the Script Task and click Edit from the drop-down menu. The first thing you'll notice is the availability of the Microsoft Visual C# option in the ScriptLanguage property of the task available in both the Script Task and Component. Figure 9-1 shows these options in the Script Task Editor.

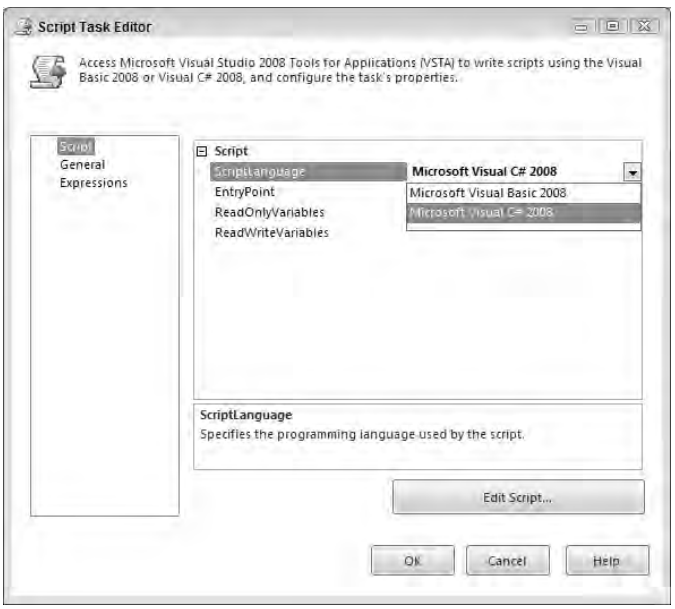

Figure 9-1

Once you click the Edit Script button, you'll be locked into the script language that you chose and won't be able to change it without deleting and recreating the Script Task. This is because each Script Component contains its own internal Visual Studio project in VB or C#. You can create separate Script Tasks where each one uses a different language within a package. However, having Script Tasks in both languages is not recommended because this makes maintenance of the package a more complex issue. The developer maintaining the package would have to be competent in both languages.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

Using the VSTA Scripting IDE

To add programmatic code to a Script Task or Component, access its editor by right-clicking the component and selecting the Edit option from the drop-down menu. While the Script Task and Script Component editors look completely different, they both have a common way to access the development IDE for scripting. In Figure 9-2, notice the same Edit Script button that is in both editors.

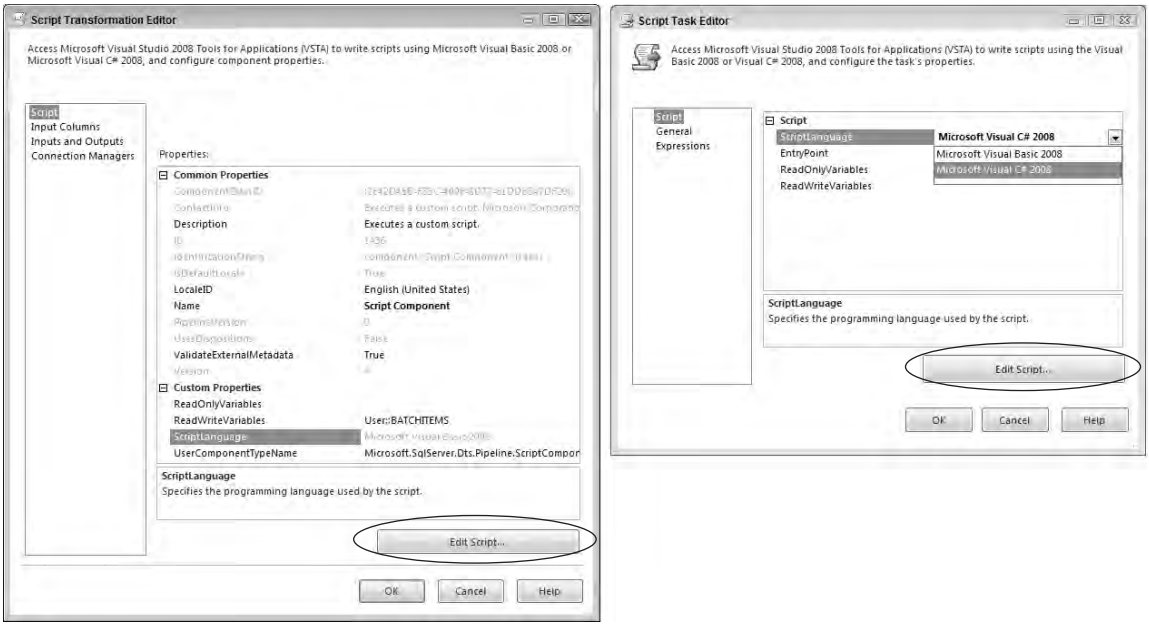

Figure 9-2

The button labeled Edit Script provides access into the scripting IDE. Once in the IDE, notice that it really looks and feels just like Visual Studio. Figure 9-3 shows an example of how this IDE looks after opening up the Script Task Component for the C# scripting language.

The previous Visual Studio 2005 VBA implementation of the scripting IDE presented the coding IDE like the Macro VBA environment in Excel. If you are still using this older environment, the same look can be achieved by navigating to the View menu option and selecting the Project Explorer and Property windows. Arrange them on the right side of the IDE to make the 2005 BIDS scripting IDE look almost like the new Visual Studio work environment.

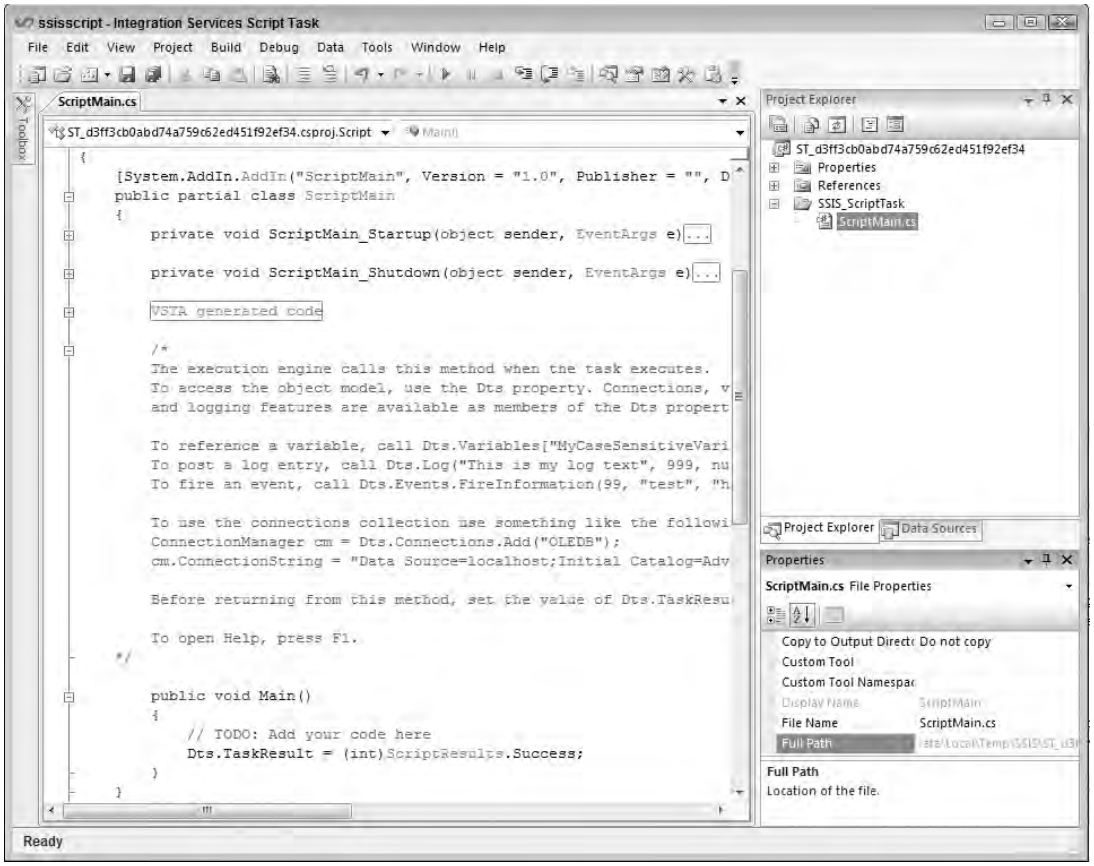

Figure 9-3

The code window on the left side of the IDE contains the code for the item selected in the Project Explorer on the top-right window. The Project Explorer shows the structure for the project that is being used within the Scripting Task. A complete .NET project is created for each Script Task or Component, and is temporarily written to a project file on the local drive where it can be altered in the Visual Studio IDE. This persistence of the project is the reason that once you pick a scripting language, and generate code in the project within, you'll be locked into that language for that Scripting Component. Notice in Figure 9-3 that a project has been created with the namespace of ST_d3ff3cb0abd74a759o62ed451f92ef34. The path to this temporary project can also be seen under the property Full Path for the `ScriptMain` class shown in Figure 9-2. However, you can't open up this project directly, nor need you worry about the project during deployment. These project files are extracted from stored DTS Package metadata similar to the way the SQL CLR objects are stored as metadata in SQL Server. With the project created and opened, it is ready for coding.

Example: Hello World

In the IDE, the Script Task only contains a class named `ScriptMain`. If you open a Script Component, you'll see more classes to support the component and the data buffer. We'll discuss what these additional classes do a little later in the chapter during the examination of the Script Component. Both components

use a public class named `ScriptMain`, but the filename you see in the Project Explorer will be either `ScriptMain` or `Main` to host a public entry-point function. The function name is different for the Script Task than the Script Component because the interfaces and purposes are different. In the entry-point functions, `Main()` for the Script Task, you'll either put all the code you want to execute, or you can call into separately defined functions or classes either in or out of process. However, if you want to change the entry-point function for some reason in the Script Task only, type the name of the entry-point function in the property called EntryPoint on the Script page of the editor. (Alternatively, you could change the name of the entry point at runtime using an expression.)

In the VSTA co-generated class `ScriptMain`, you'll also see a set of assembly references already added to your project and namespaces set up in the class. Depending upon whether you chose VB or C# as your scripting language, you'll see either:

```
C#
Using System
Using System.Data
Using Microsoft.SqlServer.Dts.Runtime.VSTAProxy;
Using System.Windows.Forms;
```

Or:

```
VB
Imports System
Imports System.Data
Imports System.Math
Imports Microsoft.SqlServer.Dts.Runtime.VSTAProxy
```

These assemblies are needed to provide base functionality as a jump start to your coding. The remainder of the class includes VSTA co-generated methods for startup and shutdown operations, and finally the entry-point `Main()` function shown here in both languages:

```
C#
public void Main()
{
    // TODO: Add your code here
    Dts.TaskResult = (int)ScriptResults.Success
}

VB
Public Sub Main()
    '
    ' Add your code here
    '
    Dts.TaskResult = ScriptResults.Success
End Sub
```

This `Main()` function is a good example of one of the differences between the Script Task and the Script Component. A Script Task must return a result to notify the runtime of whether the script completed successfully or not. This result is that the `Dts.TaskResult` property is being set to indicate to the package that the task completed successfully. The Script Component does not have to do this, since it runs in the context of a data pump with many rows. There are other differences pertaining to each component that we'll discuss separately later in the chapter.

To get a message box to pop up with the phrase "Hello, World!" you need access to a class called `MessageBox` in a namespace called `System.Windows.Forms`. This namespace can either be called directly by the complete name, or the namespace can be added after the `Microsoft.SqlServer.Dts.Runtime` namespace to shorten the coding required in the class. Both of these methods are shown in the following code to insert the `MessageBox` code into the `Main()` function:

```
C#
using System.Windows.Forms
. . .
MessageBox.Show("Hello World!");
Or
System.Windows.Forms.MessageBox.Show("Hello World!");

VB
Imports System.Windows.Forms
. . .
MessageBox.Show("Hello World!")
Or
System.Windows.Forms.MessageBox.Show("Hello World!")
```

After you have added this code, get in the habit now of building the project when you are finished with the coding. The Build option is directly on the menu when you are coding. Previous versions of SSIS gave you the opportunity to run in precompile or compiled modes. SSIS now will automatically compile your code prior to executing the package in the runtime. Compiling gives you an opportunity to see what the errors are before the package finds them. Once the build is successful, close the IDE, the editor, and right-click and execute the Script Task. A pop-up message box should appear with the words "Hello World!" like Figure 9-4.

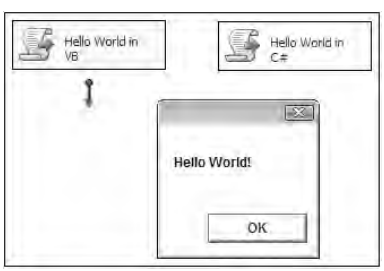

Figure 9-4

Adding Code and Classes

Using modal message boxes is obviously not the type of typical coding we want to do in production SSIS package development. Message boxes are synchronous and block until a click event is received, so they can stop a production job dead in its tracks. However, this is a basic technique to demonstrate the capabilities in the new scripting environments before getting into some of the details of passing values in and out using variables. We also don't want to always put the main blocks of code in the `Main()` function. With just a little more work, we can get some code reuse from previously written code using some cut-and-paste development techniques. At the very least, code can be structured in a less-procedural way. Consider a common task of generating a unique filename to give an archived file.

Typically, the filename might be generated by appending a prefix and an extension to something variable like a datetime value.

These functions can be added within the `ScriptMain` class bodies to look like this:

```
C#
Public partial class ScriptMain
{
   . . .
   public void Main()
   {
      System.Windows.Forms.MessageBox.Show(GetFileName("bankfile", "txt"));
      Dts.TaskResult = (int)ScriptResults.Success;
   }

   public string GetFileName(string Prefix, string Extension)
   {
       return Prefix + DateTime.Now.ToString("yyyyMMddhhmmss") +
                "." + Extension;
   }
}

VB
Partial Class ScriptMain
    . . .
    Public Sub Main()
        System.Windows.Forms.MessageBox.Show(GetFileName("bankfile", "txt"))
        Dts.TaskResult = ScriptResults.Success
    End Sub

    Public Function GetFileName(ByVal Prefix As String, _
                                ByVal Extension As String) As String
        GetFileName = Prefix + DateTime.Now.ToString("yyyyMMddhhmmss") + _
                      "." + Extension
    End Function
End Class
```

Instead of all the code residing in the `Main()` function, we can separate and organize SSIS scripting using structured programming techniques. In this example, the `GetFileName` function builds the filename and then returns the value in the message box, as shown in Figure 9-5.

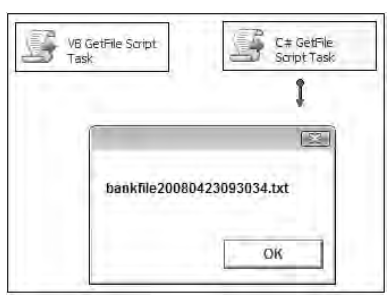

Figure 9-5

But copying code and pasting it into multiple Script Components is pretty cheesy. If you have preexisting compiled code, shouldn't you be able to reuse this code without finding the original source for the copy-and-paste operation? You can, with some qualification.

Using Managed Assemblies

The capability to reuse code written in other languages is the hallmark of COM and its successor, .NET. While you can only write SSIS scripts using Visual Basic.NET and C#, you can reuse assemblies that are part of the .NET core assemblies or any assembly created using a .NET-compliant language, including C#, J#, and even Delphi, but there are some qualifications. These are rather important so we'll state them like this:

- For a managed assembly to be used in an Integration Service, you must install the assembly in the Global Assembly Cache (GAC).
- Additionally, all dependent or referenced assemblies must also be registered in the GAC. This implies that the assembly must be strongly named.
- For development purposes only, VSTA can use managed assemblies anywhere on the local machine.

If you think about this it makes sense, but within SSIS, it might seem confusing at first. On one hand, a sub-project is created for the Script Component, but it is absorbed into the metadata of the package. In this case, you don't have to worry about deployment of individual script projects. However, when you use an external assembly, it does not get absorbed into the package metadata and here you *do* have to worry about deployment of the assembly. So where do you deploy the assembly you want to use? Since DTS packages can be deployed within SQL Server, the most universal place to find the assembly would be in the GAC.

If you are using any of the standard .NET assemblies, they are already loaded and stored in the GAC and the .NET Framework folders. As long as you are using the same framework for your development and production locations, using standard .NET assemblies requires no additional work. To use a standard .NET assembly in your script, you must reference it. To add a reference to a scripting project, you must be in the VSTA environment for editing your script code — not the SSIS package itself. Right-click the References Node in the Project Explorer, or go to the Project menu and select the Add Reference option. The standard .NET Add Reference window will appear as shown in Figure 9-6.

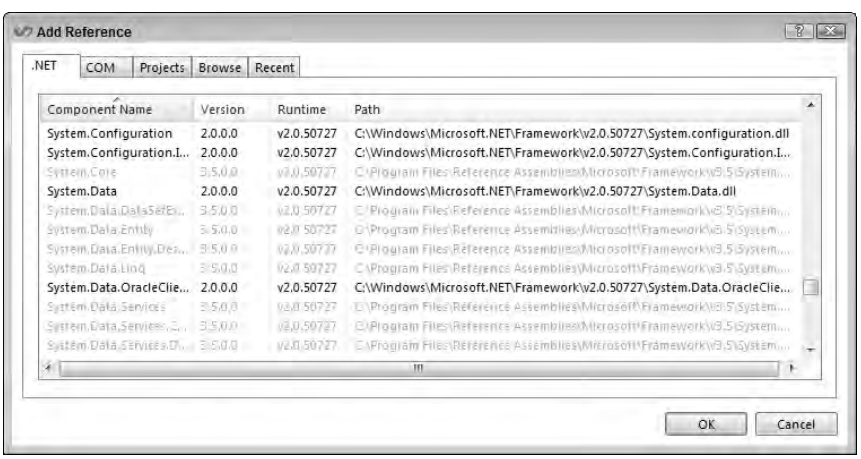

Figure 9-6

Select the assemblies from the list that you wish to reference and click the OK button to add the references to your project. Now you can use any objects located in the referenced assemblies by either directly referencing the full assembly or by adding the namespaces to your `ScriptMain` classes for shorter references. References can also be removed by right-clicking the reference in the References Node of the Project Explorer. (The References Node is hidden in the VB project. Click the menu option Project ⇨ Show All Files to make this node visible.) Expand the References Node to see all the references in your project. Right-click a reference and select the Remove option to remove it from the project.

Example: Using Custom .NET Assemblies

Although using standard .NET assemblies is interesting, being able to use already compiled .NET assemblies really opens up the capabilities of your SSIS development. Using code already developed and compiled means not having to copy-and-paste into each Script Task or Component and allows you to reuse code already developed and tested. To show an example of how this works, you'll create an external custom .NET library that can validate a postal code and see how to integrate this simple validator into a Script Task. (To do this, you'll need the standard class library templates that are part of Visual Studio. If you only installed BIDS, these templates are not installed by default.) You can also download the precompiled versions of these classes as well as any code from this chapter at `www.wrox.com`.

To start, open up a standard class library project in the language of your choice, and create a standard utility class in the project that looks something like this:

```
C#
using System;
using System.Text.RegularExpressions;
namespace ssistestlib
{
    public static class DataUtilities
    {
        public static bool isValidUSPostalCode(string PostalCode)
        {
            return Regex.IsMatch(PostalCode, "^[0-9]{5}(-[0-9]{4})?$");
        }
    }
}
VB
Imports System.Text.RegularExpressions

Public Class DataUtilities
    Public Shared Function isValidUSPostalCode
                              (ByVal PostalCode As String) As Boolean
        isValidUSPostalCode = Regex.IsMatch(PostalCode,
                                    "^[0-9]{5}(-[0-9]{4})?$")
    End Function
End Class
```

Since you are creating projects for both languages, the projects (and assemblies) are named `SSISUtilityLib_VB` and `SSISUtilityLib_Csharp`. Notice the use of *static* or *shared* methods. This is not required, but is useful because you are simulating the development of what could later be a utility library loaded with many stateless data validation functions. A static or shared method allows the utility functions to be called without instantiating the class for each evaluation.

Now sign the assembly by right-clicking the project to access the Properties menu option. In the Signing tab, there is an option to select Sign the assembly, as shown in Figure 9-7. Click New on the drop-down and name the assembly to get a strong name key added to the assembly.

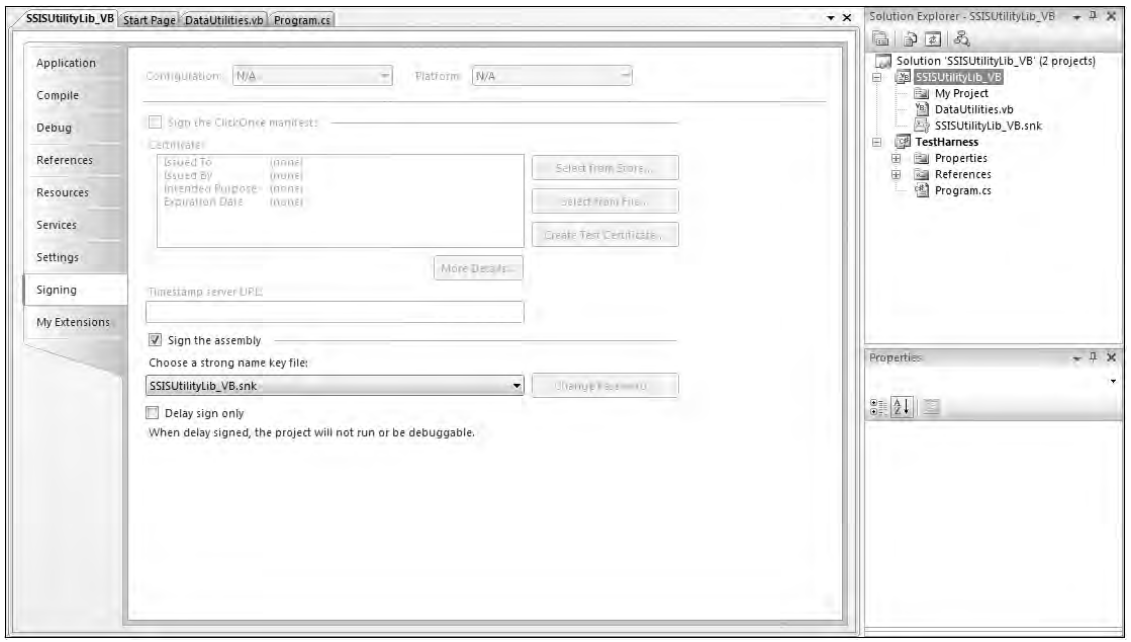

Figure 9-7

In this example, the VB version of the `SSISUtilityLib` project is being signed. Now, the assembly can be compiled by clicking the Build option in the Visual Studio menu and the in-process DLL will be built with a strong name to allow it to be registered in the GAC.

On the target development machine, go to the command-line prompt from the Visual Studio Tools menu to register your assembly with a command like this:

```
C:\Program Files\Microsoft Visual Studio 9.0\VC>gacutil /I
c:\ssis\scripts\SSISUtilityLib_VB\SSISUtilityLib_VB\bin\debug\
SSISUtilityLib_VB.dll
```

Note that you may have to run the command line as administrator or have the User Access Control feature of Vista turned off to register the assembly.

If you are running on a production machine, you'll also need to copy the assembly into the appropriate .NET Framework directory so that you can use the assembly in the Visual Studio IDE. Use the Microsoft .NET Framework 2.0 Configuration wizard task to Manage the Assembly Cache. Select Add an Assembly to the Assembly Cache to copy an assembly file into the global cache.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

To use the compiled assembly in an SSIS package, open a new SSIS package and add a new Script Task onto the Control Flow surface. Select the scripting language you wish and click Edit Script. You'll need to right-click the Project Explorer Node for references and find the reference for `SSISUtilityLib_VB.dll` or `SSISUtilityLib_CSharp.dll` depending upon which one you built. (Remember that you may have to use the menu option Project ⇨ Show All Files in the VB projects to see the References Node.) If you've registered the assembly in the GAC, you'll be able to find it in the .NET tab. If you are in a development environment, you can simply browse to the `.dll` to select.

Add the namespace into the `ScriptMain` class. Then add these namespaces to the `ScriptMain` class:

```
C#
using SSISUtilityLib_CSharp;

VB
Imports SSISUtilityLib_VB
Imports System.Windows.Forms
```

Note that the SSIS C# Script Task in the sample packages that you'll see if you download the chapter materials from `www.wrox.com` use the C# version of the utility library. However, this is not required. The compiled .NET class libraries may be intermixed within the SSIS Script Task or Components regardless of the scripting language you choose. All that is left is to code a call to the utility function into the `Main()` function like this:

```
C#
public void Main()
{
    string postalCode = "12345-1111";
    string msg = string.Format(
          "Validating PostalCode {0}\nResult..{1}", postalCode,
                      DataUtilities.isValidUSPostalCode(postalCode));
    MessageBox.Show(msg);
    Dts.TaskResult = (int)ScriptResults.Success;
}

VB
Public Sub Main()
    Dim postalCode As String = "12345-1111"
    Dim msg As String = String.Format("Validating PostalCode {0}" +
         vbCrLf + "Result..{1}", postalCode,
         DataUtilities.isValidUSPostalCode(postalCode))
    MessageBox.Show(msg)
    Dts.TaskResult = ScriptResults.Success
End Sub
```

Compile the Script Task and execute it. The result should be a message box displaying a string to validate the postal code 12345-1111. The postal code format is validated by the DataUtility function `IsValidUSPostalCode`. There was no need to copy the function in the script project. The logic of validating the format of a U.S. Postal code is stored in the shared DataUtility function and can easily be used in both Script Tasks and Components with minimal coding and maximum consistency. The only downside to this is that there is now an external dependency in the SSIS package upon this assembly. If the assembly changes version numbers, you'll need to open and recompile all the script projects for each SSIS package using this. Otherwise, you could get an error if you aren't following backward compatibility

guidelines to ensure that existing interfaces are not broken. If you have a set of well-tested business functions that rarely change, using external assemblies may be a good idea for your SSIS development.

Using the Script Task

Now that you've gotten a good overview of the scripting environment in SSIS, it's time to dig into one of the Scripting Components and give it a spin. We used the Script Task heavily to demonstrate how the SSIS scripting environment works with Visual Studio and during the execution of a package. Generally, anything that you can script in the .NET managed environment that should run once per package or code loop belongs in the Script Task. Script Tasks are extremely useful and end up being the general-purpose utility component similar to the role ActiveX Tasks performed for DTS package development.

Configuring the Script Task Editor

Earlier we looked at the Script Task Editor to point out that there are now two selections available for the scripting language. Let's look at that editor again and go over the other details. Drop a Script Task on the Control Flow surface and display the editor you see in Figure 9-8.

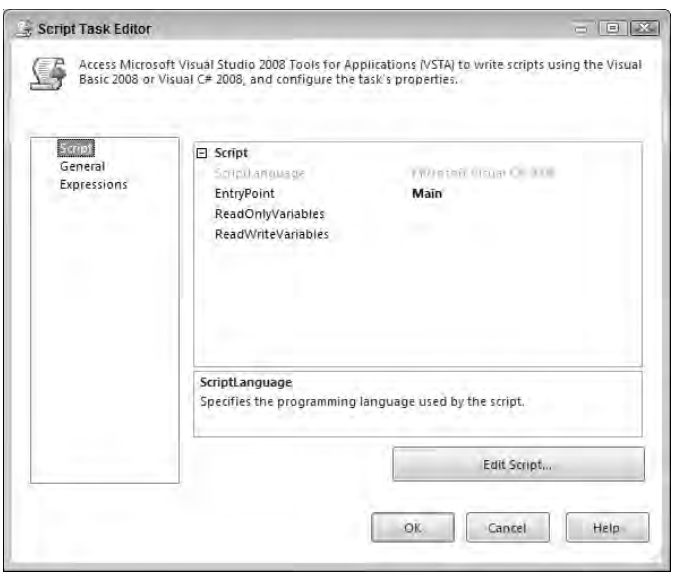

Figure 9-8

Here are the four properties on the Script tab that you can use to configure the Script Task:

- **ScriptLanguage:** This property defines the .NET language that will be used for the script. As demonstrated earlier, VB and C# are now both supported.
- **EntryPoint:** This is the name of the class that must contain a public `Main()` method that will be called inside your script to begin execution.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

- **ReadOnlyVariables:** This property enumerates a case-sensitive, comma-separated list of SSIS variables that you will allow explicit rights to be read by the Script Task.
- **ReadWriteVariables:** This property enumerates a case-sensitive, comma-separated list of SSIS variables that you are allowing to be read from and written to by the Script Task.

Missing from the first release of SSIS is a property that allowed the option to precompile script code into binary code before execution. In the latest version of SSIS, all scripts are precompiled by default. This is part of the performance improvements made to reduce the overhead of loading the language engine when running a package.

The second tab, labeled General, contains the properties for the task name and description. The latest version of SSIS moves this tab down since it is not accessed as frequently as the Script tab.

The final page available on the left of this dialog is the Expression tab. The Expression tab provides access to the properties that can be set using an expression or expression-based variable. See Chapter 6 for how to use expressions and variables. Practically, changing the `ScriptLanguage` at runtime is not really possible, nor desired. The most common property manipulated by an expression is the Disable property that is used to bypass the task.

Once the script language is set and the script accessed, a project file with a class named `ScriptMain` and a default entry point named `Main()` is created. An example of the `Main()` function is provided here without the supporting class:

```
C#
public void Main()
{
    // TODO: Add your code here
    Dts.TaskResult = (int)ScriptResults.Success;
}

VB
Public Sub Main()
    Dts.TaskResult = ScriptResults.Failure
End Try
```

The code provided automatically includes the statement to set the `TaskResult` of the `Dts` object to the enumerated value for Success. The Script Task itself is a task in the collection of tasks for the package. Setting the `TaskResult` property of the task sets the return value for the Script Task and tells the package whether the end result was a success or failure.

By now, you've probably noticed all the references to DTS. What is this object and what can you do with it? We'll answer this question in the next section, as you peel back the details on the DTS object.

The Script Task Dts Object

The `Dts` object is actually a property on your package that is an instance of the `Microsoft.SqlServer.Dts.Tasks.ScriptTask.ScriptObjectModel`. The `Dts` object provides a window into the package in which your script executes. While you can't change properties of the DTS as it executes, the `Dts` object

has seven properties and one method that allow you to interact with the package. The following is an explanation of these members:

- `Connections`: A collection of Connection Managers defined in the package. You can use these connections in your script to retrieve any extra data you may need.
- `Events`: A collection of events that are defined for the package. You can use this interface to fire off these predefined events and any custom events.
- `ExecutionValue`: A read-write property that allows you to specify additional information about your task's execution using a user-defined object. This can be any information you want.
- `TaskResult`: This property allows you to return the Success or Failure status of your Script Task to the package. This is the main way of communicating processing status or Controlling Flow in your package. This property must be set before exiting your script.
- `Transaction`: Obtains the transaction associated with the container in which your script is running.
- `VariableDispenser`: Gets the `VariableDispenser` object that you can use to retrieve variables when using the Script Task.
- `Variables`: A collection of all the variables that are available to any script. This is used by default in the Script Component.
- `Log`: This method allows you to write to any log providers that have been enabled.

DTS developers are sometimes locked into the fact that the Script Task can no longer alter an executing package, but in truth, between the additions of the expressions and the `Dts` object, you can do almost everything you could want to with the executing package. The method is just different. In the next few sections, we'll go through some of the more common things that the Active Script Task can be employed to accomplish.

Accessing Variables in the Script Task

Variables and expressions are an important feature of the SSIS roadmap. We aren't talking about scripting variables, but rather package variables that serve as intermediate communication mediums between your Script Task and the rest of your package. As discussed in Chapter 6, variables are used to drive the runtime changes within a package by allowing properties to infer their values at runtime from variables, which can be static or defined through the expression language.

The common method of using variables is to send them into a Script Task as decision-making elements or to drive downstream decisions by setting the value of the variable in the script based on some business rules. The `VariableDispenser` object provides methods for locking variables for read-only or read-write access and then retrieving them. Initially this was the standard way of accessing variables in scripts. The reason for the explicit locking mechanism is to allow control in the Script Task to keep two processes from competing for accessing and changing a variable.

To retrieve a variable using the `VariableDispenser` object, you would have to deal with the implementation details of locking semantics, and write code like the following:

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

```
C#
Variables vars = null;
String myval = null;
Dts.VariableDispenser.LockForRead("User::SomeStringVariable")
Dts.VariableDispenser.GetVariables(ref vars)
Myval = vars[0].Value;
vars.Unlock();  //Needed to unlock the variables
System.Windows.Forms.MessageBox.Show(myval);

VB
Dim vars As Variables
Dim myval As String
Dts.VariableDispenser.LockForRead("User::SomeStringVariable")
Dts.VariableDispenser.GetVariables(vars)
myval = vars(0).Value
vars.Unlock()  'Needed to unlock the variables
MsgBox(myval)
```

The downside to this method is that it was easy to forget to unlock the variables in an efficient way and as a result, a variable could be locked and rendered unavailable downstream in the package.

However, this type of control is not always required. Sometimes you simply want the variables that you are using in a Script Task to be locked when you are reading and writing, and not have to worry about the locking implementation details. Luckily, a much easier abstraction was created to add a `Variables` collection on the `Dts` object, and the `ReadOnlyVariables` and `ReadWriteVariables` properties for the Script Task were introduced. The only constraint is that you have to define upfront which variables going into the Script Task can be read and not written to, and which ones can be read and writable.

The `ReadOnlyVariables` and `ReadWriteVariables` properties tell the Script Task which variables to lock and how. The `Variables` collection in the `Dts` object then gets populated with these variables. The code to retrieve a variable then becomes much simpler, and the complexities of locking are abstracted, so you only have to worry about one line of code to read a variable:

```
C#
Dts.Variables["User::SomeStringVariable"].Value;
or
Dts.Variables[0].Value;

VB
Dts.Variables("User::SomeStringVariable").Value
Or
Dts.Variables(0).Value
```

Using the ordinal position of the variable in the Variables collection is the safest method if you are unsure of how to name the variable in your script. Just remember that the variables are ordered from the editor left to right, starting in the `ReadOnlyVariables` and then down to the `ReadWriteVariables`, also moving left to right. Now, if you choose to use the named variable, you are safer to use the fully qualified variable name like `User::SomeStringVariable`. Attempting to read a variable from the Variables collection that hasn't been specified in one of the variable properties of the task will throw an exception. Likewise, attempting to write to a variable not included in the `ReadWriteVariables`

property also throws an exception. The biggest frustration for new SSIS developers writing VB script is dealing with this error message:

```
Error: 0xc0914054 at VB Script Task: Failed to lock variable
"SomestringVariable" for read access with error 0xc0910001 "The variable
cannot be found. This occurs when an attempt is made to retrieve a variable
from the Variables collection on a container during execution of the package,
and the variable is not there. The variable name may have changed or the
variable is not being created."
```

The resolution is simple. Either the variable name listed in the Script Task Editor or the variable name in the script doesn't match, so one must be changed to match the other. It is more confusing for the VB developers because this language is not case-sensitive. However, the SSIS variables are case-sensitive, even within the VB script.

Although Visual Basic.NET is not case-sensitive, SSIS variables are.

Another issue that happens occasionally is that more than one variable with the same name can be created with different scopes. When this happens, you have to make sure you explicitly refer to the variable by the fully qualified variable name. One of the latest helpful features of SSIS is a pop-up UI that allows the selection of the variables. Figure 9-9 is an example of this UI that allows the selection of user-defined variables.

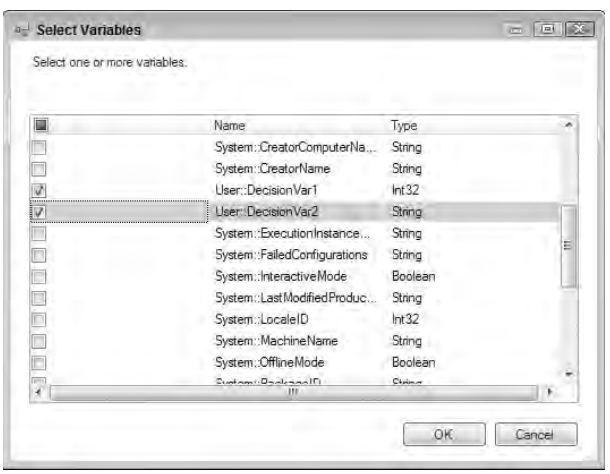

Figure 9-9

The best part is that the Script Task property for the `ReadOnlyVariables` or `ReadWriteVariables` is auto-filled with the fully qualified names: `User::DecisionVar1` and `User::DecisionVar2`. This reduces most of the common issues with passing variables into the Script Task. All this information will now come in handy as we run through an example using the Script Task and variables to control SSIS package flow.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

Example: Using Script Task Variables to Control Package Flow

In this example, we'll set up a Script Task that uses two variables to determine which one of two branches of Control Flow logic should be taken when the package executes. First, create a new SSIS package and set up these three variables:

Variable	Type	Value
`DecisionIntVar`	`Int32`	`32`
`DecisionStrVar`	`String`	`txt`
`HappyPathEnum`	`Int32`	`0`

Then drop three Script Tasks on the Control Flow design surface so that the package looks like Figure 9-10.

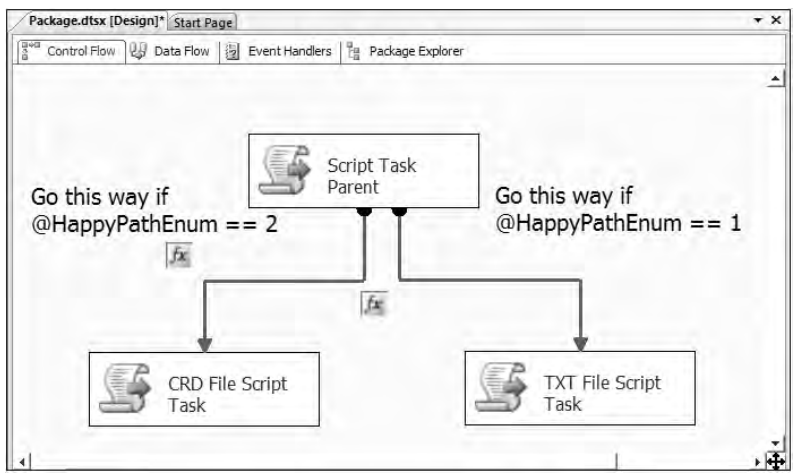

Figure 9-10

What we want to do is feed the two variables (`DecisionIntVar` and `DecisionStrVar`) that represent the number of rows determined to be in a file and the file extension into the Script Task through these variables. Assume that these values have been set by yet another process. Logic in the Script Task will determine whether the package should execute the CRD File Path Script Task or the TXT File Script Task. The control of the package is handled by the other external variable named `HappyPathEnum`. If the value of this variable is equal to 1, then the TXT File Script Task will be executed. If the value of the variable is equal to 2, then the CRD File Path Script Task will be executed. Open up the main Script Task Editor to set up the properties. It should look like Figure 9-11.

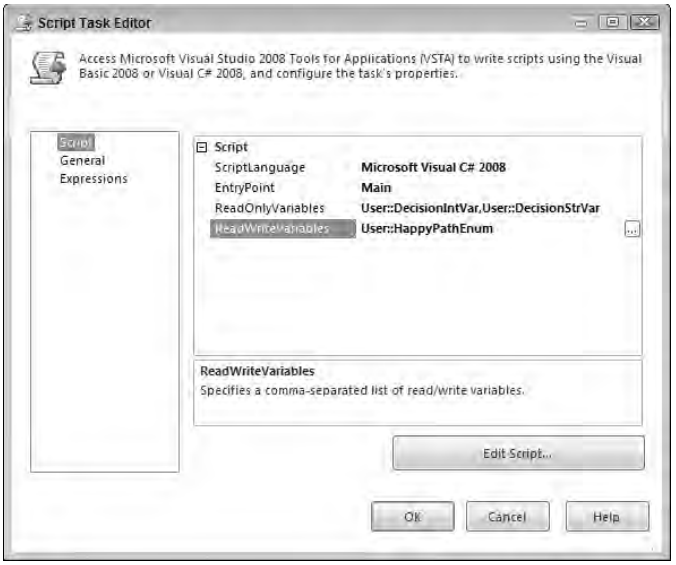

Figure 9-11

Set the Script Language and then use the ellipsis button to bring up the variable selection UI that we discussed and demonstrated in Figure 9-9. Select the variables for the ReadOnlyVariables and ReadWriteVariables separately if you are using the variable selection UI. You can also type these variables in, but remember that the variable names are case-sensitive. It is noteworthy to stop and point out the ordinal positions of these variables for this example. You can see the ordinal positions superimposed onto the editor in Figure 9-12.

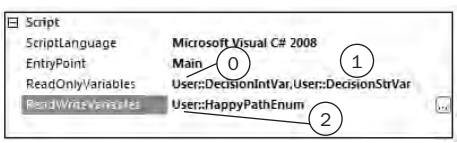

Figure 9-12

We'll keep this script simple for demonstration purposes. The most important parts are the retrieving and setting of the variables. This code uses the named references for the variables but the code lines like this:

```
C#
int rowCnt = (int)Dts.Variables["User::DecisionIntVar"].Value;

VB
Dim rowCnt As Integer = Dts.Variables("User::DecisionIntVar").Value
```


Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

Could easily be replaced with ordinal-based references like this:

```
C#
int rowCnt = (int)Dts.Variables[0].Value;

VB
Dim rowCnt As Integer = Dts.Variables(0).Value
```

The setting of variables uses the same syntax but reverses the assignment. The code that should be pasted into the `Main()` function of the `ScriptMain` class will evaluate the two variables and set the `HappyPathEnum` variable:

```
C#
//Retrieving the value of Variables
int rowCnt = (int)Dts.Variables["User::DecisionIntVar"].Value;
string fileExt = (string)Dts.Variables["User::DecisionStrVar"].Value;

if (fileExt.Equals("txt") && rowCnt > 0)
{
    Dts.Variables["User::HappyPathEnum"].Value = 1;
}
else if (fileExt.Equals("crd") && rowCnt > 0)
{
    Dts.Variables["User::HappyPathEnum"].Value = 2;
}
Dts.TaskResult = (int)ScriptResults.Success;

VB
'Retrieving the value of Variables
Dim rowCnt As Integer = Dts.Variables("User::DecisionIntVar").Value
Dim fileExt As String = Dts.Variables("User::DecisionStrVar").Value

If (fileExt.Equals("txt") And rowCnt > 0) Then
    Dts.Variables(2).Value = 1
    Dts.Variables("User::HappyPathEnum").Value = 1
ElseIf (fileExt.Equals("crd") And rowCnt > 0) Then
    Dts.Variables(2).Value = 2
    Dts.Variables("User::HappyPathEnum").Value = 2
End If
Dts.TaskResult = ScriptResults.Success
```

To alter the flow of the package, set the two precedence constraints in the package hierarchy to be based on a successful completion of the previous Script Task and an Expression that tests the expected values of the `HappyPathEnum` variable. This precedence defines that the Control Flow should only go in a direction if the value of an expression tests true. Set the precedence between each Script Task to one of these expressions:

```
@HappyPathEnum == 1
Or
@HappyPathEnum == 2
```

A sample of the precedence between the Script Task and the TXT File Script Task should look like Figure 9-13.

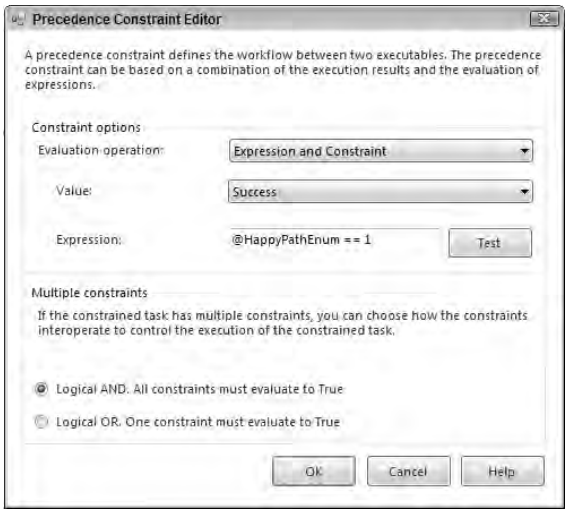

Figure 9-13

Now, to give the package something to do, simply retrieve the value of the set variable in each child Script Task to provide visual proof that the `HappyPathEnum` variable was properly set. Add this code into the `Main()` function of each child Script Task (make sure you set the message to display TXT or CRD for each associated Script Task):

```
C#
int ival = (int)Dts.Variables[0].Value;
string msg = string.Format("TXT File Found\nHappyPathEnum Value = {0}",
                   Dts.Variables[0].Value.ToString());

System.Windows.Forms.MessageBox.Show(msg);
Dts.TaskResult = (int)ScriptResults.Success;

VB
 Dim ival As Integer = Dts.Variables(0).Value
 Dim msg As String = _
 String.Format("TXT File Found" + vbCrLf + "HappyPathEnum Value = {0}",
                Dts.Variables(0).Value.ToString())

 System.Windows.Forms.MessageBox.Show(msg)
 Dts.TaskResult = ScriptResults.Success
```

To see how this works, set the value of the `User::DecisionIntVar` variable to a positive integer number value, and the `User::DecisionStrVar` variable to either `txt` or `crd`, and watch the package switch from one Control Flow to the other. If you provide a value other than `txt` or `crd` (even `"txt"` with quotes will cause this), the package will not run either leg, which is as designed. This is a simple example that you can refer back to as your packages get more complicated, and you are referring to or

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

updating variables within the Script Tasks. Later, we'll look at the Script Component that accesses variables in a slightly different way.

Connecting to Data Sources in a Script Task

A common use of an ActiveX Script Task in DTS packages was to grab a connection to various Data Sources for decision-making data from Excel files, INI files, flat files, or even databases like Oracle or Access. This capability provided ways to get to other Data Sources for configuring the packages, or to retrieve or output data for things we didn't have a direct connection object to use. In SSIS, with the Scripting Task, you can still make connections using any of the .NET libraries directly, or you can use connections that are defined in a package. Connections in SSIS are abstractions for connection strings that can be copied, passed around, and configured more easily than the ADO, script-based version in DTS.

The Connections collection hangs off of the DTS object in the Script Task. To retrieve a connection you call the `AcquireConnection` method on a specific named (or ordinal position) connection in the collection. The only thing you really should know ahead of time is what type of connection you are going to be retrieving, because you'll need to cast the returned connection to the proper connection type. In .NET, connections are not generic like the ADO model. Examples of concrete connections are `SqlConnection`, `OleDb.Connection`, `OdbcConnection`, and the `OracleConnection` Managers that connect using SqlClient, OLE DB, ODBC, and even Oracle data access libraries respectively. There are some things you can do to query the Connection Manager to determine what is in the connection string or if it supports transactions, but you shouldn't expect to use one connection in SSIS for everything, especially with the added Connection Managers for FTP, HTTP, and WMI.

Assuming that you're up to speed on the different types of connections covered earlier in this book, let's look at how you can use them in everyday SSIS package tasks.

Example: Retrieving Data into Variables from a Database

Although SSIS provides configurable abilities to set package-level values, there are use-cases that require you to retrieve actionable values from a database that can be used for package Control Flow or other functional purposes. One example would be some variable aspect of the application that may change, like an email address for events to use for notification. In this example, you'll retrieve a log file path for a package at runtime using a connection within a Script Task. The database that contains the settings for the log file path stores this data using the package ID. You'll first need a table in the AdventureWorks2008 database called `SSIS_SETTING`. Create the table with three fields, `PACKAGE_ID`, `SETTING`, and `VALUE`, or use this script:

```
CREATE TABLE [dbo].[SSIS_SETTING](
  [PACKAGE_ID] [uniqueidentifier] NOT NULL,
  [SETTING] [nvarchar](2080) NOT NULL,
  [VALUE] [nvarchar](2080) NOT NULL
) ON [PRIMARY]
GO
INSERT INTO SSIS_SETTING
SELECT '{INSERT YOUR PACKAGE ID HERE}', 'LOGFILEPATH', 'c:\myLogFile.txt'
```

Then create an SSIS package with one ADO.NET Connection Manager to the AdventureWorks database called `local.aw` and add a package-level variable named `LOGFILEPATH` of type `String`. Add a Script Task to the project and send in two variables: the ReadOnly `System::PackageID` and a ReadWrite

variable `User::LOGFILEPATH`. Click the Edit Script button to open the Script project and add the namespace to `System.Data.SqlClient` in the top of the class. Then add the following code to the `Main()` method:

```
C#
public void Main()
{
    string myPackageId = Dts.Variables["System::PackageID"].Value.ToString();
    string myValue = string.Empty;
    string cmdString = "SELECT VALUE FROM SSIS_SETTING " +
             "WHERE PACKAGE_ID= @PACKAGEID And SETTING= @SETTINGID";

    try
    {
        SqlConnection mySqlConn =
              (SqlConnection)Dts.Connections[0].AcquireConnection(null);
        mySqlConn = new SqlConnection(mySqlConn.ConnectionString);
        mySqlConn.Open();
        SqlCommand cmd = new SqlCommand();
        cmd.CommandText = cmdString;
        SqlParameter parm = new SqlParameter("@PACKAGEID",
                    SqlDbType.UniqueIdentifier);
        parm.Value = new Guid(myPackageId);
        cmd.Parameters.Add(parm);
        parm = new SqlParameter("@SETTINGID", SqlDbType.NVarChar);
        parm.Value = "LOGFILEPATH";
        cmd.Parameters.Add(parm);
        cmd.Connection = mySqlConn;
        cmd.CommandText = cmdString;
        SqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            myValue = reader["value"].ToString();
        }

        Dts.Variables["User::LOGFILEPATH"].Value = myValue;

        reader.Close();
        mySqlConn.Close();
        mySqlConn.Dispose();
    }
    catch
    {
        Dts.TaskResult = (int)ScriptResults.Failure;
        throw;
    }

    System.Windows.Forms.MessageBox.Show(myValue);
    Dts.TaskResult = (int)ScriptResults.Success;
}

VB
Public Sub Main()
    Dim myPackageId As String = _
```

```
                Dts.Variables("System::PackageID").Value.ToString()
    Dim myValue As String = String.Empty
    Dim cmdString As String = "SELECT VALUE FROM SSIS_SETTING " + _
                "WHERE PACKAGE_ID= @PACKAGEID And SETTING= @SETTINGID"
    Try
        Dim mySqlConn As SqlClient.SqlConnection
        mySqlConn = DirectCast(Dts.Connections(0).AcquireConnection(Nothing),
                        SqlClient.SqlConnection)
        mySqlConn = New SqlClient.SqlConnection(mySqlConn.ConnectionString)
        mySqlConn.Open()
        Dim cmd = New SqlClient.SqlCommand()
        cmd.CommandText = cmdString
        Dim parm As New SqlClient.SqlParameter("@PACKAGEID", _
                    SqlDbType.UniqueIdentifier)
        parm.Value = New Guid(myPackageId)
        cmd.Parameters.Add(parm)
        parm = New SqlClient.SqlParameter("@SETTINGID", SqlDbType.NVarChar)
        parm.Value = "LOGFILEPATH"
        cmd.Parameters.Add(parm)
        cmd.Connection = mySqlConn
        cmd.CommandText = cmdString
        Dim reader As SqlClient.SqlDataReader = cmd.ExecuteReader()
        Do While (reader.Read())
            myValue = reader("value").ToString()
        Loop
        Dts.Variables("User::LOGFILEPATH").Value = myValue
        reader.Close()
        mySqlConn.Close()
        mySqlConn.Dispose()
    Catch ex As Exception
        Dts.TaskResult = ScriptResults.Failure
        Throw
    End Try

    System.Windows.Forms.MessageBox.Show(myValue)
    Dts.TaskResult = ScriptResults.Success
End Sub
```

In this code, the package ID is passed into the Script Task as a read-only variable and is used to build a TSQL statement to retrieve the value of the `LOGFILEPATH` setting from the `SSIS_SETTING` table. The `AcquireConnection` method creates an instance of a connection to the local AdventureWorks database managed by the Connection Manager and allows other `SqlClient` objects to access the Data Source. The retrieved setting from the `SSIS_SETTING` table is then stored in the writable variable `LOGFILEPATH`. This is a basic example, but you use this exact same technique to retrieve a recordset into an object variable that can be iterated within your package as well.

Example: Saving Data to an XML File

Another common requirement is to generate data of a certain output format. When the output is a common format like Flat File, Excel, CSV, or other database format, you can simply pump the data stream into one of the Data Flow Destinations. If you want to save data to an XML file, the structure is not homogeneous, and not as easy to transform from a column-based data stream into an XML structure without some logic or structure around it. This is where the Script Task comes in handy.

The easiest way to get data into an XML file is to load and save the contents of a dataset using the method `WriteXML` on the dataset. With a new Script Task in a package with an ADO.NET connection to AdventureWorks2008, add a reference to the `System.Xml.dll`, then add the namespaces for `System.Data.SqlClient`, `System.IO`, and `System.Xml`. Then code the following into the Script Task to open a connection and get all the `SSIS_SETTING` rows and store as XML.

See the previous example for the DDL to create this table in the AdventureWorks2008 database.

```
C#
public void Main()
{
    SqlConnection sqlConn;
    string cmdString = "SELECT * FROM SSIS_SETTING ";
    try
    {
        sqlConn =
(SqlConnection)(Dts.Connections["local.aw"]).AcquireConnection(Dts.Transaction
);
        sqlConn = new SqlConnection(sqlConn.ConnectionString);
        sqlConn.Open();
        SqlCommand cmd = new SqlCommand(cmdString, sqlConn);
        SqlDataAdapter da = new SqlDataAdapter(cmd);
        DataSet ds = new DataSet();
        da.Fill(ds);
        ds.WriteXml(new
System.IO.StreamWriter("C:\\SSIS\\scripts\\ScriptDataIntoXMLFile\\
myPackageSettings.xml"));
        sqlConn.Close();
    }
    catch
    {
        Dts.TaskResult = (int)ScriptResults.Failure;
        throw;
    }
    Dts.TaskResult = (int)ScriptResults.Success;
}

VB
Public Sub Main()
    Dim sqlConn As New SqlConnection
    Dim cmdString As String = "SELECT * FROM SSIS_SETTING "
    Try
        sqlConn =
DirectCast(Dts.Connections("local.aw").AcquireConnection(Dts.Transaction),
SqlConnection)
        sqlConn = New SqlConnection(sqlConn.ConnectionString)
        sqlConn.Open()
        Dim cmd = New SqlCommand(cmdString, sqlConn)
        Dim da = New SqlDataAdapter(cmd)
        Dim ds = New DataSet
        da.Fill(ds)
        ds.WriteXml(New
StreamWriter("C:\\SSIS\\scripts\\ScriptDataIntoXMLFile\\myPackageSettings.xml"
))
```


Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

```
            sqlConn.Close()
        Catch
            Dts.TaskResult = ScriptResults.Failure
            Throw
        End Try
        Dts.TaskResult = ScriptResults.Success
    End Sub
```

The results are in XML, and there is not much to this file, except that it is in XML format.

```
<NewDataSet>
  <Table>
    <PACKAGE_ID>a5cf0c2f-8d85-42eb-91b9-cbd1fd47e5b1</PACKAGE_ID>
    <SETTING>LOGFILEPATH</SETTING>
    <VALUE>C:\SSIS\scripts\ScriptDataIntoVariable\myLogFile.txt</VALUE>
  </Table>
</NewDataSet>
```

If you need more control of the data you are exporting, or you need to serialize data, you'll need to use the Script Task in a different way. See the next example for some tips on how to do this.

Example: Serializing Data to XML

In the last example, you looked at simply dumping a recordset into an XML format by loading data into a dataset and using the `WriteToXML` method to dump the XML out to a file stream. If you need more control over the format or the data is hierarchical, using .NET XML object-based serialization can be helpful. Imagine implementations that pull data from flat-file mainframe dumps and fill fully hierarchical object models. Alternatively, imagine serializing data into an object structure to pop an entry into an MSMQ application queue. This is easy to do using some of the same concepts. Create another package with a connection to the AdventureWorks2008 database; add a Script Task with a reference to the `System.Data.SqlClient` namespace. Use the data from the previous example and create a class structure within your `ScriptMain` to hold the values for each row of settings that looks like this:

```
C#
[Serializable()]
public class SSISSetting
{
    public string PackageId { get; set; }
    public string Setting { get; set; }
    public string Value { get; set; }
}

VB
<Serializable()> Public Class SSISSetting
    Private m_PackageId As String
    Private m_Setting As String
    Private m_Value As String

    Public Property PackageId() As String
        Get
            PackageId = m_PackageId
        End Get
```

```
            Set(ByVal Value As String)
                m_PackageId = Value
            End Set
        End Property
        Public Property Setting() As String
            Get
                PackageId = m_Setting
            End Get
            Set(ByVal Value As String)
                m_Setting = Value
            End Set
        End Property
        Public Property Value() As String
            Get
                Value = m_Value
            End Get
            Set(ByVal Value As String)
                m_Value = Value
            End Set
        End Property
    End Class
```

This class will be used to fill based on the dataset like we had in the last example. It is still a flat model, but more complex class structures would have collections within the class. An example would be a student object with a collection of classes, or an invoice with a collection of line items. To persist this type of data you'll need to traverse multiple paths to fill the model. Once the model is filled, the rest is easy. First, add the namespaces `System.Xml.Serialization`, `System.Collections.Generic`, `System.IO`, and `System.Data.SqlClient` into your Script Task project. Then a simple example with the `SSIS_SETTING` table would look like this:

```
    C#
    public void Main()
    {
        SqlConnection sqlConn;
        string cmdString = "SELECT * FROM SSIS_SETTING ";

        try
        {
            sqlConn =
    (SqlConnection)(Dts
    .Connections["local.aw"]).AcquireConnection(Dts.Transaction);
            sqlConn = new SqlConnection(sqlConn.ConnectionString);
            sqlConn.Open();
            SqlCommand cmd = new SqlCommand(cmdString, sqlConn);
            SqlDataReader dR = cmd.ExecuteReader();
            List<SSISSetting> arrayListSettings = new List<SSISSetting>();

            while (dR.Read())
            {
                SSISSetting oSet = new SSISSetting();
                oSet.PackageId = dR["PACKAGE_ID"].ToString();
```


Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

```
            oSet.Setting = dR["SETTING"].ToString();
            oSet.Value = dR["VALUE"].ToString();
            arrayListSettings.Add(oSet);
        }

        StreamWriter outfile = new
StreamWriter("C:\\SSIS\\scripts\\ScriptDataintoSerializableObject\\myObjectXml
Settings.xml");

        XmlSerializer ser = new XmlSerializer(typeof(List<SSISSetting>));
        ser.Serialize(outfile, arrayListSettings);
        outfile.Close();
        outfile.Dispose();
        sqlConn.Close();
    }
    catch
    {
        Dts.TaskResult = (int)ScriptResults.Failure;
        throw;
    }
    Dts.TaskResult = (int)ScriptResults.Success;
}

VB
Public Sub Main()
    Dim sqlConn As SqlConnection
    Dim cmdString As String = "SELECT * FROM SSIS_SETTING "

    Try
        sqlConn =
DirectCast(Dts.Connections("local.aw").AcquireConnection(Dts.Transaction),
           SqlConnection)
        sqlConn = New SqlConnection(sqlConn.ConnectionString)
        sqlConn.Open()
        Dim cmd As SqlCommand = New SqlCommand(cmdString, sqlConn)
        Dim dR As SqlDataReader = cmd.ExecuteReader()
        Dim arrayListSettings As New List(Of SSISSetting)
        Do While (dR.Read())
            Dim oSet As New SSISSetting()

            oSet.PackageId = dR("PACKAGE_ID").ToString()
            oSet.Setting = dR("PACKAGE_ID").ToString()
            oSet.Value = dR("PACKAGE_ID").ToString()
            arrayListSettings.Add(oSet)
        Loop

        Dim outfile As New
StreamWriter("C:\\SSIS\\scripts\\ScriptDataintoSerializableObject\\myObjectXml
Settings.xml")
        Dim ser As New XmlSerializer(GetType(List(Of SSISSetting)))

        ser.Serialize(outfile, arrayListSettings)
        outfile.Close()
        outfile.Dispose()
```

```
            sqlConn.Close()
        Catch
            Dts.TaskResult = ScriptResults.Failure
            Throw
        End Try

        Dts.TaskResult = ScriptResults.Success
    End Sub
```

The `StreamWriter` here just gets an IO stream from the file system to use for data output. The `XmlSerializer` does the heavy lifting and converts the data from the object format into an XML format. The only trick here is to understand how to deal with the `Generic List` or the collection of all the `SSISSetting` objects. This is handled by using the override where you can add the specific types to the serializer along with the `List`. The resulting XML payload will now look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfSSISSetting xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SSISSetting>
    <PackageId>34050406-2e0f-423a-8af3-1ec95399a6c2</PackageId>
    <Setting>34050406-2e0f-423a-8af3-1ec95399a6c2</Setting>
    <Value>34050406-2e0f-423a-8af3-1ec95399a6c2</Value>
  </SSISSetting>
</ArrayOfSSISSetting>
```

Although the XML content looks a little bit different than dumping the content of the recordset directly to XML as we did in the earlier example, it is optimized for object serialization. This is the type of content that you could push into application queues or share with external applications.

Raising an Event in a Script Task

All existing SSIS Tasks and Components raise events that can be captured and displayed by the Execution Results tab by default. Optionally these events can also be captured and logged into SSIS logging or event handlers. Event handlers are Control Flows that you set up and define to respond to specific events. They are literally Control Flow workflows within a package and give you the advantage of customizing the diagnostic information that the packages can provide at runtime.

If you have done any Windows GUI programming, you will be familiar with events. An *event* is simply a message sent from some object saying that something just happened or is about to happen. To raise or *fire* an event with a Script Task, you use the `Events` property of the `Dts` object. The `Events` property on the `Dts` object is really an instance of the `IDTSComponentEvents` interface. This interface specifies seven methods for firing events:

- `FireBreakpointHit`: This method supports the SQL Server infrastructure and is not intended to be used directly in code.
- `FireError`: Fires an event when an error occurs.
- `FireInformation`: Fires an event with information. You can fire this event when you want some set of information to be logged, possibly for auditing later.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

- `FireProgress`: Fires an event when a certain progress level has been met.
- `FireQueryCancel`: Fires an event to determine if package execution should stop.
- `FireWarning`: Fires an event that is less serious than an error, but more than just information.
- `FireCustomEvent`: Fires a custom defined event.

In SSIS, any events you fire will be written to all enabled log handlers that are set to log that event. Logging allows you to see what happened with your script when you're not there to watch it run. This is useful for troubleshooting and auditing purposes, as you'll see in the following example.

Example: Raise Some Events

The default event handler at design time is the Execution Results tab at the top of your package in the BIDS design environment. The simplest way to use the events is to fire off some sample events and see them in this Execution Results tab. To do this create a new package with a Script Task and add the System variable `System::TaskName` as a Read-Only variable. Then add the following code to the `Main()` function:

```
C#
public void Main()
{
    string taskName =
       Dts.Variables["System::TaskName"].Value.ToString();
    bool retVal = false;

    Dts.Events.FireInformation(0, taskName,
        String.Format("Starting Loop Operation at {0} ",
        DateTime.Now.ToString("MM/dd/yyyy hh:mm:ss")), "", 0,
        ref retVal);

    for(int i=0; i <= 10; i++)
    {
        Dts.Events.FireProgress(String.Format("Loop in iteration {0}", i),
                i * 10, 0, 10, taskName, ref retVal);
    }

    Dts.Events.FireInformation(0, taskName,
        String.Format("Completion Loop Operation at {0} ",
        DateTime.Now.ToString("mm/dd/yyyy hh:mm:ss")), "", 0,
        ref retVal);

    Dts.Events.FireWarning(1, taskName,
        "This is a warning we want to pay attention to...",
        "", 0);
    Dts.Events.FireWarning(2, taskName,
        "This is a warning for debugging only...",
```

```
            "", 0);

        Dts.Events.FireError(0, taskName,
            "If we had an error it would be here", "", 0);
    }

     VB
    Public Sub Main()
        Dim i As Integer = 0
        Dim taskName As String =
               Dts.Variables("System::TaskName").Value.ToString()
        Dim retVal As Boolean = False

        Dts.Events.FireInformation(0, taskName, _
              String.Format("Starting Loop Operation at {0} ", _
              DateTime.Now.ToString("MM/dd/yyyy hh:mm:ss")), "", 0, _
              True)

        For i = 0 To 10
            Dts.Events.FireProgress( _
                String.Format("Loop in iteration {0}", i), _
                i * 10, 0, 10, taskName, True)
        Next

        Dts.Events.FireInformation(0, taskName, _
             String.Format("Completion Loop Operation at {0} ", _
             DateTime.Now.ToString("mm/dd/yyyy hh:mm:ss")), "", 0, False)

        Dts.Events.FireWarning(1, taskName, _
            "This is a warning we want to pay attention to...", _
            "", 0)
        Dts.Events.FireWarning(2, taskName, _
            "This is a warning for debugging only...", _
            "", 0)

        Dts.Events.FireError(0, taskName, _
            "If we had an error it would be here", "", 0)
    End Sub
```

This code will perform a simple loop operation and demonstrate firing the information, progress, warning, and error events. If you run the package, you'll be able to view the information embedded in these fire event statements in the Execution Results tab in Figure 9-14. Note that raising the error event results in the Script Task failure. You may comment out the `FireError` event to see the task complete successfully.

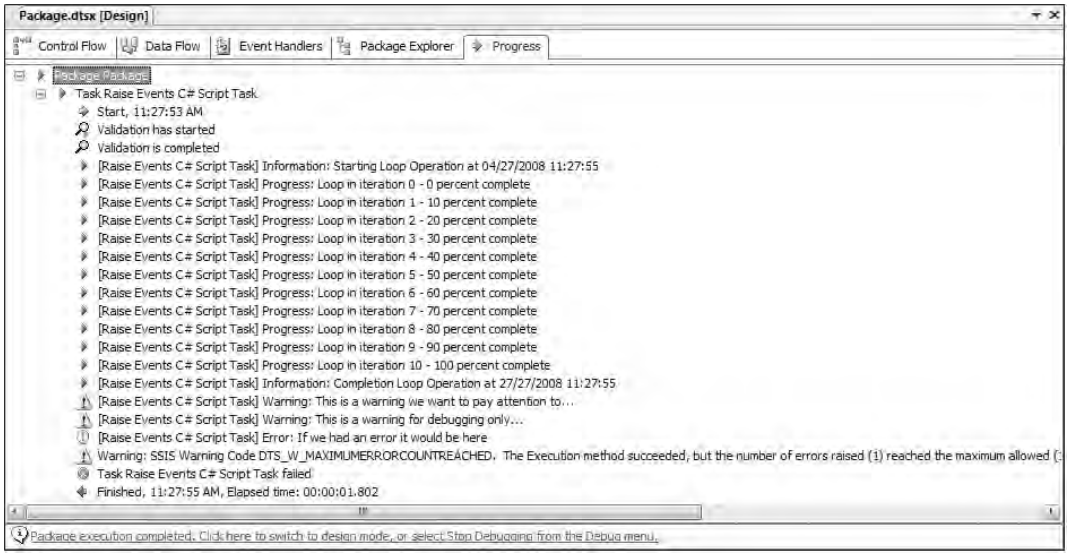

Figure 9-14

All the statements prefixed with the string `[Script Task]` were generated using these events fired from the Script Task. We'll leave it with you to comment out the `Dts.Events.FireError` method calls to demonstrate to yourself that the task can complete successfully for warnings and informational events. Note that with the firing of an error you can also force the task to generate a custom error with an error code and description. In fact, each of the events has a placeholder as the first parameter to store a custom information code. Continue to the next example to see how you can create an error handler to respond to the warning events that are fired from this Script Task.

Example: Respond to an Event

If you have already created a package for the Raise Some Events example, navigate to the Event Handlers tab. Event handlers are separate Control Flows that can be executed in response to an event. In the Raise Some Events example, you generated two Warning events. One had an information code of one (1) and the other had the value of two (2). You are going to add an event handler to respond to those warning events and add some logic to respond to the event if the information code is equal to one (1). Select the executable of Script Task and select the event handler of `OnWarning`. Then click the hot link that states:

```
Click here to create an 'OnWarning' event handler for executable 'Script Task'
```

This will create a Control Flow surface where you can drop SSIS Control Tasks onto the surface that will execute if an `OnWarning` event is thrown from the Script Task you added to the package earlier. Drop a new Script Task into the Event Handler Control Flow surface and name it OnWarning Script Task. Your designer should look like Figure 9-15.

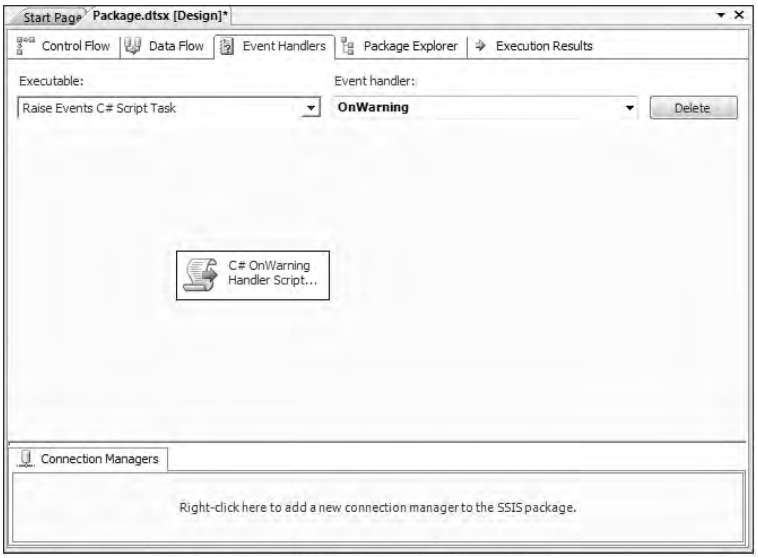

Figure 9-15

To retrieve the information code sent in the `Dts.Events.FireWarning` method call, add two system-level variables `System::ErrorCode,System::ErrorDescription` to the Read-Only Variables collection of the OnWarning Script Task. These variables will contain the values of the `InformationCode` and `Description` parameters in the `Dts.Events()` methods. You can then retrieve and evaluate these values when an event is raised by adding the following code:

```
C#
        long lWarningCode = long.Parse(Dts.Variables[0].Value.ToString());
        String sMsg = string.Empty;
        if(lWarningCode == 1)
        {
            sMsg = String.Format(
               "Would do something with this warning:\n{0}: {1}",
               lWarningCode.ToString(), Dts.Variables(1).ToString());
            System.Windows.Forms.MessageBox.Show(sMsg);
        }
        Dts.TaskResult = (int)ScriptResults.Success;

VB
        Dim lWarningCode As Long = _
               Long.Parse(Dts.Variables(0).Value.ToString())
        Dim sMsg As String
        If lWarningCode = 1 Then
            sMsg = String.Format("Would do something with this warning: " _
                  + vbCrLf + "{0}: {1}", _
            lWarningCode.ToString(), Dts.Variables(1).ToString())
            System.Windows.Forms.MessageBox.Show(sMsg)
        End If
        Dts.TaskResult = ScriptResults.Success
```


Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

The code checks the value of the first parameter, which is the value of the `System::ErrorCode` and the value raised in the `Dts.Events.FireWarning` method. If the value is equivalent to one (1), an action is taken to show a message box. This action could just as well be logging an entry to a database, or sending an email. If you rerun the package now, you'll see that the first `FireWarning` event will be handled in your event handler and generate a message box warning. The second `FireWarning` event will also be captured by the event handler, but no response is made. You can see the event handler counter in the Progress or Execution Results tab is incremented to two (2). Raising events in the Script Tasks are great ways to get good diagnostic information without resorting to MessageBoxes in your packages. See Chapter 17 for much more detail about this type of development in SSIS.

Example: Logging Event Information

Scripts can also be used to log custom event information. To configure the previous example events SSIS package to log event information, go to SSIS ⇨ Logging in the Business Intelligence Designer Studio. The Configure SSIS Logs dialog will appear. Select "SSIS log provider for XML files" in the Provider Type drop-down and click Add. Click the column named Configuration and then select <New Connection> from the list to create an XML File Editor. For Usage type, select Create File and specify a path to a filename similar to `c:\ssis\scripts\raisingevents\myLogFile.xml`. (This would be something you'd use an expression or package configuration to set during runtime.) Click OK to close the File Connection Manager Editor dialog box. Your screen should look something like Figure 9-16.

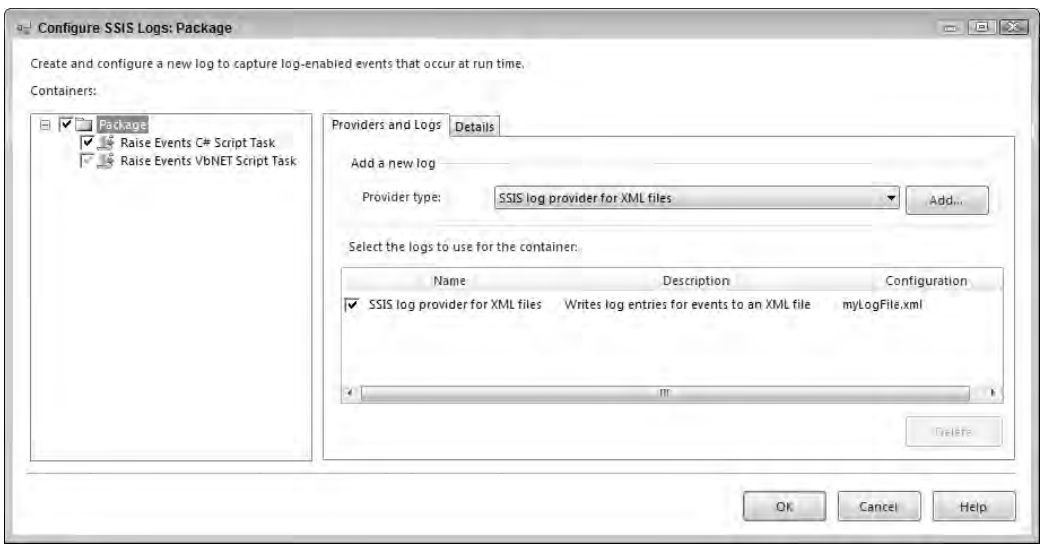

Figure 9-16

Now click the Package Node to start selecting what tasks in the package should log to the new provider, and check the box next to the provider name so that the log will be used. In the Details tab, select the `OnWarning` events specifically to log. You can choose to log any of the available event types to the providers by also selecting them in the Details tab. Now your provider configuration should look like Figure 9-17.

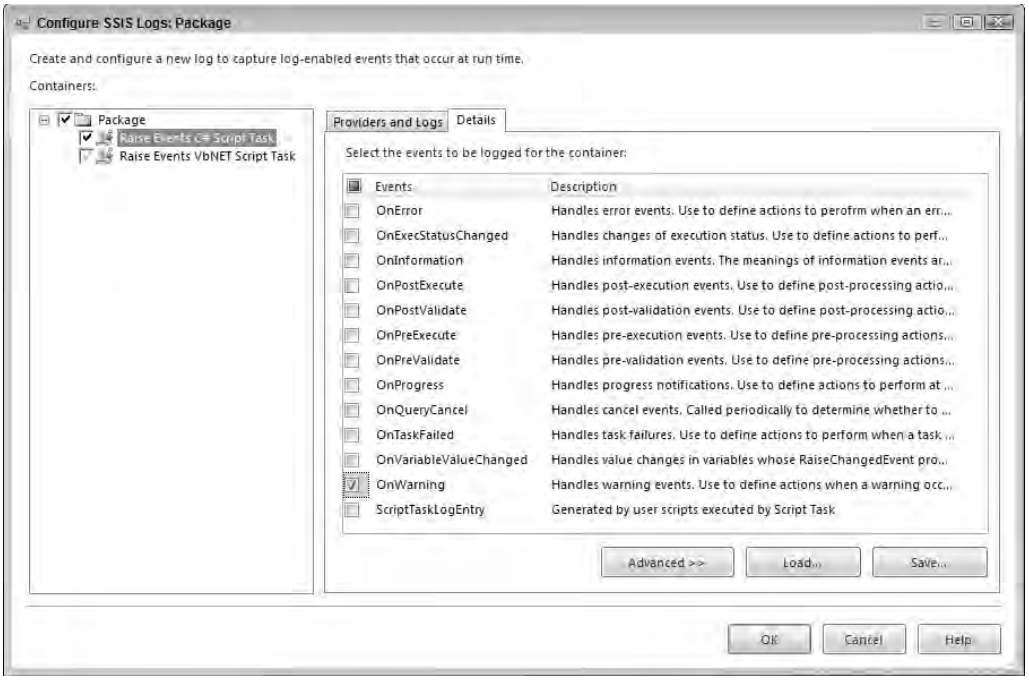

Figure 9-17

You can also go to the Advanced tab for each selected event to control exactly what properties of the event get logged as well. If you run the package again, the file specified in the logging provider will be created with content similar to the following:

```
<record>
      <event>OnWarning</event>
      <message>This is a warning we want to pay attention to...</message>
      <computer>MYCOMPUTER</computer>
      <operator>MYCOMPUTER\ADMIN</operator>
      <source>Package</source>
      <sourceid>{D86FF397-6C9B-4AD9-BACF-B4D41AC89ECB}</sourceid>
      <executionid>{8B6F6392-1818-4EE5-87BF-EDCB5DC37ACB}</executionid>
      <starttime>1/22/2008 9:30:08 PM</starttime>
      <endtime>1/22/2008 9:30:08 PM</endtime>
      <datacode>2</datacode>
      <databytes>0x</databytes>
</record>
```

You'll have other events in the file, such as Package Start and Package End, but the preceding code snippet focuses on the event that your code fired. This record contains the basic information on the event including the message, event execution time, and the computer and user that raised the event.

Using the Script Task to raise an event is just one way to get more diagnostic information into your SSIS log files. Read on to get a brief look at generating simple log entries.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

Writing a Log Entry in a Script Task

Within a Script Task, the `Log` method of the `Dts` object writes a message to all enabled log providers. The `Log` method is simple and has but three arguments:

- `messageText`: The message to log
- `dataCode`: A field for logging a message code
- `dataBytes`: A field for logging binary data

The `Log` method is similar to the `FireInformation` method of the `Events` property, but it is easier to use and more efficient — you also do not need to create a specialized event handler to respond to the method call. All you need to do is set up a log provider within the package. In the previous section, you learned about how to add a log provider to a package. The following code logs a simple message with some binary data to all available log providers. This is quite useful for troubleshooting and auditing purposes. You can write out information at important steps in your script and even print out variable values to help you track down a problem.

Example: Script a Log Entry

You can see an example of how to script a log entry by adding a few lines of code to the package in the previous examples that you used to raise events. First add these lines to the appropriate Script Task that matches the language you chose in the previous example:

```
C#
Byte[] myByteArray[] = new byte[0];
Dts.Log("Called procedure: usp_Upsert with return code 4", 0, myByteArray);

VB
Dim myByteArray(0) As Byte
Dts.Log("Called procedure: usp_Upsert with return code 4", 0, myByteArray)
```

Then, select the events for the ScriptTaskLogEntry event in the Details tab of the logging configuration. This tells the SSIS package logger to expect to log any custom logging instructions like you just coded. Then run the package. You'll see a set of additional logging instructions that look like this:

```
<record>
   <event>User:ScriptTaskLogEntry</event>
   <message>Called Procedure: usp_Upsert with return code 4</message>
   <computer>MYCOMPUTER</computer>
   <operator>MYCOMPUTER\ADMIN</operator>
   <source>Raise Events C# Script Task</source>
   <sourceid>{CE53C1BB-7757-47FF-B173-E6088DA0A2A3}</sourceid>
   <executionid>{B7828A35-C236-451E-99DE-F679CF808D91}</executionid>
   <starttime>4/27/2008 2:54:04 PM</starttime>
   <endtime>4/27/2008 2:54:04 PM</endtime>
   <datacode>0</datacode>
   <databytes>0x</databytes>
</record>
```

As you can see, the Script Task is highly flexible with the introduction of the .NET-based VSTA capabilities. As far as controlling package flow or one-off activities, the Script Task has clearly taken over

the role of the DTS ActiveX Script Component. However, the Script Task doesn't do all things well. If you want to apply programmatic logic to data in the data pump portion or Data Flow in an SSIS package, then you need to continue and add to your knowledge of scripting in SSIS and the Script Component.

Using the Script Component

The Script Component provides another area where programming logic can be applied in an SSIS package. This component can only be used in the Data Flow portion of an SSIS package and allows programmatic tasks to occur in the data stream. Anything you can do in .NET at a stream level can be done in this task. Connect to an HTTP Source to create a stream, parse through an existing stream, or send a stream to a custom destination. This component exists to provide, consume, or transform data using .NET code. To differentiate the different uses of the Script Component, when you create one you have to choose one of the following three types:

- **Source Type Component:** The role of this Script Component is to provide data to your Data Flow Task. You can define outputs and their types and use script code to populate them. An example would be reading in a complex file format, possibly XML or something that requires custom coding to read, like HTTP or RSS Sources.
- **Destination Type Component:** This type of Script Component consumes data much like an Excel or Flat File Destination. This component is the end of the line for the data in your data pump or stream. Here you'll typically put the data into a dataset variable to pass back to the Control Flow for further processing or send the stream to custom output destinations not supported in SSIS by a control. Examples of these output destinations can be Web service calls, custom XML formats, and multi-record formats for mainframe systems. You can even programmatically connect and send a stream to a printer object.
- **Transformation Type Component:** This type of Script Component can perform custom transformations on data. It consumes input columns and produces output columns. You would use the component when one of the built-in transformations just isn't flexible enough.

In this section, you'll get up to speed on all the specifics of the Script Component, starting first with an explanation of the differences between the Script Task and Component and then looking at the coding differences in the two models. In the end, you'll get an example of each implementation type of the Script Component to put all of this information to use.

Differences from a Script Task

You might ask, "Why are there two controls for the Script Task and Script Component?" Well, underneath the SSIS architecture there are really two different implementations of how the VSTA environment is used for performance. The Script Task is only going to be called once within a Control Flow, unless it is in a looping control. The Script Component has to be higher octane because it is going to be called per row of data in the data stream. You are also in the context of being able to access the data buffers directly, so there is a slight overhead in learning the differences between these two tasks.

When you are working with these two controls, the bottom line for you is that there are slightly different ways of doing the same types of things in each. This section of the chapter cycles back through some of the things you did with the Script Task and points out the differences. First, let's look at the differences in configuring the editor, then what changes when performing programmatic tasks such as accessing variables, using connections, raising events, and logging. Finally, we'll look at an example from an overall perspective.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

Configuring the Script Component Editor

You'll notice the differences starting with the task editor. Adding a Script Component to the Data Flow designer brings up the editor shown in Figure 9-18, requesting the component type.

You must first add a Data Flow Task to a package to be able to add the Script Component.

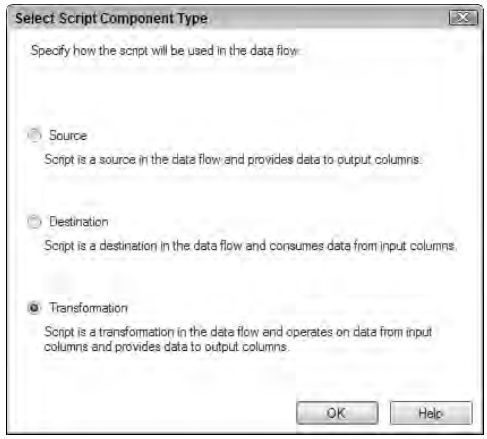

Figure 9-18

Selecting one of these choices changes the way the editor displays to configure the control. Essentially, you are choosing whether the control has input buffers, output buffers, or both. Figure 9-19 shows an example of the transformation Script Control that has both buffers.

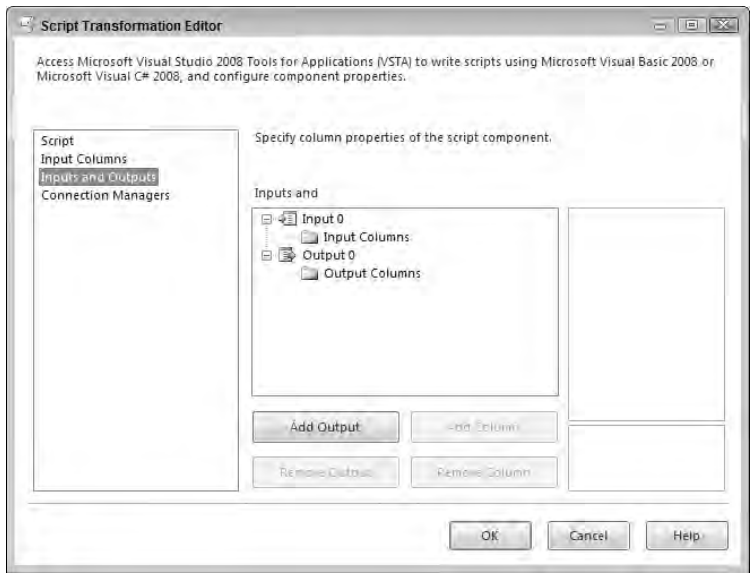

Figure 9-19

In the source version of the Script Component, the input buffers would not be available; the opposite is true of the destination version. You are responsible for defining these buffers by providing the set of typed columns for either the input or outputs. If the data is being fed into the component, the editor can reflect on the stream and set these up for you. Otherwise, you'll have to define them yourself. You can do this programmatically in the code, or ahead of time using the editor. Just select the input or output columns collection on the UI, and click the Add Column button to add a column as shown in Figure 9-20.

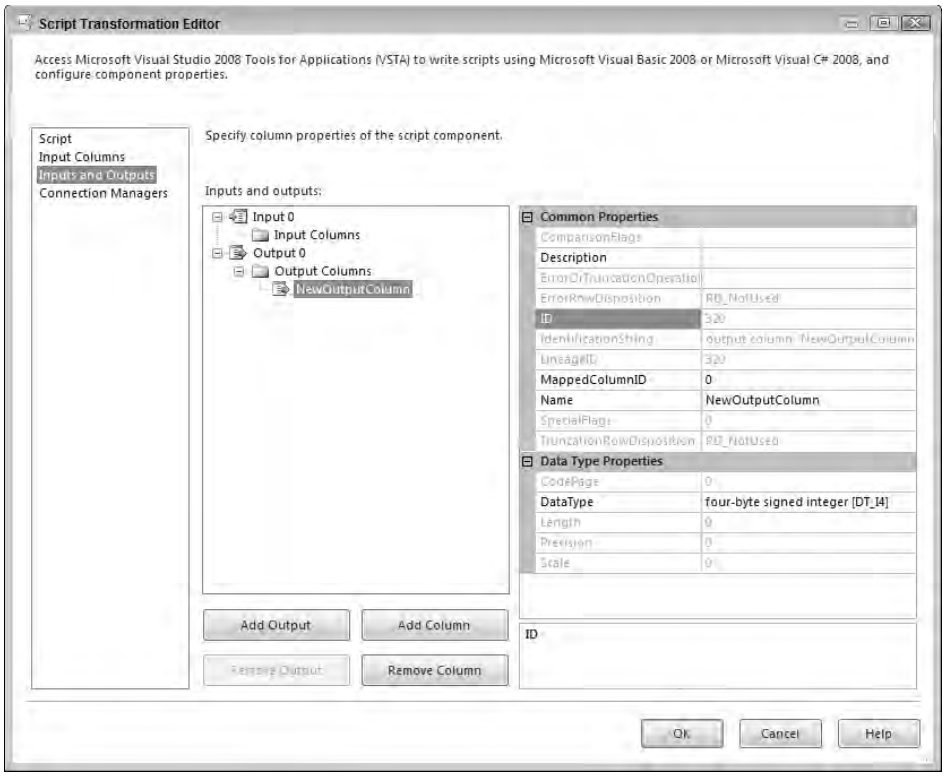

Figure 9-20

A helpful tip is to select the Output Columns Node on the tree view, so that the new column gets added to the bottom of the collection. Once you add a column, you can't move it up or down. Once you add the column, you'll need to set the Data Type, Length, Precision, and Scale. For details about the SSIS data types, see Chapter 6.

When you access the scripting environment, you'll notice some additional differences between the Script Component and the Script Task. Namely, that there are some new classes added to the Project Explorer, as seen in Figure 9-21.

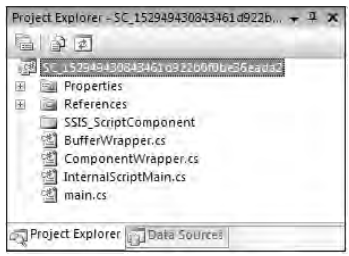

Figure 9-21

The class that is used to host custom code is named differently from the Script Task. The class name changed from `ScriptMain` to simply being called `main`. Internally there are also some changes. The main change is that there is not only one entry point method like there was in the Script Task. The methods you'll see in the `main` class depend upon the Script Component type. At least three of the following methods are typically coded and can be used as entry points in the Script Component:

- `PreExecute` is used for preprocessing tasks like creating expensive connections or file streams.
- `PostExecute` is used for cleanup tasks or setting variables at the completion of each processed row.
- `CreateNewOutputRows` is the method to manage the output buffers.
- `Input0 _ProcessInputRow` is the method to manage anything coming from the input buffers.

The remaining classes are generated automatically based on your input and output columns when you enter into the script environment, so don't make any changes to these, or they will be overwritten when you reenter the script environment.

One inconsistency that can occur within the Script Component Editor and the generation of the `BufferWrapper` class is that you can name columns in the editor that use keywords or are otherwise invalid when the `BufferWrapper` class is generated. An example would be an output column named `125K_AMOUNT`. If you create such a column, you'll get an error in the `BufferWrapper` class stating:

```
Invalid Token 125 in class, struct, or interface member declaration
```

Don't be tempted to change the property in the buffer class to something like `_125K_AMOUNT`, because this property will be rebuilt the next time you edit the script. Change the name of the output column to `_125K_AMOUNT`, and the buffer class will change automatically. The biggest difference that you need to pay attention to with the Script Component is that if you make *any* changes to this editor, you'll need to open up the Script environment so that all these base classes can be generated.

Last, but not least, you'll notice a Connections Managers tab that was not available in the Script Task Editor. This allows you to name specifically the connections that you want to be able to access within the Script Component. Although it is not required that you name these connections up front, it is extremely helpful if you do. You'll see why later, when you connect to a Data Source. Figure 9-22 shows an example of an Oracle connection added to a Script Component.

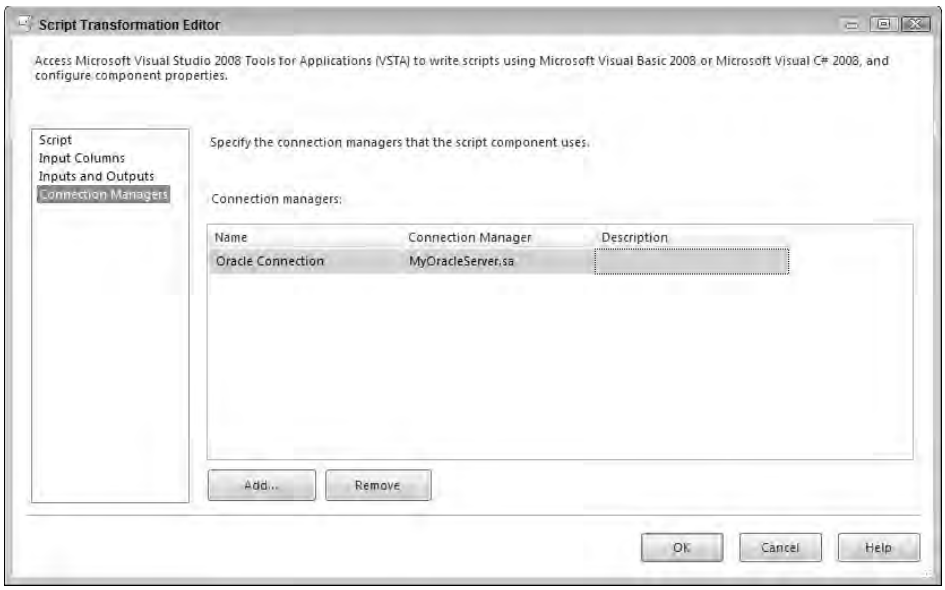

Figure 9-22

Now that you understand the differences in the Script Task and Component from a setup perspective, we'll look at how the coding is a little different. We'll start first with how you'll need to access package variables.

Accessing Variables in a Script Component

The same concepts behind accessing variables apply to the Script Component. You can either send the variables in to the control by adding them to the `ReadOnlyVariables` or `ReadWriteVariables` properties of the editor. You can also choose not to specify them up front and just use the variable dispenser within your Script Task to access, lock, and manipulate variables within the Script Component. We'd recommend that you use the properties in the editor for this component. The reason why is that the variables provided in the editor are added to the auto-generated base class variables collection as strongly typed variables. In this control, adding variables to the editor not only removes the need to lock and unlock the variables, but you get the added benefit of not having to remember the variable name within the component. Here's an example of setting the variable `VALIDATION_ERRORS` within a Script Component:

```
C#
this.Variables.VALIDATION_ERRORS = 1;
VB
me.Variables.VALIDATION_ERRORS = 1
```

As you can see this is much easier to use because the variable names are available in IntelliSense, and this is more maintainable because of the checking at compile time. However, if you have instances where you don't want to have to add a variable to each Script Component task, you can still use the variable dispenser in this component. It is located on the base class and can be accessed using the base class and

not the DTS object. Other than these changes, the variable examples in the Script Task section of this chapter are still applicable. The remaining tasks of connecting to Data Sources, raising events, and logging will follow similar patterns. The methods of performing the tasks are more strongly named and this makes sense because we don't need any late-binding (or runtime type checking) within a high-performing Data Stream Task.

Connecting to Data Sources in a Script Component

Typically, you'll see connections being used in Source types of the Script Component, because in these types of Data Flow Tasks, the mission is to create data stream. The origination of that data is usually another external source. If you had a defined SSIS Source Component, it would be used and you wouldn't need the Script Component to connect to it.

In the latest version of SSIS, the coding has been greatly simplified. You can instantiate a specific Connection Manager and simply assign it the reference to a connection in the component's collection. Using the connections collection in the Script Component is very similar to the variables collection. The collection of strongly typed Connection Managers is created every time the script editor is opened. Again, this is helpful because you don't have to remember the names, and you get compile-time verification and checking. If you have a package with an OLE DB Connection Manager named `myOracleServer` and add it to the Script Component with the name `OracleConnection`, you'll have access to the connection using this code:

```
C#
ConnectionManagerOleDb oracleConnection =
      (ConnectionManagerOleDb)base.Connections.OracleConnection;

VB
Dim oracleConn as ConnectionManagerOleDb
oracleConn = Connections.OracleConnection
```

Raising Events

For the Script Task, we've already looked at the ability of SSIS to raise events and demonstrated with examples, scripting capabilities that manage how the package can respond to these events. These same capabilities exist in the Script Components, although you do need to consider that Script Components run in a data pipeline or stream, so the potential for repeated calls is highly likely. You should fire events sparingly within a Script Component that is generating or processing data in the pipeline to reduce overhead and increase performance. The methods are essentially the same, but without the static DTS object. Here is the code to raise an informational event in a Script Component:

```
C#
Boolean myBool=false;
this.ComponentMetaData.FireInformation(0, "myScriptComponent",
                "Removed non-ASCII Character", "", 0, ref myBool);
VB
Dim myBool As Boolean
Me.ComponentMetaData.FireInformation(0, "myScriptComponent", _
                "Removed non-ASCII Character", "", 0, myBool)
```

Either version of code will generate an event in the Progress Tab that looks like this:

```
[myScriptComponent] Information: Removed non-ASCII Character
```

Raising an event is preferred to logging because of the ability to develop a separate workflow for handling the event, but there are some instances when logging may be preferred. We'll look into logging for the Script Component in the next section.

Logging

Like the Script Task, the logging in the Script Component writes a message to all enabled log providers. It has the same interface as the Script Task, but it is exposed on the base class. Remember that Script Components run in a data pipeline or stream, so the potential for repeated calls is highly likely. Follow the same rules as with raising events and log sparingly within a Script Component that is generating or processing data in the pipeline to reduce overhead and increase performance. If you need to log a message within a Data Flow, you can improve performance by logging only in the `PostExecute` method, so that the results are only logged once.

Example: Script a Log Entry

This example shows how to log one informational entry to the log file providers at the end of a Data Flow Task. To use this code create a package with a Data Flow Task and add a Script Component as a source with one output column named NewOutputColumn. Create these integer variables as private variables to the `main.cs` class: `validationBadChars`, `validationLength`, and `validationInvalidFormat`. Then add this code to the `CreateNewOutputRows()` method in the `main.cs` class:

```
C#
    int validationLengthErrors = 0;
    int validationCharErrors = 0;
    int validationFormatErrors = 0;

    //..in the CreateNewOutputRows() Method
    string validationMsg =
    string.Format("Validation Errors:\nBad Chars {0}\nInvalid Length " +
                  "{1}\nInvalid Format {2}",
                  validationCharErrors, validationLengthErrors,
                  validationFormatErrors);
    this.Log(validationMsg, 0, new byte[0]);

    //This is how to add rows to the outputrows Output0Buffer collection.
    Output0Buffer.AddRow();
    Output0Buffer.AddNewOutputColumn = 1;

VB
    Dim validationLengthErrors As Integer = 0
    Dim validationCharErrors As Integer = 0
    Dim validationFormatErrors As Integer = 0

    '..in the CreateNewOutputRows() Method
    Dim validationMsg As String
    validationMsg = String.Format("Validation Errors:" + _
```


Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

```
            vbCrLf + "Bad Chars {0}" + _
            vbCrLf + "Invalid Length {1}" + _
            vbCrLf + "Invalid Format {2}", _
            validationCharErrors, validationLengthErrors, _
            validationFormatErrors)
    Dim myByteArray(0) As Byte
    Me.Log(validationMsg, 0, myByteArray)
    Output0Buffer.AddRow()
    Output0Buffer.AddNewOutputColumn = 1
```

For this sample to produce a log entry, remember you will have to set up a logging provider using menu option SSIS ⇨ Logging. Make sure you specifically select the Data Flow Task in which the Script Component is hosted within SSIS and the logging events specifically for the Script Component. Running the package will produce logging similar to this:

```
User:ScriptComponentLogEntry,MYPC,MYPC\ADMIN,"CSharp Basic Logging Script
Component" (1),{00000001-0000-0000-0000-000000000000},{3651D743-D7F6-43F8-
8DE2-F7B40423CC28},4/27/2008 10:38:56 PM,4/27/2008 10:38:56 PM,0,0x,
Validation Errors:
Bad Chars 0
Invalid Length  0
Invalid Format 0
OnPipelinePostPrimeOutput, MYPC,MYPC\ADMIN,Data Flow Task,{D2118DFD-DAEE-470B-
9AC3-9B01DFAA993E},{3651D743-D7F6-43F8-8DE2-F7B40423CC28},4/27/2008 10:38:55
PM,4/27/2008 10:38:55 PM,0,0x,A component has returned from its PrimeOutput
call. : 1 : CSharp Basic Logging Script Component
```

Example: Data Validation

The Script Component takes a little longer runway to get up to speed on how to use it and to get a handle on how it is different from the Script Task. Now it's time to take a look at a more comprehensive example and get the bigger picture of how this component can be used in your everyday package development.

A typical use of the Script Component is to validate data within a Data Flow. In this example, the data is contact information from a custom application that did not validate data entry, so assume the data quality is poor. However, the destination database has a strict set of requirements for the data. Your task is to validate the contact data from the Flat File Source and separate valid from invalid records into two streams: the good stream and the error stream. The good records can continue to another Data Flow; the questionable records will be sent to an error table for manual cleansing.

Create the contacts table with the following script:

```
CREATE TABLE [dbo].[Contacts](
  [ContactID] [int] IDENTITY(1,1) NOT NULL,
  [FirstName] [varchar](50) NOT NULL,
  [LastName] [varchar](50) NOT NULL,
  [City] [varchar](25) NOT NULL,
  [State] [varchar](15) NOT NULL,
  [Zip] [char](11) NULL
) ON [PRIMARY]
```

The error queue table is virtually identical except that here there are no strict requirements and we've added a column to capture the rejection reason. All data fields are nullable and set to the maximum known size:

```
CREATE TABLE dbo.ContactsErrorQueue
(
  ContactErrorID int NOT NULL IDENTITY (1, 1),
  FirstName varchar(50) NULL,
  LastName varchar(50) NULL,
  City varchar(50) NULL,
  State varchar(50) NULL,
  Zip varchar(50) NULL,
  RejectReason varchar(50) NULL
)  ON [PRIMARY]
```

Finally, the incoming data format is fixed-width and is defined as follows:

Field	Starting Position	New Field Name
First Name	1	FirstName
Last Name	11	LastName
City	26	City
State	44	State
Zip	52	Zip

The data file provided as a test sample looks like this:

```
Jason      Gerard         Jacksonville      FL          32276-1911
Joseph     McClung        JACKSONVILLE      FLORIDA     322763939
Andrei     Ranga          Jax               fl          32276
Chad       Crisostomo     Orlando           FL          32746
Andrew     Ranger         Jax               fl
```

Create a sample of this data file or download a copy from `www.wrox.com`. Create a new package and add a Data Flow Task. Click on the Data Flow design surface and add a Connection Manager to the Connection Managers tab. Name the Connection Manager "Contacts Mainframe Extract," browse to the data file, and set the file format to Ragged Right. Flat files with spaces at the end of the specifications are typically difficult to process in some ETL platforms. The Ragged Right option in SSIS provides a way to handle these easily without having to run the file through a Script Task to put a character into a consistent spot, or having the origination system reformat their extract files. Use the Columns tab to visually define the columns. Flip to the Advanced tab to define each of the column names, types, and widths to match the copy book data definition and the new database field name. (You may need to delete an unused column if this is added by the designer.) The designer at this point looks like Figure 9-23.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

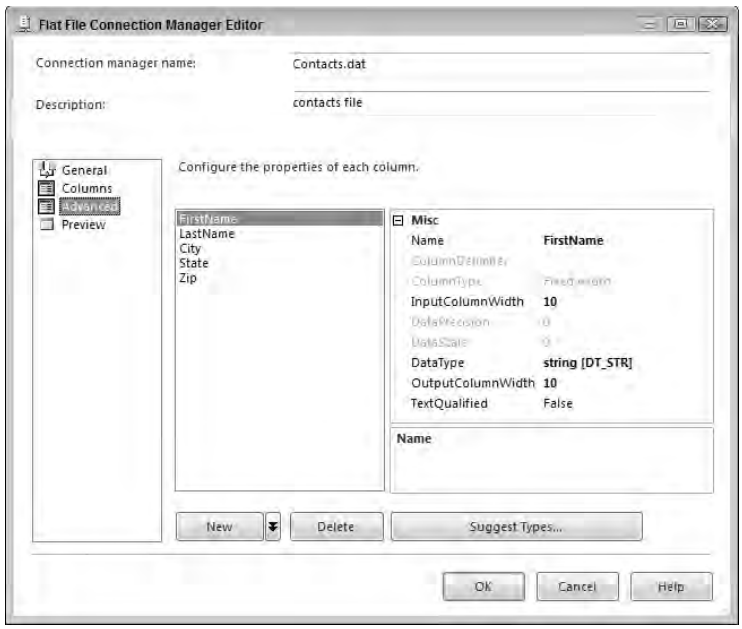

Figure 9-23

This data is all string-based, so we are hiding a complexity here. Typically, there is some data that you may want to define with strong types. Make the decision about whether you want to do that here in the Connection Manager, or later using a derived column depending upon how confident you are in the source of the data. If the Data Source is completely unreliable, import data using Unicode strings and use your Data Flow Tasks to validate the data. Then move good data into a strong data type using the Derived Column Transform.

On the Data Flow surface, drag a Flat File Source to the Data Flow editor pane. Edit the Flat File Source and set the Connection Manager to the Contract Mainframe Extract Connection Manager. This sets up the origination of the data to stream into the Data Flow Task. Check the box labeled "Retain null values from the source as null values in the Data Flow." This new feature allows the consistent testing of null values later. This hiding of null values was one of the problems with earlier versions of SSIS. Now, add a Script Component to the Data Flow. When you drop the Script Component, you will be prompted to pick the type of component to create. Select Transformation and click OK. Connect the output of the Flat File Source to the Script Component to pipe the data into this component where we can program some validation on the data.

Open up the Script Component and set the ScriptLanguage to the language of your choice. On the Input Columns tab, you will notice that Input Name is a drop-down with the name Input 0. It is possible to have more than one source pointed to this Script Component. If you had this situation, this drop-down would allow you to individually configure the inputs and select the columns from each input. For this example, select all the input columns. Set the Usage Type for the State and Zip columns to ReadWrite. The reason will be clear later.

Select the Inputs and Outputs tab to see the collection of inputs and outputs and the input columns defined previously. Here you can create additional input and output buffers and columns within each. Expand all the nodes and add these two output columns:

Column Name	Type	Size
Good Flag	DT_BOOL	N/A
RejectReason	DT_STR	50

You'll use the flag to separate the data from the data stream. The reject reason will be useful to the person who'll have to perform any manual work on the data later. The designer with all nodes expanded should look like Figure 9-24.

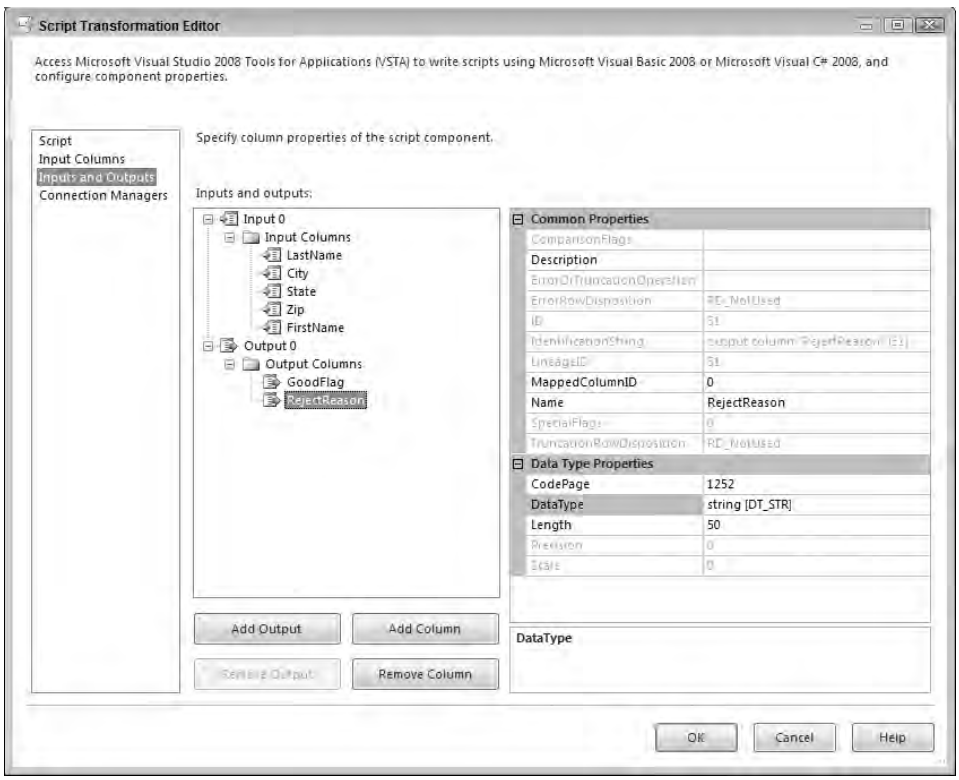

Figure 9-24

Back on the Script tab, click the Edit Script button to enter the VSTA scripting IDE. In the main class, the rules for validation need to be programmatically applied to each data row. In the `Input0_ProcessInputRow` method that was co-generated by SSIS using the Script Component designer, add the rules for data validation, which are:

- All fields are required except for the zip code.
- The zip code must be in the format #####-#### or ##### and a numeric digit from 0 through 9. If the zip code is valid for the first five characters, but the whole string is not, strip the trailing records and use the first five.
- The state must be two uppercase characters.

Here's the overall plan: The contents of the file will be sent into the Script Component. This is where programmatic control will be applied to each row processed. The incoming row has three data fields that need to be validated to determine that all necessary data is present. The State and Zip columns need to be validated additionally by rule and even to be cleaned up if possible. The need to fix the data in the stream is why the Zip and State column usage types had to be set to ReadWrite in the designer earlier.

To aid in accomplishing these rules, the data will be validated using regular expressions. Regular expressions are a powerful utility that should be in every developer's tool belt. They allow you to perform powerful string matching and replacement routines. You can find an excellent tutorial on regular expressions at `http://www.regular-expressions.info`. The regular expressions for matching the data are here:

Regular Expression	Validation Description
^\d{5}([\-]\d{4})?$	Matches a five-digit or nine-digit zip code with dash
\b([A-Z]{2})\b	Ensures that the state is only two capital characters

To use the regular expression library, add the .NET `System.Text.RegularExpressions` namespace to the top of the main class. For performance reasons, create the instances of the `RegEx` class to validate the `ZipCode` and the `State` in the `PreExecute()` method of the Script Component. This method and the private instances of the Regex classes should look like this:

```
C#
    private Regex zipRegex;
    private Regex stateRegex;
    public override void PreExecute()
    {
        base.PreExecute();
        zipRegex = new Regex("^\\d{5}([\\-]\\d{4})?$", RegexOptions.None);
        stateRegex = new Regex("\\b([A-Z]{2})\\b", RegexOptions.None);
    }

VB
    Private zipRegex As Regex
    Private stateRegex As Regex

    Public Overrides Sub PreExecute()
        MyBase.PreExecute()
        zipRegex = New Regex("^\d{5}([\-]\d{4})?$", RegexOptions.None)
        stateRegex = New Regex("\b([A-Z]{2})\b", RegexOptions.None)
    End Sub
```

To break up the tasks, create two new private functions to validate the `ZipCode` and `State`. Using `byRef` arguments for the reason and the `ZipCode` enables the data to be cleaned and the encapsulated logic to return both a true/false as well as the reason. The `ZipCode` validation functions should look like this:

```
C#
private bool ZipIsValid(ref string zip, ref string reason)
    {
        zip = zip.Trim();
        if (zipRegex.IsMatch(zip))
        {
            return true;
        }
        else
        {
            if (zip.Length > 5)
            {
                zip = zip.Substring(0, 5);
                if (zipRegex.IsMatch(zip))
                {
                    return true;
                }
                else
                {
                    reason = "Zip larger than 5 Chars, " +
                                    "Retested at 5 Chars and Failed";
                    return false;
                }
            }
            else
            {
                reason = "Zip Failed Initial Format Rule";
                return false;
            }
        }
    }

VB
    Private Function ZipIsValid(ByRef zip As String, _
                  ByRef reason As String) As Boolean
        zip = zip.Trim()
        If (zipRegex.IsMatch(zip)) Then
            Return True
        Else
            If (zip.Length > 5) Then
                zip = zip.Substring(0, 5)
                If (zipRegex.IsMatch(zip)) Then
                    Return True
                Else
                    reason = "Zip larger than 5 Chars, " + _
                         "Retested at 5 Chars and Failed"
                    Return False
                End If
            Else
```


Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

```
                reason = "Zip Failed Initial Format Rule"
                Return False
            End If
        End If
    End Function
```

The state validation functions look like this:

```
C#
    private bool StateIsValid(ref string state, ref string reason)
    {
        state = state.Trim().ToUpper();
        if (stateRegex.IsMatch(state))
        {
            return true;
        }
        else
        {
            reason = "Failed State Validation";
            return false;
        }
    }

VB
    Private Function StateIsValid(ByRef state As String, _
    ByRef reason As String) As Boolean
        state = state.Trim().ToUpper()
        If (stateRegex.IsMatch(state)) Then
            Return True
        Else
            reason = "Failed State Validation"
            Return False
        End If
    End Function
```

Now, to put it all together add the driver method `Input0_ProcessInputRow()` that is fired upon each row of the flat file:

```
C#
    public override void Input0_ProcessInputRow(Input0Buffer Row)
    {
        Row.GoodFlag = false;
        string myZip = string.Empty;
        string myState = string.Empty;
        string reason = string.Empty;

        if (!Row.FirstName_IsNull && !Row.LastName_IsNull &&
            !Row.City_IsNull && !Row.State_IsNull && !Row.Zip_IsNull)
        {
            myZip = Row.Zip;
            myState = Row.State;
            if (ZipIsValid(ref myZip, ref reason) &&
                        StateIsValid(ref myState, ref reason))
```

```
            {
                Row.Zip = myZip;
                Row.State = myState;
                Row.GoodFlag = true;
            }
            else
            {
                Row.RejectReason = reason;
            }
        }
        else
        {
            Row.RejectReason = "All Required Fields not completed";
        }
    }

VB
    Public Overrides Sub Input0_ProcessInputRow(ByVal Row As Input0Buffer)
        Dim myZip As String = String.Empty
        Dim myState As String = String.Empty
        Dim reason As String = String.Empty

        If (Row.FirstName_IsNull = False And _
            Row.LastName_IsNull = False And _
            Row.City_IsNull = False And _
            Row.State_IsNull = False And _
            Row.Zip_IsNull = False) Then
            myZip = Row.Zip
            myState = Row.State
            If (ZipIsValid(myZip, reason) And _
                    StateIsValid(myState, reason)) Then
                Row.Zip = myZip
                Row.State = myState
                Row.GoodFlag = True
            Else
                Row.RejectReason = reason
            End If
        Else
            Row.RejectReason = "All Required Fields not completed"
        End If
    End Sub
```

Notice that all fields are checked for null values using a property on the `Row` class that is the field name and an additional tag `_IsNull`. This is a property code generated by SSIS when you set up the input and output columns on the Script Component. Properties like `Zip_IsNull` explicitly allow the checking of a null value without encountering a Null Exception. This is handy as the property returns true if the particular column is NULL.

Next, if the Zip column is not NULL, its value is matched against the regular expression to see if it's in the correct format. If it is, the value is assigned back to the Zip column as a cleaned data element. If the value of the Zip column doesn't match the regular expression, the script checks to see if it is at least five

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

characters long. If true, then the first five characters are retested for a valid `ZipCode` pattern. Non-matching values result in a GoodFlag in the output columns being set to `False`.

The state is trimmed of any leading or trailing white space, and then converted to uppercase and matched against the regular expression. The expression simply checks to see if it's two uppercase letters between A and Z. If it is, the GoodFlag is set to `True` and the state value is updated; otherwise, the GoodFlag is set to `False`.

To send the data to the appropriate table based on the GoodFlag, you must use the Conditional Split Task. Add this task to the Data Flow designer and connect the output of the Script Component Task to the Conditional Split Transformation. Edit the Conditional Split Transformation, and add an output named `Good` with the condition `GoodFlag == FALSE` and another output named `Bad` with the condition `GoodFlag == TRUE`. This separates the data rows coming out of the Script Component Task into two separate streams. Another way to do this is only define one stream and let the default stream be the other condition, but it seems more explicit to create streams for both conditions. The Conditional Split Transform Editor should look like Figure 9-25.

Figure 9-25

Add an OLE Connection Manager that uses the database you created for the Contacts and ContactsErrorQueue tables. Add two SQL Server Destinations to the Data Flow designer. One, named Good Destination, should point to the Contacts table; the other, to the ContactsErrorQueue table. Drag the Good output of the Conditional Split Task to the Good Destination. Set the output stream named

Good to the destination. Then open the Mappings tab in the destination to map the input stream to the columns in the Contacts table. Repeat this for the other Bad output of the Conditional Split Task to the Bad Destination.

Your final Data Flow should look something like Figure 9-26. If you run this package with the `Contacts.dat` file described at the top of the use-case, three contacts will validate, and two will fail with these rejection reasons:

```
Failed State Validation
Joseph    McClung        JACKSONVILLE      FLORIDA 322763939

Zip Failed Initial Format Rule
Andrew    Ranger         Jax               fl
```

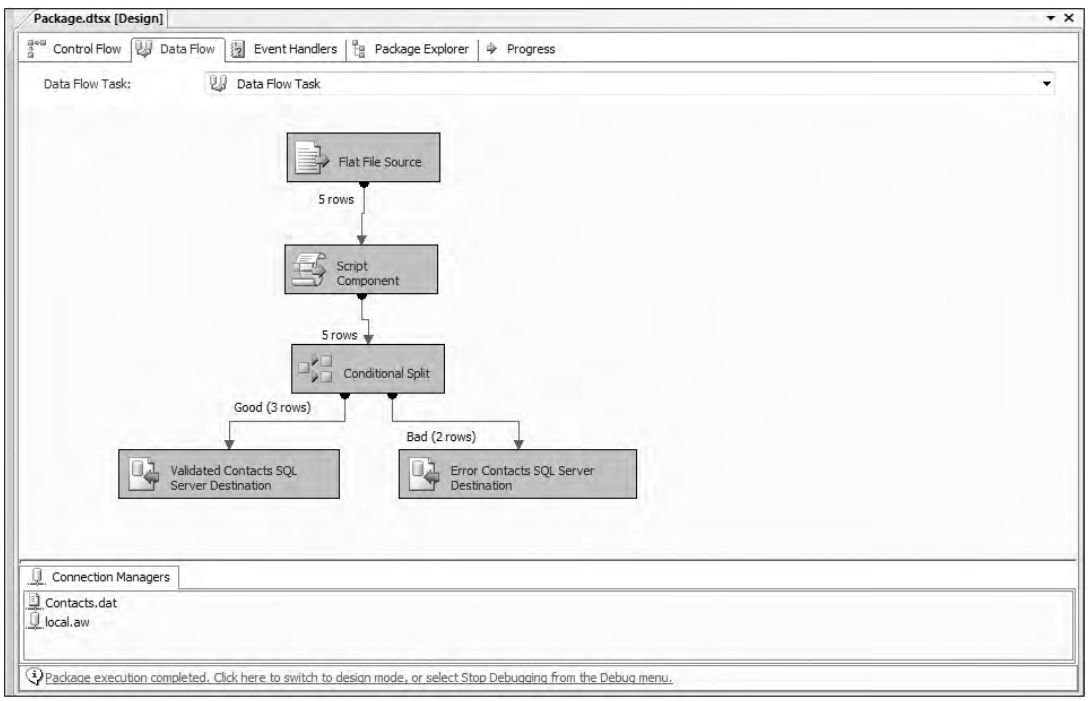

Figure 9-26

At this point, you've gotten a good overview of how scripting works in SSIS and the difference between the Script Task and the Scripting Component, but as with any programming environment, you need to know how to troubleshoot and debug your code to get all of this to work. We'll look at some of the common coding tasks in the next section that you'll need to allow for more advanced SSIS scripting development.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

Essential Coding, Debugging, and Troubleshooting Techniques

We have been all over the new VSTA development environment and have introduced you to the addition of C# that moves SSIS development into the managed code arena. Now, we need to circle up and dig into some of the techniques of hardening our code for unexpected issues that occur during runtime and look at some of the techniques of troubleshooting SSIS packages. There are some differences between the Script Task and the Script Component for some of these techniques that we'll highlight here, now that you are familiar with both.

Structured Exception Handling

Structured Exception Handling (SEH) allows you to catch specific errors as they occur and perform any appropriate action needed. In many cases, you just want to log the error and stop execution, but there are some instances where you may want to try a different plan of action, depending on the error.

Here is an example of exception handling in SSIS scripting code in both languages:

```
C#
public void Main()
{
    try
    {
        string fileText = string.Empty;
        fileText = System.IO.File.ReadAllText("c:\\data.csv");
    }
    catch (System.IO.FileNotFoundException ex)
    {
        //Log Error Here
        //MessageBox here for demo purposes only
        System.Windows.Forms.MessageBox.Show(ex.ToString());
        Dts.TaskResult = (int)ScriptResults.Failure;
    }
    Dts.TaskResult = (int)ScriptResults.Success;
}

VB
Public Sub Main()
    Try
      Dim fileText As String
      fileText = FileIO.FileSystem.ReadAllText("C:\data.csv")
    Catch ex As System.IO.FileNotFoundException
      'Log Error Here
      'MessageBox here for demo purposes only
      System.Windows.Forms.MessageBox.Show(ex.ToString())
      Dts.TaskResult = ScriptResults.Failure
      Return
    End Try
    Dts.TaskResult = ScriptResults.Success
End Sub
```

This trivial example attempts to read the contents of the file at `C:\data.csv` into a string variable. This code makes some assumptions that might not be true. An obvious assumption is that the file exists. That is why this code was placed in a `Try` block. It is trying to perform an action that has the potential for failure. If the file isn't there, a `System.IO.FileNotFoundException` is thrown. A `Try` block marks a section of code that contains function calls with potentially known exceptions. In this case, the `FileSystem ReadAllText` function has the potential to throw a concrete exception.

The `Catch` block is the error handler for this specific exception. You would probably want to add some code to log the error inside the `Catch` block. For now, we've sent the exception to the message box as a string so that it can be viewed. See later in the chapter under each Script object type for the method to perform logging of this type. This code obviously originates from a Scripting Task since it returns a result. The result is set to `Failure`, and the script is exited with the `Return` statement if the exception occurs. If the file is found, no exception is thrown, and the next line of code is executed. In this case, it would go to the line that sets the `TaskResult` to the value of the Success enum, right after the `End Try` statement.

If an exception is not caught, the exception propagates up the call stack until an appropriate handler is found. If none is found, the exception stops execution. You can have as many `Catch` blocks associated with a `Try` block as you wish. When an exception is raised, the `Catch` blocks are walked from top to bottom until an appropriate one is found that fits the context of the exception. Only the first block that matches will be executed. Execution does not fall through to the next block, so it's important to place the most specific `Catch` block first and descend to the least specific. A `Catch` block specified with no filter will catch all exceptions. Typically, the coarsest `Catch` block is listed last. The previous code was written to anticipate the error of a file not being found, so not only does the developer have an opportunity to add some recovery code, but the framework assumes that you'll handle the details of the error itself. If the same code only contained a generic `Catch` statement, the error would simply be written to the package output. To see what this looks like replace the `Catch` statement in the preceding code snippet with these:

```
C#
Catch()

VB
Catch
```

Then the error would simply be written to the package output like this:

```
SSIS package "Package.dtsx" starting.
Error: 0x1 at VB Script Task: System.Reflection.TargetInvocationException,
mscorlib
System.IO.FileNotFoundException, mscorlib

System.Reflection.TargetInvocationException: Exception has been thrown by the
target of an invocation. ---> System.IO.FileNotFoundException: Could not find
file 'C:\data.csv'.
File name: 'C:\data.csv'
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access,
Int32 rights, Boolean useRights, FileShare share, Int32 bufferSize,
FileOptions options, SECURITY_ATTRIBUTES secAttrs, String msgPath, Boolean
bFromProxy)
…
Task failed: VB Script Task
SSIS package "Package.dtsx" finished: Success.
```


Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

The full stack is cut off for brevity and to point out that the task status shows that it failed.

Another feature of SEH is the `Finally` block. The `Finally` block exists inside a `Try` block and executes after any code in the `Try` block and any `Catch` blocks that were entered. Code in the `Finally` block is always executed, regardless of what happens in the `Try` block and in any `Catch` blocks. You would put code to dispose of any resources, such as open files or database connections, in the `Finally` block. Following is an example of using the `Finally` block to free up a connection resource:

```
C#
public void OpenConnection(string myConStr)
{
    SqlConnection con = new SqlConnection(myConStr);
    try
    {
        con.Open();
        //do stuff with con
    }
    catch (SqlException ex)
    {
        //log error here
    }
    finally
    {
        if (con != null)
        {
           con.Dispose();
        }
     }
}

VB
Public Sub OpenConnection(myConStr as String)
   Dim con As SqlConnection = New SqlConnection(myConStr)
   Try
       con.Open()
       'do stuff with con
   Catch ex As SqlException
       'Log Error Here
       Dts.TaskResult = Dts.Results.Failure
       Return
   Finally
       If Not con Is Nothing Then con.Dispose()
   End Try
End Sub
```

In this example, the `Finally` block is hit regardless of whether the connection is open or not. A logical `If` statement checks to see if the connection is open and closes it to conserve resources. Typically you want to follow this pattern if you are doing anything resource intensive like using the `System.IO` or `System.Data` assemblies.

> *For a full explanation of the Try/Catch/Finally structure in Visual Basic.NET, see the language reference in MSDN or Books Online.*

Script Debugging and Troubleshooting

Debugging is an important new feature of scripting in SSIS. You can still use the technique of popping up a message box function to see the value of variables, but there are more sophisticated techniques that will help you pinpoint the problem. Using the Visual Studio for Applications environment, you now have the ability to set breakpoints, examine variables, and even evaluate expressions interactively.

Breakpoints

Breakpoints allow you to flag a line of code where execution pauses while debugging. Breakpoints are invaluable in determining what's going on inside your code. They allow you to step into your code and see what happens as it executes. Unfortunately, breakpoints only work inside of Script Tasks.

You can set a breakpoint in several ways. One way is to click in the gray margin at the left of the text editor at the line where you wish to stop execution. Another way is to move the cursor to the line you wish to break on and hit F9. Yet another way is to select Debug ⇨ Toggle Breakpoint.

To continue execution from a breakpoint, press F10 to step to the next line or F5 to run all the way through to the next breakpoint.

When you have a breakpoint set on a line, the line gets a red highlight like the one shown in Figure 9-27.

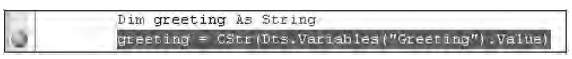

Figure 9-27

When a Script Task has a breakpoint set somewhere in the code, it will have a red dot on it similar to the one in Figure 9-28.

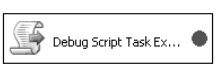

Figure 9-28

Row Count and Data Viewers

Previously, you looked at using the Visual Studio for Applications environment to debug a Script Task using breakpoints and other tools. Unfortunately, you do not have the ability to debug the Script Component using this environment. Any breakpoints that you set will be ignored. Instead, you must resort to inspecting the data stream using the Row Count Component or a Data Viewer.

The Row Count Component is very straightforward; it simply states how many rows passed through it. The Data Viewer is a much better way to debug your component, however. To add a Data Viewer, select the connector arrow, leaving the component that you want to see data for. In the previous example, this would be the connector from the Script Component to the Conditional Split Task. Right-click this connection and select Data Viewers. The Data Flow Path Editor will pop up. Click Add to add the Data Viewer. On the Configure Data Viewer screen, select Grid as the type. Click the Grid tab and make sure all the columns you wish to see are in the Displayed Columns list. Close out this window and the Data Path Flow Editor window by clicking OK. Figure 9-29 shows the Data Path Flow Editor with a Data Viewer configured on Output 0.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission

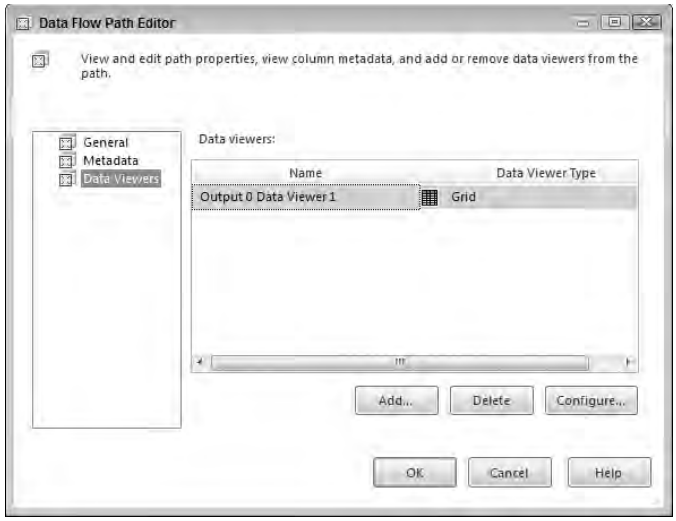

Figure 9-29

Now when you run this package again, you will get a Data Viewer window after the Script Component has executed. This view will show the data output by the Script Component. Figure 9-30 shows an example. Click the Play button to continue package execution, or simply close the window.

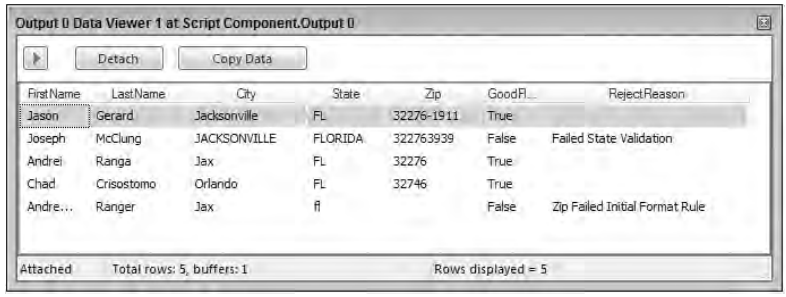

Figure 9-30

While using the Data Viewer certainly helps with debugging, it is no replacement for being able to step into the code. An alternative is to use the `FireInformation` event on the `ComponentMetaData` class in the Script Component. It is like the message box, but without the modal effect.

Autos, Locals, and Watches

The Visual Studio environment provides you with some powerful views into what is happening with the execution of your code. These views consist of three windows known as the Autos window, Locals window, and Watch window. These windows share a similar layout and display the value of expressions and variables, though each has a distinct method determining what data to display.

The Locals window displays variables that are local to the current statement, as well as three statements behind and in front of the current statement. For a running example, the Locals window would appear, as in Figure 9-31.

Figure 9-31

Watches are another very important feature of debugging. Watches allow you to specify a variable to watch. You can set up a watch to break execution when a variable's value changes or some other condition is met. This will allow you to see exactly when something is happening, such as a variable that has an unexpected value.

To add a watch, select the variable you want to watch inside the script, right-click it, and select Add Watch. This will add an entry to the Watch window.

You can also use the Quick Watch window, accessible from the Debug menu, or through the Ctrl+Alt+Q key combination. The Quick Watch window is shown in Figure 9-32 in the middle of a breakpoint, and you can see the value of `ival` as it is being assigned the variable value of 2.

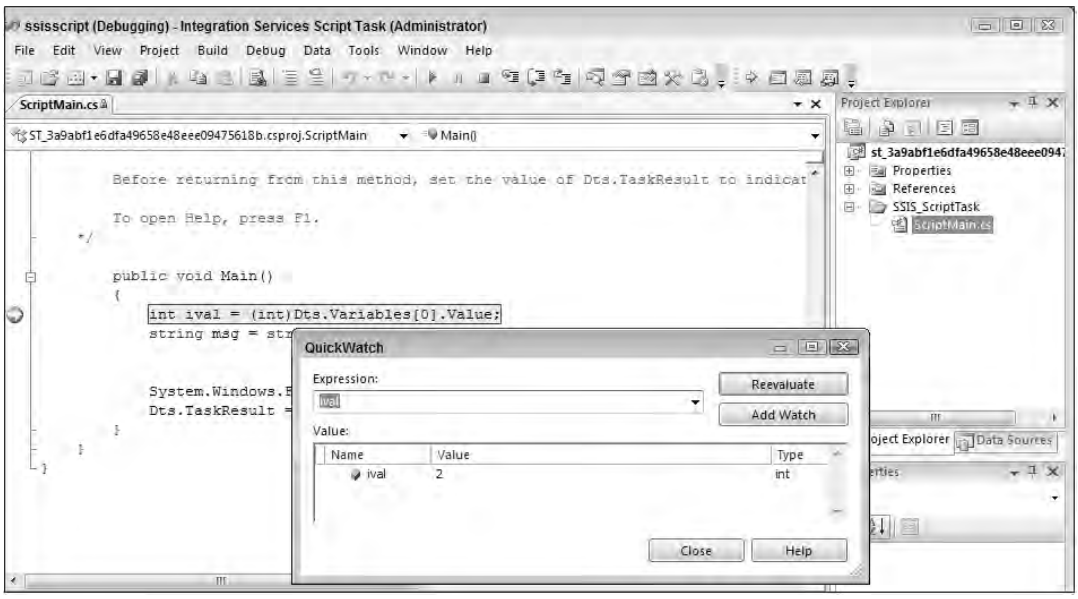

Figure 9-32

This window allows you to evaluate an expression at runtime and see the result in the window. You can then click the Add Watch button to move it to the Watch window.

The Immediate Window

The Immediate window allows you to evaluate expressions, execute procedures, and print out variable values. It is really a mode of the Command window, which allows you to issue commands to the IDE. Unfortunately, this too is only useful when you are within a breakpoint and this can only be done within a Script Task.

> *If you can't find the Immediate window, but see the Command window, just type the command* `immed` *and press Enter.*

The Immediate window is very useful while testing. You can see the outcome of several different scenarios. Suppose you have an object `obj` of type `MyType`. `MyType` declares a method called `DoMyStuff()` that takes a single integer as an argument. Using the Immediate window, you could pass different values into the `DoMyStuff()` method and see the results. To evaluate an expression in the Immediate window and see its results, you must start the command with a question mark (?):

```
?obj.DoMyStuff(2)
"Hello"
```

Commands are terminated by pressing the Enter key. The results of the execution are printed on the next line. In this case, calling `DoMyStuff()` with a value of 2 returns the string "Hello."

You can also use the Immediate window to change the value of variables. If you have a variable defined in your script and you want to change its value, perhaps for negative error testing, you can use this window, as shown in Figure 9-33.

Figure 9-33

In this case, the value of the variable greeting is printed out on the line directly below the expression. After the value is printed, it is changed to "Goodbye Cruel World." The value is then queried again, and the new value is printed. If you are in a Script Task and need to get additional information, this is a useful way to do it.

Summary

In this chapter, you learned about the available scripting options in SSIS from the beginning with DTS and the use of ActiveX scripts to the new versions of SSIS that support managed code development and a robust IDE development environment. You used the new Visual Studio Tools for Applications IDE to

develop some basic Script Tasks. Then, to see how all this fits together in SSIS, we dove right in to using the Script Tasks to retrieve data into variables, save data into external XML files, and used some .NET serialization techniques that can allow custom serialization into MSMQ queues or Web services. To understand how to leverage existing code libraries, you even created a utility class, registered it into the GAC, and accessed it in an SSIS script to validate data.

SSIS scripting is powerful, but it has been difficult for some to differentiate between when to use a Script Task and when a Script Component is appropriate. You have been all over both of these in detail in this chapter and can now use these with confidence in your daily development.

Experiment with the scripting features of SSIS using the examples in this chapter, and you will find all kinds of uses for them. Don't forget to review the chapter on expressions to learn about the capabilities of controlling properties within the SSIS model at runtime. Now we are going to take what we've learned so far about the SSIS toolset capability from Control Flow and Data Flow Tasks, to Expressions and Scripting Tasks and Components and put them to work. Read on to the next chapter for a breakdown on the techniques you need to do a typical job of loading a data warehouse using SSIS services.

Excerpted from Professional Microsoft SQL Server 2008 Integration Services
ISBN: 978-0-470-24795-2, Wrox Press, Posted by Permission