5

Advanced WMI Scripting Techniques

5.1 Objective

In the previous chapter, we examined the WMI scripting object model with its basic operations to retrieve real-world manageable object information. Although, we perform the exact same task in this chapter, we explore some advanced scripting techniques with the same set of WMI scriptable objects. In the previous chapter, we used a specific set of methods that implemented synchronous scripting techniques, which resulted in a WMI operation (i.e., retrieving instance information) executing in-line with the code execution. In this chapter, we see how to use the WMI asynchronous operations by examining another set of methods available from the same WMI object collection. The asynchronous scripting techniques allow the execution of one more subroutine in parallel, which increases the execution performance. WMI extends ADSI by adding new properties and methods to facilitate the instantiation of some WMI scriptable objects. Moreover, WMI also exposes some helper objects to manipulate the DMTF date format or to retrieve instance information in XML. Let's see how these advanced features work.

5.2 Working with CIM instances asynchronously

The **SWbemServices** and **SWbemObject** objects expose various methods to perform different tasks (see Tables 4.8 and 4.10). Some of these methods are executed in line with the script code and force the script to pause until the method is executed (synchronous execution). Depending on the action to be performed, some methods may take some time to execute (i.e., stopping the Exchange Information Store service of an Exchange server hosting a mailbox store of 25 GB or more). To increase script execution performance and responsiveness, the WMI scripting API exposes a second set of identical methods but, at invocation time, these methods do not wait for

the end of the WMI method execution; they are executed asynchronously. This allows execution of the script to continue while the invoked method is performed. When monitoring a system, it is likely that a process watching different manageable entities must perform several actions at the same time. A process that stops while executing one operation may cause trouble if an action with higher priority must be taken at the same time.

Table 5.1 *The Synchronous Methods Versus the Asynchronous Methods*

SWbemServices ***Methods***		
Synchronous	*Asynchronous*	
AssociatorsOf	AssociatorsOfAsync	Returns a collection of objects (classes or instances) that are associated with a specified object. This method performs the same function that the ASSOCIATORS OF WQL query performs.
Delete	DeleteAsync	Deletes an instance or class.
ExecMethod	ExecMethodAsync	Executes an object method.
ExecNotificationQuery	ExecNotificationQueryAsync	Executes a query to receive events.
ExecQuery	ExecQueryAsync	Executes a query to retrieve a collection of objects (classes or instances).
Get	GetAsync	Retrieves a class or instance.
InstancesOf	InstancesOfAsync	Returns a collection of instances of a specified class.
ReferencesTo	ReferencesToAsync	Returns a collection of objects (classes or instances) that refer to a single object. This method performs the same function that the REFERENCES OF WQL query performs.
SubclassesOf	SubclassesOfAsync	Returns a collection of subclasses of a specified class.
SWbemObject ***Methods***		
Synchronous	*Asynchronous*	
Associators_	AssociatorsAsync_	Retrieves the associators of the object.
Delete_	DeleteAsync_	Deletes the object from WMI.
ExecMethod_	ExecMethodAsync_	Executes a method exported by a method provider.
Instances_	InstancesAsync_	Returns a collection of instances of the object (which must be a WMI class).
Put_	PutAsync_	Creates or updates the object in WMI.
References_	ReferencesAsync_	Returns references to the object.
Subclasses_	SubclassesAsync_	Returns a collection of subclasses of the object (which must be a WMI class).

Moving from the WMI synchronous scripting technique to the WMI asynchronous scripting technique moves us closer to the WMI event scripting we discover in the next chapter. The asynchronous scripting provides a good foundation to work with the WMI event scripting because it allows a simultaneous event management. Table 5.1 shows the list of asynchronous methods available from **SWbemServices** and **SWbemObject** objects with their synchronous equivalent methods.

5.2.1 Retrieving a CIM instance

Basically, all operations that can be performed with a synchronous method can be performed with an asynchronous method. In the previous chapter, we saw that the *Get* method of the **SWbemServices** object was able to retrieve a class or an instance based on the WMI path used. We can perform the same operation with the *GetAsync* method. In this case the *GetAsync* method does not return an object. However, if the method invocation is successful, a sink routine receives an **SWbemObject** equivalent to the one obtained with the *Get* method.

Here we introduce a new term in the WMI scripting vocabulary: a sink routine. When working with the WMI asynchronous scripting technique, the script structure is a bit different. Although the WMI connection processed with the **SWbemLocator** object (or with a moniker) and the creation of the **SWbemServices** object are the same, the way the object instance is retrieved is different. Basically, every asynchronous method corresponds to a subroutine that must be part of the script invoking the method. When invoking an asynchronous method, an **SWbemSink** object must be created. This object is passed as a parameter to reference the subroutines that receive the created instance. WMI asynchronous scripting supports four subroutines (called *sink routines*):

1. *OnCompleted* is triggered when an asynchronous operation is completed.
2. *OnObjectPut* is triggered after an asynchronous put operation.
3. *OnObjectReady* is triggered when an object provided by an asynchronous call is available.
4. *OnProgress* is triggered to provide the status of an asynchronous operation. It is important to note that the WMI provider must support the status update to use this sink routine.

Let's look at an example instead of going through long theoretical explanations. Sample 5.1 retrieves the *Win32_Service* SNMP instance asynchro-

nously. It performs the exact same task as Sample 4.1 ("A VBScript listing the Win32_Service instance properties"). Sample 4.1 retrieves the *Win32_Service* SNMP instance synchronously, whereas Sample 5.1, retrieves the same information asynchronously.

Sample 5.1 *A Windows Script File listing the Win32_Service instance properties asynchronously*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
24:    ' ---------------------------------------------------------------------------------------
25:    Const cComputerName = "LocalHost"
26:    Const cWMINameSpace = "root/cimv2"
27:    Const cWMIClass  = "Win32_Service"
28:    Const cWMIInstance = "SNMP"
..:
33:    Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
34:
35:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
36:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
37:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
38:
39:    objWMIServices.GetAsync objWMISink, cWMIClass & "='" & cWMIInstance & "'"
40:
41:    WScript.Echo "Continuing script execution ..."
42:    PauseScript "Click on 'Ok' to terminate the script ..."
43:
44:    objWMISink.Cancel
..:
51:    ' ---------------------------------------------------------------------------------------
52:    Sub SINK_OnCompleted (intHResult, objWMILastError, objWMIAsyncContext)
53:
54:        Wscript.Echo
55:        Wscript.Echo "BEGIN - OnCompleted."
56:
57:        If intHResult = 0 Then
58:           WScript.Echo "WMI Scripting API Asynchronous call successful."
59:        Else
60:           WScript.Echo "WMI Scripting API Asynchronous call failed."
61:        End If
62:
63:        Wscript.Echo "END   - OnCompleted."
64:
65:    End Sub
66:
67:    ' ---------------------------------------------------------------------------------------
68:    Sub SINK_OnObjectPut (objWMIPath, objWMIAsyncContext)
69:
70:        Wscript.Echo
```

```
 71:         Wscript.Echo "BEGIN - OnObjectPut."
 72:         Wscript.Echo "END   - OnObjectPut."
 73:
 74:     End Sub
 75:
 76:     ' ------------------------------------------------------------------------------
 77:     Sub SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
 78:
 79:         Wscript.Echo
 80:         Wscript.Echo "BEGIN - OnObjectReady."
 81:         WScript.Echo objWMIInstance.Name & " (" & objWMIInstance.Description & ")"
 82:         WScript.Echo "   AcceptPause=" & objWMIInstance.AcceptPause
 83:         WScript.Echo "   AcceptStop=" & objWMIInstance.AcceptStop
 84:         WScript.Echo "   Caption=" & objWMIInstance.Caption
 ..:
 98:         WScript.Echo "   Started=" & objWMIInstance.Started
 99:         WScript.Echo "   StartMode=" & objWMIInstance.StartMode
100:         WScript.Echo "   StartName=" & objWMIInstance.StartName
101:         WScript.Echo "   State=" & objWMIInstance.State
102:         WScript.Echo "   Status=" & objWMIInstance.Status
103:         WScript.Echo "   SystemCreationClassName=" & objWMIInstance.SystemCreationClassName
104:         WScript.Echo "   SystemName=" & objWMIInstance.SystemName
105:         WScript.Echo "   TagId=" & objWMIInstance.TagId
106:         WScript.Echo "   WaitHint=" & objWMIInstance.WaitHint
107:         Wscript.Echo "END   - OnObjectReady."
108:
109:     End Sub
110:
111:     ' ------------------------------------------------------------------------------
112:     Sub SINK_OnProgress (intUpperBound, intCurrent, strMessage, objWMIAsyncContext)
113:
114:         Wscript.Echo
115:         Wscript.Echo "BEGIN - OnProgress."
116:         Wscript.Echo "END   - OnProgress."
117:
118:     End Sub
119:
120:     ' ------------------------------------------------------------------------------
121:     Function PauseScript (strMessage)
122:
123:     Dim WshShell
124:
125:         Set WshShell = Wscript.CreateObject("Wscript.Shell")
126:         WshShell.Popup strMessage, 0, "Pausing Script ...", cExclamationMarkIcon + cOkButton
127:         Wscript.DisconnectObject (WshShell)
128:         Set WshShell = Nothing
129:
130:     End Function
131:
132:     ]]>
133:     </script>
134: </job>
135:</package>
```

Once the **SWbemServices** object is created (line 37), the *GetAsync* method can be invoked (line 39). The particularity of the invocation resides in the parameter containing an **SWbemSink** object. Accessing information asynchronously implies a call to a subroutine for the desired operation. It

means that the invoked asynchronous method must know where the call must be executed. We saw that we have four possible sink routines with WMI asynchronous operations, each of which corresponds to a particular event. As we saw in Chapter 1, when working with the Windows Script Components event support, a reference between the event handler and the event name is performed. With WMI, the logic is exactly the same. However, the mechanism is implemented with an **SWbemSink** object created at line 33. It links the sink routines targeted to receive asynchronous calls with the execution of the asynchronous **SWbemServices** method. Since the script specifies a "SINK_" name in an **SWbemSink** object (line 33) for the sink routines concerning the *GetAsync* method execution, the script may have four sink routines named as follows:

1. SINK_OnCompleted () at line 52.
2. SINK_OnObjectPut () at line 68.
3. SINK_OnObjectReady () at line 77.
4. SINK_OnProgress () at line 112.

Of course, because the script executes a *GetAsync* method, the SINK_OnObjectPut() function is never called, as it does not relate to this operation type.

The other parameter of the *GetAsync* method is a traditional instance name to select the WMI instance to work with (line 39). Once the asynchronous method is invoked, the script must pause or proceed with some other tasks. As it is only a small sample designed to show how things works, the script pauses by calling the PauseScript() subfunction (line 42). The PauseScript() function invokes the *Popup* method of the **Wshshell** object from WSH to display a message (lines 121 to 130). Even if the script is stopped by the popup message, you will see that by executing the script the properties of the instantiated SNMP *Win32_Service* are displayed on the screen (see Figure 5.1).

The output shown in Figure 5.1 is generated by the SINK_OnObjectReady() function that receives the instance from the *GetAsync* method. When the method is executed, it produces an **SWbemObject** available from the parameters provided by the sink routine itself (line 77). In comparison with the synchronous *Get* method of the **SWbemServices** object, the **SWbemObject** is produced on return of the call of the *Get* method. Now, with the asynchronous execution of the *GetAsync* method, the produced **SWbemObject** is available from the sink routine called SINK_OnObjectReady(). Except for the location where this object is available, this is

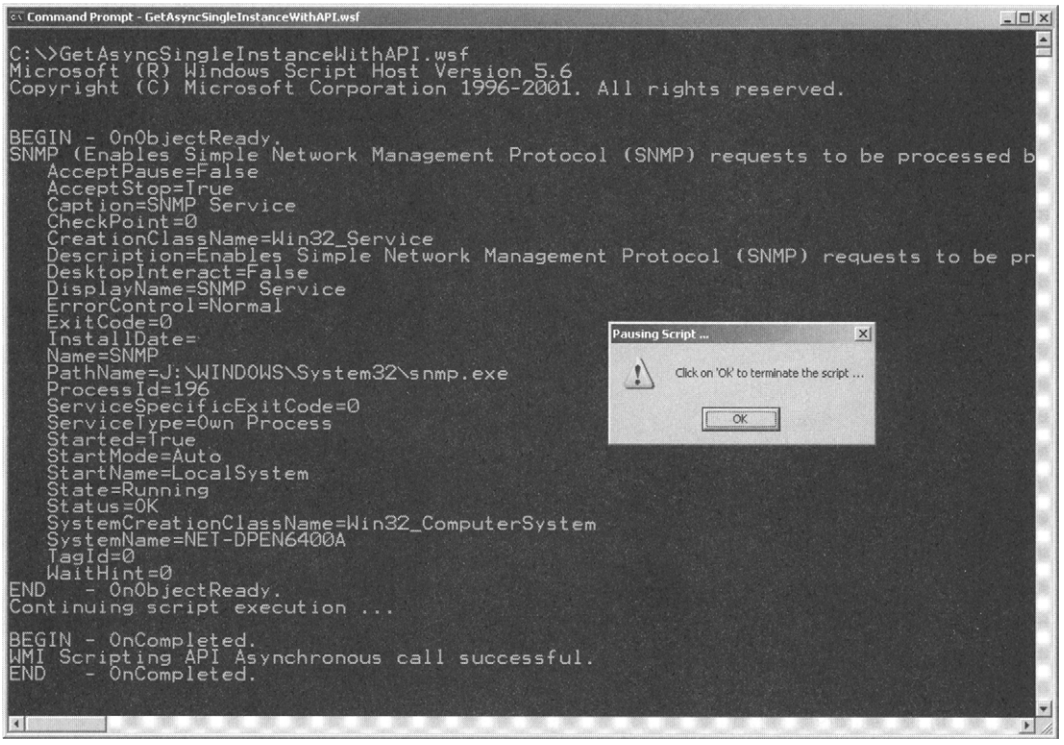


Figure 5.1 *The GetAsync method execution output.*

exactly the same object as before. This is why the routine displays all the properties available from the object instance (lines 81 to 106) as made in Sample 4.1. Once the *GetAsync* method is complete, the SINK_OnCompleted() executes. The execution of the SINK_OnCompleted() uses the **intHResult** parameter to determine whether the asynchronous method execution has been successful.

This last sample retrieves one instance of a *Win32_Service*. What happens if the script retrieves a collection of instances of the *Win32_Service* class? The same mechanisms take place, but every instance available from the asynchronous method generates a SINK_OnObjectReady() function call. To retrieve all instances of the *Win32_Service* class, line 39 can be changed to

```
39:    objWMIServices.InstancesOfAsync objWMISink, cWMIClass
```

or to

```
39:    objWMIServices.ExecQueryAsync objWMISink, "Select * From " & cWMIClass
```

Note the interesting aspect of such logic. With the synchronous scripting technique, when an **SWbemServices** method returns an object collection, it is necessary to enumerate the returned **SWbemObjectSet** object collection with a For Each loop. Now, as the method is asynchronous, there is no need to perform an enumeration, because the sink routine is called for every instance available. At the end of the asynchronous method execution, the SINK_OnCompleted() sink routine is called.

5.2.2 Retrieving CIM information

Because the WMI object retrieved in the sink routine is a traditional **SWbemObject**, all the properties and methods exposed by the COM object itself (see Table 4.10) and by its CIM class definition are available. The complete object model (see Figures 4.3 – 4.5) can be browsed as before to discover the properties, methods, and all related qualifiers. This means that the function SINK_OnObjectReady() in Sample 5.1 can be changed with the code developed in the previous chapter:

- Sample 4.13 "Retrieving the property collection of a WMI instance"
- Sample 4.16 "Retrieving the property qualifiers of a WMI class"
- Sample 4.17 "Retrieving the method collection of a WMI class"
- Sample 4.22 "Retrieving the method qualifiers of a WMI class"
- Sample 4.23 "Retrieving the method properties with their qualifiers of a WMI class"

Sample 5.2 reuses the DisplayQualifiers() function. The same set of information is retrieved as before, except that now the information is retrieved in the core of the SINK_OnObjectReady() function.

Sample 5.2 *Retrieving the properties, methods, and all associated qualifiers of a WMI class asynchronously*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <script language="VBScript" src="..\Functions\DisplayQualifiersFunction.vbs" />
14:
15:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
16:
17:    <script language="VBscript">
18:    <![CDATA[
..:
26:    ' ---------------------------------------------------------------------------------
27:    Const cComputerName = "LocalHost"
```

```
28:    Const cWMINameSpace = "root/cimv2"
29:    Const cWMIClass  = "Win32_Service"
30:    Const cWMIInstance = "SNMP"
..:
37:    Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
38:
39:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
40:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
41:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
42:
43:    objWMIServices.GetAsync objWMISink, cWMIClass & "='" & cWMIInstance & "'"
44:
45:    WScript.Echo "Continuing script execution ..."
46:    PauseScript "Click on 'Ok' to terminate the script ..."
47:
48:    objWMISink.Cancel
..:
55:    ' ------------------------------------------------------------------------------
56:    Sub SINK_OnCompleted (intHResult, objWMILastError, objWMIAsyncContext)
..:
71:    End Sub
72:
73:    ' ------------------------------------------------------------------------------
74:    Sub SINK_OnObjectPut (objWMIPath, objWMIAsyncContext)
..:
82:    End Sub
83:
84:    ' ------------------------------------------------------------------------------
85:    Sub SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
..:
93:        Wscript.Echo
94:        Wscript.Echo "BEGIN - OnObjectReady."
95:
96:        Wscript.Echo String (60, 45)
97:        Wscript.Echo "Class Qualifiers:"
98:
99:        WScript.Echo "  " & objWMIInstance.Path_.RelPath
100:       DisplayQualifiers objWMIInstance.Qualifiers_, 0
101:
102:       Wscript.Echo String (60, 45)
103:       Wscript.Echo "Property Qualifiers:"
104:
105:       Set objWMIPropertySet = objWMIInstance.Properties_
106:       For Each objWMIProperty In objWMIPropertySet
107:           WScript.Echo "  " & objWMIProperty.Name
108:           DisplayQualifiers objWMIProperty.Qualifiers_, 0
109:       Next
110:       Set objWMIPropertySet = Nothing
111:
112:       Wscript.Echo String (60, 45)
113:       Wscript.Echo "Method Qualifiers:"
114:
115:       Set objWMIMethodSet = objWMIInstance.Methods_
116:       For Each objWMIMethod In objWMIMethodSet
117:           WScript.Echo "  " & objWMIMethod.Name
118:           DisplayQualifiers objWMIMethod.Qualifiers_, 0
119:
120:           Set objWMIObject = objWMIMethod.InParameters
121:           WScript.Echo "  " & objWMIObject.Path_.RelPath
122:           DisplayQualifiers objWMIObject.Qualifiers_, 2
```

```
123:
124:            Set objWMIPropertySet = objWMIObject.Properties_
125:            If Err.Number = 0 Then
126:               For Each objWMIProperty In objWMIPropertySet
127:                   WScript.Echo "    " & objWMIProperty.Name
128:                   DisplayQualifiers objWMIProperty.Qualifiers_, 4
129:               Next
130:            Else
131:               Err.Clear
132:            End If
133:            Set objWMIPropertySet = Nothing
134:
135:            Set objWMIObject = Nothing
136:
137:            Set objWMIObject = objWMIMethod.OutParameters
138:            WScript.Echo "  " & objWMIObject.Path_.RelPath
139:            DisplayQualifiers objWMIObject.Qualifiers_, 2
140:
141:            Set objWMIPropertySet = objWMIObject.Properties_
142:            If Err.Number = 0 Then
143:               For Each objWMIProperty In objWMIPropertySet
144:                   WScript.Echo "    " & objWMIProperty.Name
145:                   DisplayQualifiers objWMIProperty.Qualifiers_, 4
146:               Next
147:            Else
148:               Err.Clear
149:            End If
150:            Set objWMIPropertySet = Nothing
151:
152:            Set objWMIObject = Nothing
153:
154:        Next
155:        Set objWMIMethodSet = Nothing
156:
157:        Wscript.Echo "END   - OnObjectReady."
158:
159:    End Sub
160:
161:    ' ------------------------------------------------------------------------------
162:    Sub SINK_OnProgress (intUpperBound, intCurrent, strMessage, objWMIAsyncContext)
...:
170:    End Sub
171:
172:    ' ------------------------------------------------------------------------------
173:    Function PauseScript (strMessage)
...:
182:    End Function
183:
184:    ]]>
185:    </script>
186:  </job>
187:</package>
```

5.2.3 Modifying a CIM instance property

Up to now, the asynchronous script uses the SINK_OnObjectReady() and the SINK_OnCompleted() sink routines. The SINK_OnObjectPut() has

not yet been used. Moreover, the asynchronous method invocation was always made from an **SWbemServices** object. In the previous chapter, Sample 4.14 ("Setting one read/write property of a Win32_Registry class instance directly") sets one read/write property of the *Win32_Registry* class instance. In this sample, the *Put_* method of an **SWbemObject** is used to commit the change. To perform the same operation asynchronously, the *PutAsync_* method must be used (see Table 4.10). Of course, to modify an object instance, the script must create the instance representing the manageable entity by using the *Get* method of an **SWbemServices** object. If the operation is also made asynchronously, the *GetAsync* method must be used (see Table 4.8). So, in such a situation, we face two asynchronous method invocations: one to get the object instance and one to commit the changes made to the object instance. Let's see how this works in Sample 5.3.

Sample 5.3 *Setting asynchronously one read/write property of a Win32_Registry class instance*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
 ..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
 ..:
24:    ' ---------------------------------------------------------------------------
25:    Const cComputerName = "LocalHost"
26:    Const cWMINameSpace = "root/cimv2"
27:    Const cWMIClass = "Win32_Registry"
28:    Const cWMIInstance = "Microsoft Windows 2000 Server|C:\WINNT|\Device\Harddisk0\Partition1"
 ..:
33:    Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
34:
35:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
36:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
37:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
38:
39:    objWMIServices.GetAsync objWMISink, cWMIClass & "='" & cWMIInstance & "'"
40:
41:    WScript.Echo "Continuing script execution ..."
42:    PauseScript "Click on 'Ok' to terminate the script ..."
43:
44:    objWMISink.Cancel
 ..:
50:    ' ---------------------------------------------------------------------------
51:    Sub SINK_OnCompleted (intHResult, objWMILastError, objWMIAsyncContext)
 ..:
64:    End Sub
65:
66:    ' ---------------------------------------------------------------------------
67:    Sub SINK_OnObjectPut (objWMIPath, objWMIAsyncContext)
```

```
 68:
 69:         Wscript.Echo
 70:         Wscript.Echo "BEGIN - OnObjectPut."
 71:         Wscript.Echo "END   - OnObjectPut."
 72:
 73:     End Sub
 74:
 75:     ' ------------------------------------------------------------------------------
 76:     Sub SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
 77:
 78:         Wscript.Echo
 79:         Wscript.Echo "BEGIN - OnObjectReady."
 80:
 81:         WScript.Echo objWMIInstance.Name & " (" & objWMIInstance.Description & ")"
 82:
 83:         Wscript.Echo "Current registry size is: " & objWMIInstance.ProposedSize  & " MB."
 84:
 85:         objWMIInstance.ProposedSize = objWMIInstance.ProposedSize + 10
 86:         objWMIInstance.PutAsync_ objWMISink, wbemChangeFlagUpdateOnly Or wbemFlagReturnWhenComplete
 87:
 88:         Wscript.Echo "Current registry size is now: " & objWMIInstance.ProposedSize  & " MB."
 89:
 90:         Wscript.Echo "END   - OnObjectReady."
 91:
 92:     End Sub
 93:
 94:     ' ------------------------------------------------------------------------------
 95:     Sub SINK_OnProgress (intUpperBound, intCurrent, strMessage, objWMIAsyncContext)
...:
101:     End Sub
102:
103:     ' ------------------------------------------------------------------------------
104:     Function PauseScript (strMessage)
...:
113:     End Function
114:
115:     ]]>
116:     </script>
117:   </job>
118:</package>
```

As in previous samples, the script starts with the WMI connection (lines 35 to 37). The **SWbemSink** object is also created (line 33) with references to the sink routines starting with the name "SINK_". Next, the script executes the *GetAsync* method from the **SWbemServices** object (line 39) with the **SWbemSink** object just created. Once the asynchronous call is invoked, the script pauses (line 42).

When WMI retrieves the instance of the *Win32_Registry* class, the SINK_OnObjectReady() sink routine is called. Here, we recognize a code portion used in Sample 4.14 ("Setting one read/write property of a Win32_Registry class instance directly") to modify the current registry size limit and increase that limit by 10 MB (lines 81 to 88). There is no change in the logic. The only modification resides in the way the change is committed back to the instance. Sample 4.14 uses the synchronous *Put_*

method of the **SWbemObject**, whereas Sample 5.3 uses the *PutAsync_* method of the **SWbemObject** (line 86).

The particularity of the *PutAsync_* asynchronous method invocation is that the **SWbemSink** object is the same as the one used for the *GetAsync* asynchronous method invocation. This means that the same set of sink routines are used for both asynchronous operations. Since these two asynchronous operations are different, the *GetAsync* method calls the SINK_OnObjectReady() sink routine, and the *PutAsync_* method calls the SINK_OnObjectPut() sink routine. Figure 5.2 shows the sequence of the sink routine execution.

Because we have two asynchronous calls in the script, Figure 5.2 shows that the SINK_OnCompleted() sink routine is called two times. Here, we face a tricky problem. As the same piece of code is executed for both asynchronous methods, how do we determine whether the SINK_OnCompleted() call is for the *GetAsync* method execution or for the *PutAsync_*

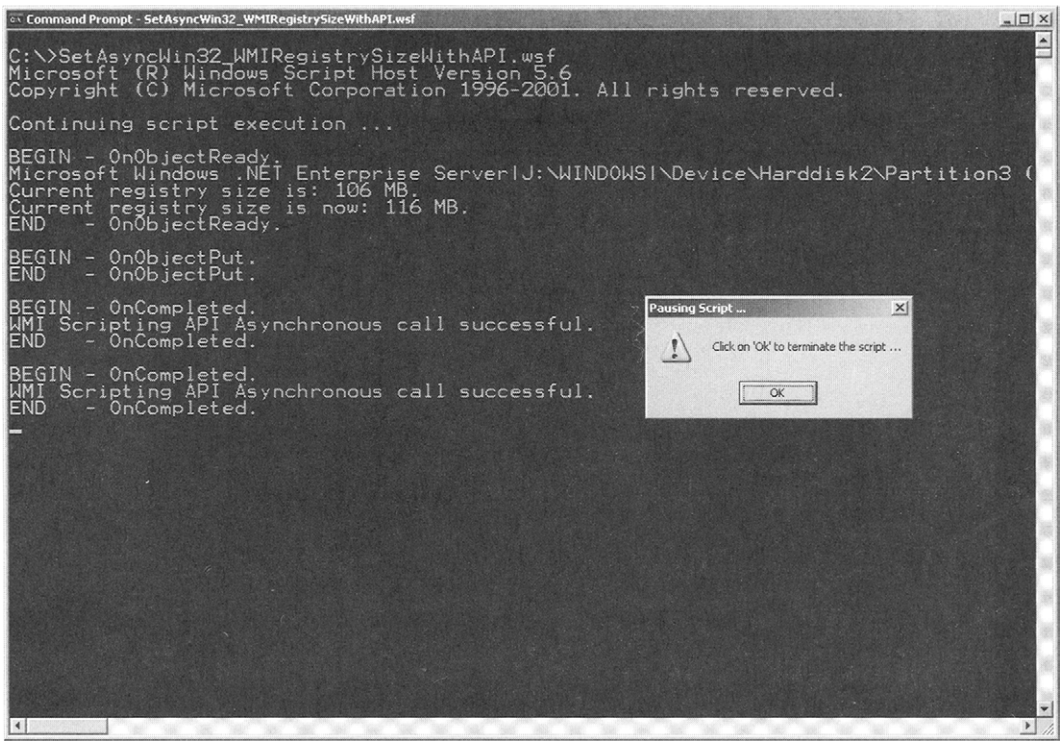


Figure 5.2 *Asynchronously setting one read/write property of a Win32_Registry class instance directly.*

method execution? Currently, the way the script is written there is no way to determine that from the sink routine. This is where the notion of context for an asynchronous call is helpful. Of course, there is another possibility. When invoking the *PutAsync_* method, it was possible to use another **SWbemSink** object that references another set of sink routines (with the name "SINK2_" for instance). Although this method is valid, it does not make use of the notion of context as provided by the WMI scripting API. The context is a parameter provided by every WMI sink routine at the condition that it is defined when invoking the asynchronous method. The context is an **SWbemNamedValueSet** object contained in the variable called **objWMIAsyncContext** passed as a parameter of the sink routine (see lines 51, 67, 76, and 95). In the previous chapter, in Figure 4.5 (number 36), we see that the **SWbemNamedValueSet** object is a collection that can be created using its progID with the **CreateObject** statement or any other equivalent technique (see Figure 4.2).

The question is: How do we define a context when invoking the asynchronous method? Sample 5.4 shows the same code as Sample 5.3, except that some additions were made (in boldface) to include this notion of context.

Sample 5.4 *Setting asynchronously with a context one read/write property of a Win32_Registry class instance*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMISinkContext" reference="true"/>
15:
16:    <script language="VBscript">
17:    <![CDATA[
..:
25:    ' ---------------------------------------------------------------------------
26:    Const cComputerName = "LocalHost"
27:    Const cWMINameSpace = "root/cimv2"
28:    Const cWMIClass = "Win32_Registry"
29:    Const cWMIInstance = "Microsoft Windows 2000 Server|C:\WINNT|\Device\Harddisk0\Partition1"
..:
34:    Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
35:
36:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
37:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
38:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
39:
40:    objWMISinkContext.Add "WMIMethod", "GetAsync"
41:    objWMIServices.GetAsync objWMISink, _
42:                            cWMIClass & "='" & cWMIInstance & "'", _
43:                            , _
44:                            , _
```

```
45:                          objWMISinkContext
46:
47:    WScript.Echo "Continuing script execution ..."
48:    PauseScript "Click on 'Ok' to terminate the script ..."
49:
50:    objWMISink.Cancel
..:
56:    ' ------------------------------------------------------------------------------
57:    Sub SINK_OnCompleted (intHResult, objWMILastError, objWMIAsyncContext)
..:
64:        Set objContextItem = objWMIAsyncContext.Item ("WMIMethod")
65:
66:        If intHResult = 0 Then
67:           WScript.Echo "'" & objContextItem.Value & _
68:                        "' WMI Scripting API Asynchronous call successful."
69:        Else
70:           WScript.Echo "'" & objContextItem.Value & _
71:                        "' WMI Scripting API Asynchronous call failed."
72:        End If
..:
76:    End Sub
77:
78:    ' ------------------------------------------------------------------------------
79:    Sub SINK_OnObjectPut (objWMIPath, objWMIAsyncContext)
..:
85:    End Sub
86:
87:    ' ------------------------------------------------------------------------------
88:    Sub SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
..:
92:        Wscript.Echo
93:        Wscript.Echo "BEGIN - OnObjectReady."
94:
95:        Set objWMISinkContext = CreateObject ("WbemScripting.SWbemNamedValueSet")
96:
97:        WScript.Echo objWMIInstance.Name & " (" & objWMIInstance.Description & ")"
98:
99:        Wscript.Echo "Current registry size is: " & objWMIInstance.ProposedSize  & " MB."
100:
101:       objWMIInstance.ProposedSize = objWMIInstance.ProposedSize + 10
102:
103:       objWMISinkContext.Add "WMIMethod", "PutAsync"
104:       objWMIInstance.PutAsync_ objWMISink, _
105:                                wbemChangeFlagUpdateOnly Or wbemFlagReturnWhenComplete, _
106:                                , _
107:                                objWMISinkContext
108:
109:       Wscript.Echo "Current registry size is now: " & objWMIInstance.ProposedSize  & " MB."
110:
111:       Set objWMISinkContext = Nothing
112:
113:       Wscript.Echo "END   - OnObjectReady."
114:
115:    End Sub
116:
117:    ' ------------------------------------------------------------------------
118:    Sub SINK_OnProgress (intUpperBound, intCurrent, strMessage, objWMIAsyncContext)
...:
124:    End Sub
125:
```

```
126:    ' ----------------------------------------------------------------------------
127:    Function PauseScript (strMessage)
...:
136:    End Function
137:
138:    ]]>
139:    </script>
140:  </job>
141:</package>
```

The first change is the creation of an **SWbemNamedObjectSet** object (line 14). This object is used as a parameter when invoking the *GetAsync* method (lines 41 to 45). At line 40, the **SWbemNamedObjectSet** object is initialized. Since this object is a collection, the script uses the *Add* method to add an item to the collection. The first parameter is the name of the value to be put in the collection; the second parameter is the value itself. Once the *GetAsync* asynchronous method is completed, the SINK_OnCompleted() sink routine is called. The sink routine provides the context information that comes from the **SWbemNamedObjectSet** collection. The first step is to retrieve the value initially provided. For this, the script retrieves the value by referencing the value name in the item list (line 64). Once the name of the value has been retrieved, it is possible for the sink routine to display the value (lines 67 and 70). Since the value is set to the name of the asynchronous method (line 40), the output will display the name of the asynchronous method.

The SINK_OnObjectReady() sink routine performs the same process to create context information. But this time, it is for the *PutAsync_* asynchronous method. Here, a new **SWbemNamedValueSet** object is created (line 95), initialized with the name of the asynchronous method to be executed (line 103) and passed as a parameter when the asynchronous method is invoked (lines 104 to 107). Note that the name of the value is the same as before (WMIMethod), the SINK_OnCompleted() sink routine searches for a WMIMethod value name (line 64), and the value name provided in the context must remain the same. It is possible to use a different value name, but then the SINK_OnCompleted() sink routine has to check two value names, which complicates the logic to determine the context. Why create a new **SWbemNamedValueSet** object in the SINK_OnObjectReady() sink routine? If the script didn't create a new object, the existing object created in the core of the script (line 14) would be used, because it is a global variable; instead of creating a new context object, the script adds one value name to the existing values in the collection. Doing so causes one **SWbemNamedValueSet** object to contain both contexts' information. Again, it is more difficult to differentiate the context during the SINK_OnCompleted() sink routine execution. The display output of Sample 5.4 is shown in Figure 5.3.

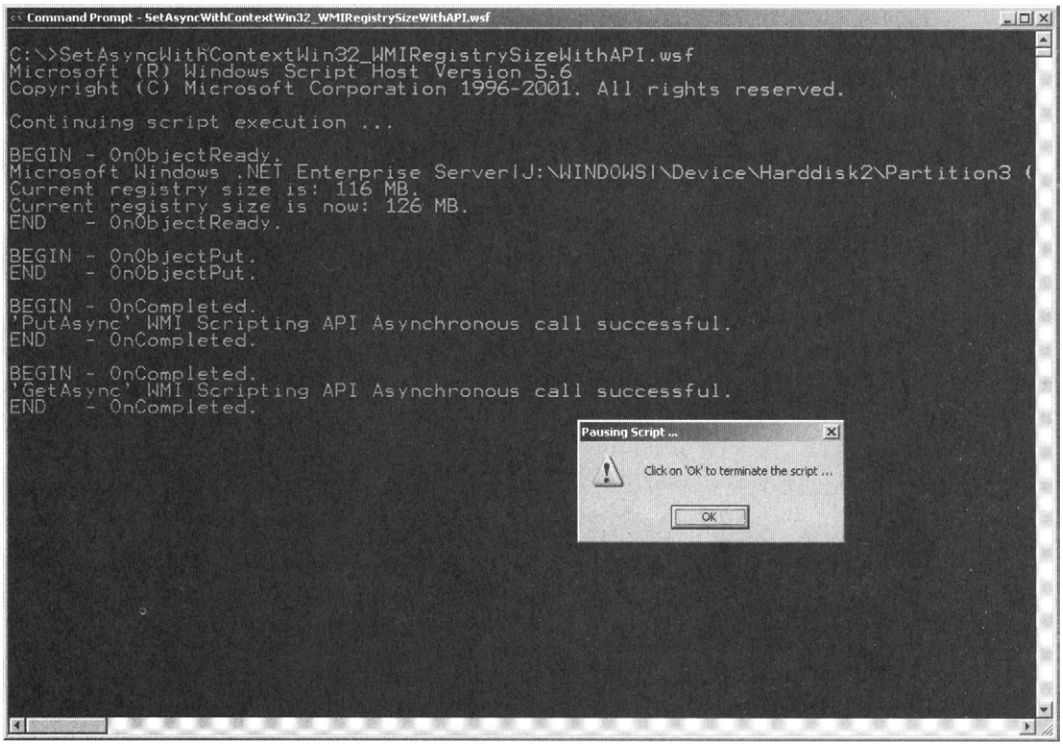


Figure 5.3 *Asynchronously setting one read/write property of a Win32_Registry class instance directly with a context.*

5.2.4 Executing CIM instances methods

We have seen how to retrieve and modify the WMI instance properties asynchronously. Since some of the WMI classes expose methods to manage real world entities, it is also possible to execute CIM class methods asynchronously.

5.2.4.1 Executing CIM instances methods without context

Sample 4.18 ("Using one method of a Win32_Service instance indirectly") executes the *StartService* and *StopService* method of the *Win32_Service* class indirectly. If the service stops, the script starts the service and vice versa. Both methods execute synchronously. An important point to note about the method invocation (see Sample 4.18) is that the invocation is indirect. Therefore, the method to be executed is passed as a parameter to a WMI Scripting API object method and is not hard coded in the script code itself.

The indirect execution of the method is the base foundation on which to execute methods asynchronously. To implement the functionality of Sample 4.18 asynchronously, the script uses the *ExecMethodAsync_* of an **SWbemObject**. The logic used is implemented in Sample 5.5.

Sample 5.5 *Using one method of a Win32_Service instance asynchronously*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
24:    ' ---------------------------------------------------------------------------------------
25:    Const cComputerName = "LocalHost"
26:    Const cWMINameSpace = "root/cimv2"
27:    Const cWMIClass = "Win32_Service"
28:    Const cWMIInstance = "SNMP"
..:
34:    Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
35:
36:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
37:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
38:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
39:
40:    Set objWMIInstance = objWMIServices.Get(cWMIClass & "='" & cWMIInstance & "'")
41:
42:    WScript.Echo vbCRLF & "'" & objWMIInstance.DisplayName & "' is currently " & _
43:                 LCase (objWMIInstance.State) & "."
44:
45:    If objWMIInstance.State = "Running" Then
46:       objWMIInstance.ExecMethodAsync_ objWMISink, "StopService"
47:    End If
48:    If objWMIInstance.State = "Stopped" Then
49:       objWMIInstance.ExecMethodAsync_ objWMISink, "StartService"
50:    End If
51:
52:    WScript.Echo "Continuing script execution ..."
53:    PauseScript "Click on 'Ok' to terminate the script ..."
54:
55:    objWMISink.Cancel
..:
63:    ' ---------------------------------------------------------------------------------------
64:    Sub SINK_OnCompleted (intHResult, objWMILastError, objWMIAsyncContext)
65:
66:        Wscript.Echo
67:        Wscript.Echo "BEGIN - OnCompleted."
68:
69:        If intHResult = 0 Then
70:           WScript.Echo "WMI Scripting API Asynchronous call successful."
71:        Else
72:           WScript.Echo "WMI Scripting API Asynchronous call failed."
73:        End If
```

```
 74:
 75:         Wscript.Echo "END   - OnCompleted."
 76:
 77:     End Sub
 78:
 79:     ' ------------------------------------------------------------------------------
 80:     Sub SINK_OnObjectPut (objWMIPath, objWMIAsyncContext)
...:
 86:     End Sub
 87:
 88:     ' ------------------------------------------------------------------------------
 89:     Sub SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
 90:
 91:         Wscript.Echo
 92:         Wscript.Echo "BEGIN - OnObjectReady."
 93:
 94:         If objWMIInstance.ReturnValue = 0 Then
 96:         Else
 97:            Wscript.Echo "Asynchronous method execution failed."
 98:         End If
 99:
100:         Wscript.Echo "END   - OnObjectReady."
101:
102:     End Sub
103:
104:     ' ------------------------------------------------------------------------------
105:     Sub SINK_OnProgress (intUpperBound, intCurrent, strMessage, objWMIAsyncContext)
...:
111:     End Sub
112:
113:     ' ------------------------------------------------------------------------------
114:     Function PauseScript (strMessage)
...:
123:     End Function
124:
125:     ]]>
126:     </script>
127:   </job>
128:</package>
```

The logic used to invoke an asynchronous process is the same as before. A WMI connection must be established (lines 36 to 38), and an **SWbemSink** object referencing the sink routines must be created (line 34). Note that the object instance of the SNMP *Win32_Service* is instantiated synchronously (line 40). Although it was possible to create the instance asynchronously, to avoid any confusion with the asynchronous method execution in the sink routine, it was decided to make the instantiation synchronously. This makes the script easier to understand, given that this is the first time that we are examining an asynchronous method execution. Next, the script tests the state of the service to determine whether the service must be stopped or started. This is where the asynchronous method invocation is performed (lines 45 to 47 and 48 to 50).

The particularity of an asynchronous method execution is the place where the result of the method execution is retrieved. Before, when using the *GetAsync* method, the instantiated **SWbemObject** was available in the SINK_OnObjectReady() sink routine. Now, with the *ExecMethodAsync_* method, the return code of the method invocation is also available in the SINK_OnObjectReady() sink routine. If you remember, in Sample 4.18 ("Using one method of a Win32_Service instance indirectly"), the returned result of the indirect method execution was available in an **SWbemObject** whose properties return the result of the executed method. It is the same when executing the method asynchronously except that the **SWbemObject** containing the result of the method execution is obtained in the SINK_OnObjectReady() sink routine. This is why the script tests the *ReturnValue* property (line 94) returned from the **SWbemObject** provided as a parameter of the SINK_OnObjectReady() sink routine.

To avoid any confusion in the sink routine, the object instantiation is synchronous (line 40), but the SINK_OnObjectReady() sink routine does not know whether the result of the asynchronous method execution is for the *StartService* or the *StopService*. So, by sharing the same set of sink routines, determining the context of the sink routine call remains ambiguous, especially for those actions that must be performed when the service is stopped. Of course, it depends on the purpose of the script and the type of operation to be performed in the sink routine. Let's see how we can clear up these ambiguities.

5.2.4.2 Executing CIM instances methods with context

If the previous Sample 5.5 was creating the SNMP *Win32_Service* instance with the *GetAsync* method (instead of the *Get* method) and used the same **SWbemSink** object for executing the method asynchronously, then the SINK_OnObjectReady() was called twice: once for the instantiation of the *Win32_Service* and once for execution of the method. Again, we see the real importance of the sink routine context, because the **SWbemObject** retrieved is the *Win32_Service* instance during the first sink routine and the result of the *StartService* or *StopService* method execution during the second sink routine. As previously shown, all sink functions use a parameter represented by the variable **objWMIAsyncContext**, which is an **SWbemNamedValueSet** object (number 36 in Figure 4.5).

Sample 5.6 performs the same logic as Sample 5.5 except that the notion of context has been added to differentiate the instance creation (which is asynchronous now) from the asynchronous method execution.

Sample 5.6 *Using one method of a Win32_Service instance asynchronously with context*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMIMethodSinkContext" reference="true"/>
15:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMIInstanceSinkContext" reference="true"/>
16:
17:    <script language="VBscript">
18:    <![CDATA[
..:
26:    ' ---------------------------------------------------------------------------------------
27:    Const cComputerName = "LocalHost"
28:    Const cWMINameSpace = "root/cimv2"
29:    Const cWMIClass = "Win32_Service"
30:    Const cWMIInstance = "SNMP"
..:
35:    Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
36:
37:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
38:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
39:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
40:
41:    objWMIInstanceSinkContext.Add "WMIMethod", "GetAsync"
42:    objWMIServices.GetAsync objWMISink, _
43:                            cWMIClass & "='" & cWMIInstance & "'", _
44:                            , _
45:                            , _
46:                            objWMIInstanceSinkContext
47:
48:    WScript.Echo "Continuing script execution ..."
49:    PauseScript "Click on 'Ok' to terminate the script ..."
50:
51:    objWMISink.Cancel
..:
58:    ' ---------------------------------------------------------------------------------------
59:    Sub SINK_OnCompleted (intHResult, objWMILastError, objWMIAsyncContext)
..:
63:        Wscript.Echo
64:        Wscript.Echo "BEGIN - OnCompleted."
65:
66:        Set objContextItem = objWMIAsyncContext.Item ("WMIMethod")
67:
68:        If intHResult = 0 Then
69:            WScript.Echo "'" & objContextItem.Value & _
70:                         "' WMI Scripting API Asynchronous call successful."
71:        Else
72:            WScript.Echo "'" & objContextItem.Value & _
73:                         "' WMI Scripting API Asynchronous call failed."
74:        End If
..:
78:        Wscript.Echo "END   - OnCompleted."
79:
80:    End Sub
```

```
 81:
 82:    ' ------------------------------------------------------------------------------------
 83:    Sub SINK_OnObjectPut (objWMIPath, objWMIAsyncContext)
 ..:
 89:    End Sub
 90:
 91:    ' ------------------------------------------------------------------------------------
 92:    Sub SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
 ..:
 96:        Wscript.Echo
 97:        Wscript.Echo "BEGIN - OnObjectReady."
 98:
 99:        Set objContextItem = objWMIAsyncContext.Item ("WMIMethod")
100:
101:        Select Case objContextItem.Value
102:               Case "GetAsync"
103:                    WScript.Echo "'" & objWMIInstance.DisplayName & "' is currently " & _
104:                                LCase (objWMIInstance.State) & "."
105:
106:                    If objWMIInstance.State = "Running" Then
107:                       objWMIMethodSinkContext.Add "WMIMethod", "ExecMethodAsync"
108:                       objWMIMethodSinkContext.Add "CIMMethod", "StopService"
109:                       objWMIInstance.ExecMethodAsync_ objWMISink, _
110:                                                       "StopService", _
111:                                                       , _
112:                                                       , _
113:                                                       , _
114:                                                       objWMIMethodSinkContext
115:                    End If
116:                    If objWMIInstance.State = "Stopped" Then
117:                       objWMIMethodSinkContext.Add "WMIMethod", "ExecMethodAsync"
118:                       objWMIMethodSinkContext.Add "CIMMethod", "StartService"
119:                       objWMIInstance.ExecMethodAsync_ objWMISink, _
120:                                                       "StartService", _
121:                                                       , _
122:                                                       , _
123:                                                       , _
124:                                                       objWMIMethodSinkContext
125:                    End If
126:               Case "ExecMethodAsync"
127:                    Set objContextItem = objWMIAsyncContext.Item ("CIMMethod")
128:                    If objWMIInstance.ReturnValue = 0 Then
129:                       Wscript.Echo "'" & objContextItem.Value & _
130:                                    "' method execution successful."
131:                    Else
132:                       Wscript.Echo "'" & objContextItem.Value & _
133:                                    "' method execution failed."
134:                    End If
135:        End Select
...:
139:        Wscript.Echo "END   - OnObjectReady."
140:
141:    End Sub
142:
143:    ' ------------------------------------------------------------------------------------
144:    Sub SINK_OnProgress (intUpperBound, intCurrent, strMessage, objWMIAsyncContext)
...:
150:    End Sub
151:
152:    ' ------------------------------------------------------------------------------------
```

```
153:    Function PauseScript (strMessage)
...:
164:    ]]>
165:    </script>
166:  </job>
167:</package>
```

Sample 5.6 uses two **SWbemNamedObjectSet** objects, one for each context. The first is initialized (lines 15 and 41) for the *GetAsync* asynchronous method execution of the **SWbemServices** object (lines 42 to 46). This context uses a named variable called "WMIMethod" with a value equal to the method name *GetAsync*. When the *GetAsync* method is executed, this context is tested (lines 101, 102 and 126 of the **Select Case** statement) in the SINK_OnObjectReady() sink routine to determine if the current context relates to the instance creation (*GetAsync*) or the method execution (*ExecMethodAsync_*).

When the instance creation is completed, the SINK_OnCompleted() sink routine is called (lines 59 to 80). As the context is defined, the routine retrieves the named value WMIMethod (line 66) to display its value (lines 69 or 72) based on the result of the asynchronous call (line 68).

During the execution of the SINK_OnObjectReady() sink routine the script tests the state of the *Win32_Service* (lines 106 and 116) to determine whether the service must be stopped or started. In both cases, the script prepares the context of the *ExecMethodAsync_* asynchronous method. For this, the script uses the second **SWbemNamedValueSet** object created (line 14). Before the asynchronous method invocation, the script initializes two **SWbemNamedValue** objects. One contains the name of the WMI scripting API object method invoked—*ExecMethodAsync_* (lines 107 and 117)—and the other contains the *Win32_Service* method executed (line 108 or 118). Once the **SWbemNamedValueSet** object initializes, the *ExecMethodAsync_* is invoked (lines 109 and 119).

Once the asynchronous method executes, the SINK_OnObjectReady() sink routine is called again but this time in a different context. As the SINK_OnObjectReady() sink routine starts by retrieving the WMIMethod named value (line 99), it determines that the context is now related to the *ExecMethodAsync_* execution. In this case, the script branches to the statement at line 127. This statement retrieves the second named value called "CIMMethod." This named value contains the name of the CIM method executed asynchronously and defined during a preceding SINK_OnObjectReady() sink routine call (lines 108 or 118). As usual, the asynchronous method execution finishes with a call to the SINK_OnCompleted() sink routine. As before, the named value WMIMethod is retrieved, and

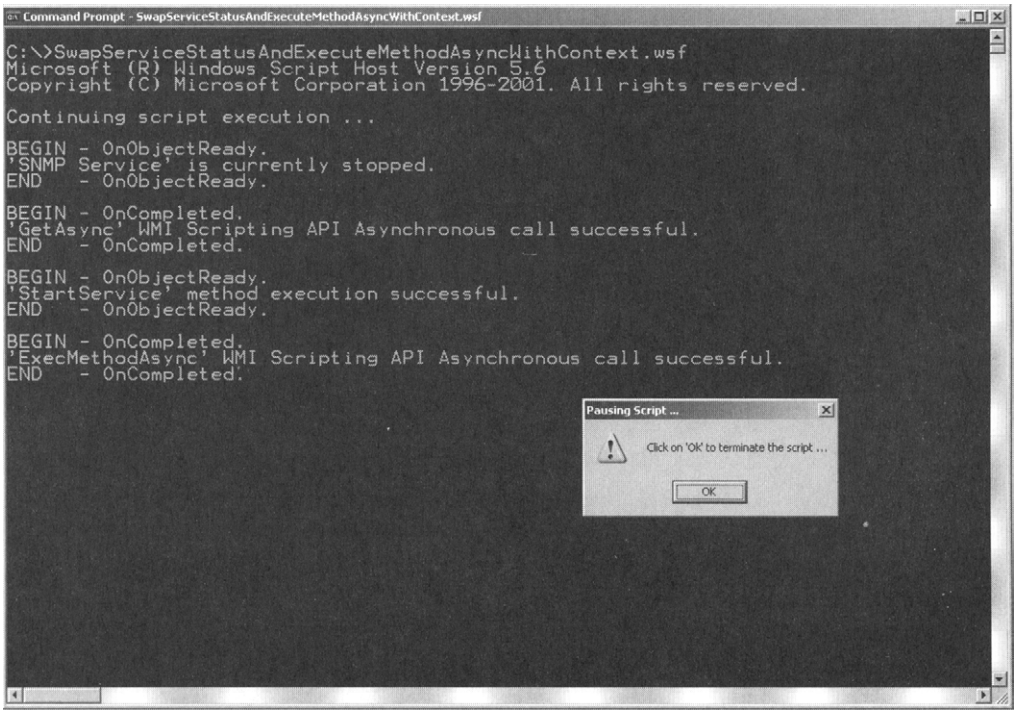


Figure 5.4 *The GetAsync method and ExecMethodAsync_ method execution output with context.*

now the context of this sink routine call relates to the *ExecMethodAsync_* method. The display output of the script is visible on Figure 5.4.

5.2.4.3 Executing CIM instances methods with context and parameters

Sample 5.6 is a full asynchronous and contextual script, but it does not use any parameters when invoking the *StartService* or *StopService* methods. When Sample 4.19 ("Using one method of a Win32_Service instance indirectly with one parameter") changes the startup mode of a *Win32_Service* instance, it uses the startup mode of the service as a parameter. This sample executes the method indirectly and uses a specific tactic to pass the method parameter. By combining the examined Sample 5.6 with Sample 4.19, it is possible to perform the same operation asynchronously with a full context determination. Only one routine in Sample 5.6 must be adapted to suit the need of the required input parameter. This routine is the SINK_OnObjectReady() sink routine shown in Sample 5.7. The rest of the script is exactly the same as Sample 5.6.

Sample 5.7 *Using one method with parameter of a Win32_Service instance asynchronously with context*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
..:
..:
91:    ' ---------------------------------------------------------------------------------------
92:    Sub SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
..:
98:        Wscript.Echo
99:        Wscript.Echo "BEGIN - OnObjectReady."
100:
101:       Set objContextItem = objWMIAsyncContext.Item ("WMIMethod")
102:
103:       Select Case objContextItem.Value
104:              Case "GetAsync"
105:                   WScript.Echo "'" & objWMIInstance.DisplayName & _
106:                                "' startup is currently " & _
107:                                objWMIInstance.StartMode & "."
108:
109:                   Set objWMIMethod = objWMIInstance.Methods_("ChangeStartMode")
110:                   Set objWMIInParameters = objWMIMethod.InParameters
111:
112:                   If objWMIInstance.StartMode = "Manual" Then
113:                      objWMIMethodSinkContext.Add "WMIMethod", "ExecMethodAsync"
114:                      objWMIMethodSinkContext.Add "CIMMethod", "ChangeStartMode (Automatic)"
115:                      objWMIInParameters.Properties_.Item("StartMode") = "Automatic"
116:                      objWMIInstance.ExecMethodAsync_ objWMISink, _
117:                                                      "ChangeStartMode", _
118:                                                      objWMIInParameters, _
119:                                                      , _
120:                                                      , _
121:                                                      objWMIMethodSinkContext
122:                   End If
123:                   If objWMIInstance.StartMode = "Auto" Then
124:                      objWMIMethodSinkContext.Add "WMIMethod", "ExecMethodAsync"
125:                      objWMIMethodSinkContext.Add "CIMMethod", "ChangeStartMode (Manual)"
126:                      objWMIInParameters.Properties_.Item("StartMode") = "Manual"
127:                      objWMIInstance.ExecMethodAsync_ objWMISink, _
128:                                                      "ChangeStartMode", _
129:                                                      objWMIInParameters, _
130:                                                      , _
131:                                                      , _
132:                                                      objWMIMethodSinkContext
133:                   End If
134:
135:                   Set objWMIInParameters = Nothing
136:                   Set objWMIMethod = Nothing
137:
138:             Case "ExecMethodAsync"
139:                   Set objContextItem = objWMIAsyncContext.Item ("CIMMethod")
140:                   If objWMIInstance.ReturnValue = 0 Then
141:                      Wscript.Echo "'" & objContextItem.Value & _
142:                                   "' method execution successful."
```

```
143:                    Else
144:                       Wscript.Echo "'" & objContextItem.Value & _
145:                                    "' method execution failed."
146:                    End If
147:        End Select
...:
151:        Wscript.Echo "END   - OnObjectReady."
152:
153:    End Sub
...:
...:
...:
176:    ]]>
177:    </script>
178:  </job>
179:</package>
```

From the point of view of context definitions and usages, the script uses the same logic as Sample 5.6. The new detail is the creation of the **SWbemObject** object targeted to carry the input parameter of the asynchronously executed method (line 110). In this sink routine, we recognize the indirect method execution with one parameter assignment as used in Sample 4.19. Depending on the current startup mode of the *Win32_Service* (line 112 and 123), the required parameter for the asynchronous method is set at line 115 or 126. Basically, Sample 5.7 is the same as Sample 4.19, but the task is executed asynchronously.

5.3 Error handling with WMI scripting

Up to now, all script samples shown in this book contain very little error management. This choice was made to ease the reading and understanding of the script principles. Although this approach is valid for an academic purpose, developing a script for a production environment is done quite differently. Whatever the programming language used, managing errors in a code is always a tricky problem because the software must include a lot of exceptions and error management that represent a supplemental logic added to the core logic. Working with WMI is no exception; a specific error-management technique must be included in the application. When scripting with WMI, there are different error types. We can classify these errors into two categories: run-time errors and WMI operational errors.

Let's take a look at the first category, the run-time errors. These errors stop the script execution when they occur. To avoid such a situation, in VBScript these errors must be processed with the **On Error Resume Next** statement combined with a test of the *Err.number* value. When *Err.number* is not equal to zero, it means that an error has occurred. With the **Err** object from VBScript, it is also possible to retrieve error messages by using the

expression *Err.Description*. With JScript, error handling must be processed with the **Try ... catch ... finally** statement. We will first use a VBScript sample and next, at the end of this section, we will review the same script logic written with JScript to complete the picture. A WMI script handles the following run-time errors:

- **Script language errors.** These errors are typically present during the development phase. Once corrected, these types of errors should no longer be present. Consequently, the script logic does not have to handle this situation.
- **Win32 errors that have a number starting with 0x8007xxxx.** These errors are not directly related to WMI. If error management is missing in the script (maybe because it is under development), with this error category it is possible to retrieve the associated message with the **NET.exe** command-line utility. For this, you must take the last four digits and convert them to a decimal value. Next, you must execute the **NET.exe** tool. For example, if you get an error code of 0x800706BA, you would take the last four digits, 0x06BA; convert the value to a decimal value: 1722; and execute the following command line:

```
C:>Net HelpMsg 1722
The RPC server is unavailable.
```

- **WMI errors that have a number starting with 0x8004xxxx.** These errors are purely WMI run-time errors. The errors are part of the **wbemErrorEnum** constants list. This list of WMI run-time errors is shown in Table 5.2.
- **Any other errors related to any other object model, such as WSH, CDOEX, CDOEXM, and ADSI, if the script makes use of these objects.** In this case, please refer to the related object model documentation for any specific error codes.

Errors of the second category, consisting of WMI operational errors, are due to mistakes or problems when performing WMI operations. These errors are not run-time errors but are values returned by WMI method execution. In this case, we must distinguish the following:

- **Errors caused by the execution of methods exposed by the WMI scripting API objects.** Depending on the method execution context (synchronous or asynchronous), the error will be retrieved differently. If the method is executed synchronously, the error will be retrieved as a run-time error (i.e., in the VBScript **Err** object). However, by using the **SWbemLastError** object from WMI, more information can be

Table 5.2 *The wbemErrorEnum Values*

wbemNoErr		0x00000000	
wbemErrFailed	WBEM_E_FAILED	0x80041001	The call failed.
wbemErrNotFound	WBEM_E_NOT_FOUND	0x80041002	The object could not be found.
wbemErrAccessDenied	WBEM_E_ACCESS_DENIED	0x80041003	The current user does not have permission to perform the action.
wbemErrProviderFailure	WBEM_E_PROVIDER_FAILURE	0x80041004	The provider has failed at some time other than during initialization.
wbemErrTypeMismatch	WBEM_E_TYPE_MISMATCH	0x80041005	A type mismatch occurred.
wbemErrOutOfMemory	WBEM_E_OUT_OF_MEMORY	0x80041006	There was not enough memory for the operation.
wbemErrInvalidContext	WBEM_E_INVALID_CONTEXT	0x80041007	The IWbemContext object is not valid.
wbemErrInvalidParameter	WBEM_E_INVALID_PARAMETER	0x80041008	One of the parameters to the call is not correct.
wbemErrNotAvailable	WBEM_E_NOT_AVAILABLE	0x80041009	The resource
wbemErrCriticalError	WBEM_E_CRITICAL_ERROR	0x8004100a	An internal
wbemErrInvalidStream	WBEM_E_INVALID_STREAM	0x8004100b	One or more network packets were corrupted during a remote session.
wbemErrNotSupported	WBEM_E_NOT_SUPPORTED	0x8004100c	The feature or operation is not supported.
wbemErrInvalidSuperclass	WBEM_E_INVALID_SUPERCLASS	0x8004100d	The superclass specified is not valid.
wbemErrInvalidNamespace	WBEM_E_INVALID_NAMESPACE	0x8004100e	The namespace specified could not be found.
wbemErrInvalidObject	WBEM_E_INVALID_OBJECT	0x8004100f	The specified instance is not valid.
wbemErrInvalidClass	WBEM_E_INVALID_CLASS	0x80041010	The specified class is not valid.
wbemErrProviderNotFound	WBEM_E_PROVIDER_NOT_FOUND	0x80041011	A provider referenced in the schema does not have a corresponding registration.
wbemErrInvalidProviderRegistration	WBEM_E_INVALID_PROVIDER_REGISTRATION	0x80041012	A provider referenced in the schema has an incorrect or incomplete registration. This error may be caused by any of the following A missing pragma namespace command in the MOF file used to register the provider
wbemErrProviderLoadFailure		0x80041013	

Table 5.2 *The wbemErrorEnum Values (continued)*

wbemErrInitializationFailure	WBEM_E_INITIALIZATION_FAILURE	0x80041014	A component
wbemErrTransportFailure	WBEM_E_TRANSPORT_FAILURE	0x80041015	A networking error that prevents normal operation has occurred.
wbemErrInvalidOperation	WBEM_E_INVALID_OPERATION	0x80041016	The requested operation is not valid. This error usually applies to invalid attempts to delete classes or properties.
wbemErrInvalidQuery	WBEM_E_INVALID_QUERY	0x80041017	The query was not syntactically valid.
wbemErrInvalidQueryType	WBEM_E_INVALID_QUERY_TYPE	0x80041018	The requested query language is not supported.
wbemErrAlreadyExists	WBEM_E_ALREADY_EXISTS	0x80041019	In a put operation
wbemErrOverrideNotAllowed	WBEM_E_OVERRIDE_NOT_ALLOWED	0x8004101a	It is not possible to perform the add operation on this qualifier because the owning object does not permit overrides.
wbemErrPropagatedQualifier	WBEM_E_PROPAGATED_QUALIFIER	0x8004101b	The user attempted to delete a qualifier that was not owned. The qualifier was inherited from a parent class.
wbemErrPropagatedProperty	WBEM_E_PROPAGATED_PROPERTY	0x8004101c	The user attempted to delete a property that was not owned. The property was inherited from a parent class.
wbemErrUnexpected	WBEM_E_UNEXPECTED	0x8004101d	The client made an unexpected and illegal sequence of calls
wbemErrIllegalOperation	WBEM_E_ILLEGAL_OPERATION	0x8004101e	The user requested an illegal operation
wbemErrCannotBeKey	WBEM_E_CANNOT_BE_KEY	0x8004101f	There was an illegal attempt to specify a key qualifier on a property that cannot be a key. The keys are specified in the class definition for an object
wbemErrIncompleteClass	WBEM_E_INCOMPLETE_CLASS	0x80041020	The current object is not a valid class definition. Either it is incomplete
wbemErrInvalidSyntax	WBEM_E_INVALID_SYNTAX	0x80041021	Reserved for future use.
wbemErrNondecoratedObject	WBEM_E_NONDECORATED_OBJECT	0x80041022	Reserved for future use.
wbemErrReadOnly	WBEM_E_READ_ONLY	0x80041023	The property that you are attempting to modify is read-only.

Table 5.2 *The wbemErrorEnum Values (continued)*

wbemErrProviderNotCapable	WBEM_E_PROVIDER_NOT_CAPABLE	0x80041024	The provider cannot perform the requested operation. This would include a query that is too complex
wbemErrClassHasChildren	WBEM_E_CLASS_HAS_CHILDREN	0x80041025	An attempt was made to make a change that would invalidate a subclass.
wbemErrClassHasInstances	WBEM_E_CLASS_HAS_INSTANCES	0x80041026	An attempt has been made to delete or modify a class that has instances.
wbemErrQueryNotImplemented	WBEM_E_QUERY_NOT_IMPLEMENTED	0x80041027	Reserved for future use.
wbemErrIllegalNull	WBEM_E_ILLEGAL_NULL	0x80041028	A value of Nothing was specified for a property that may not be Nothing
wbemErrInvalidQualifierType	WBEM_E_INVALID_QUALIFIER_TYPE	0x80041029	The variant value for a qualifier was provided that is not of a legal qualifier type.
wbemErrInvalidPropertyType	WBEM_E_INVALID_PROPERTY_TYPE	0x8004102a	The CIM type specified for a property is not valid.
wbemErrValueOutOfRange	WBEM_E_VALUE_OUT_OF_RANGE	0x8004102b	The request was made with an out-of-range value
wbemErrCannotBeSingleton	WBEM_E_CANNOT_BE_SINGLETON	0x8004102c	An illegal attempt was made to make a class singleton
wbemErrInvalidCimType	WBEM_E_INVALID_CIM_TYPE	0x8004102d	The CIM type specified is not valid.
wbemErrInvalidMethod	WBEM_E_INVALID_METHOD	0x8004102e	The requested method is not available.
wbemErrInvalidMethodParameters	WBEM_E_INVALID_METHOD_PARAMETERS	0x8004102f	The parameters provided for the method are not valid.
wbemErrSystemProperty	WBEM_E_SYSTEM_PROPERTY	0x80041030	There was an attempt to get qualifiers on a system property.
wbemErrInvalidProperty	WBEM_E_INVALID_PROPERTY	0x80041031	The property type is not recognized.
wbemErrCallCancelled	WBEM_E_CALL_CANCELLED	0x80041032	An asynchronous process has been canceled internally or by the user. Note that due to the timing and nature of the asynchronous operation the operation may not have been truly canceled.
wbemErrShuttingDown	WBEM_E_SHUTTING_DOWN	0x80041033	The user has requested an operation while WMI is in the process of shutting down.

Table 5.2 *The wbemErrorEnum Values (continued)*

wbemErrPropagatedMethod	WBEM_E_PROPAGATED_METHOD	0x80041034	An attempt was made to reuse an existing method name from a superclass
wbemErrUnsupportedParameter	WBEM_E_UNSUPPORTED_PARAMETER	0x80041035	One or more parameter values
wbemErrMissingParameter	WBEM_E_MISSING_PARAMETER_ID	0x80041036	A parameter was missing from the method call.
wbemErrInvalidParameterId	WBEM_E_INVALID_PARAMETER_ID	0x80041037	A method parameter has an invalid ID qualifier.
wbemErrNonConsecutiveParameterIds	WBEM_E_NONCONSECUTIVE_PARAMETER_IDS	0x80041038	One or more of the method parameters have ID qualifiers that are out of sequence.
wbemErrParameterIdOnRetval	WBEM_E_PARAMETER_ID_ON_RETVAL	0x80041039	The return value for a method has an ID qualifier.
wbemErrInvalidObjectPath	WBEM_E_INVALID_OBJECT_PATH	0x8004103a	The specified object path was invalid.
wbemErrOutOfDiskSpace	WBEM_E_OUT_OF_DISK_SPACE	0x8004103b	There is not enough free disk space to continue the operation.
wbemErrBufferTooSmall	WBEM_E_BUFFER_TOO_SMALL	0x8004103c	The supplied buffer was too small to hold all the objects in the enumerator or to read a string property.
wbemErrUnsupportedPutExtension	WBEM_E_UNSUPPORTED_PUT_EXTENSION	0x8004103d	The provider does not support the requested put operation.
wbemErrUnknownObjectType	WBEM_E_UNKNOWN_OBJECT_TYPE	0x8004103e	An object with an incorrect type or version was encountered during marshaling.
wbemErrUnknownPacketType	WBEM_E_UNKNOWN_PACKET_TYPE	0x8004103f	A packet with an incorrect type or version was encountered during marshaling.
wbemErrMarshalVersionMismatch	WBEM_E_MARSHAL_VERSION_MISMATCH	0x80041040	The packet has an unsupported version.
wbemErrMarshalInvalidSignature	WBEM_E_MARSHAL_INVALID_SIGNATURE	0x80041041	The packet appears to be corrupted.
wbemErrInvalidQualifier	WBEM_E_INVALID_QUALIFIER	0x80041042	An attempt has been made to mismatch qualifiers
wbemErrInvalidDuplicateParameter	WBEM_E_INVALID_DUPLICATE_PARAMETER	0x80041043	A duplicate parameter has been declared in a CIM method.
wbemErrTooMuchData	WBEM_E_TOO_MUCH_DATA	0x80041044	Reserved for future use.
wbemErrServerTooBusy	WBEM_E_SERVER_TOO_BUSY	0x80041045	A call to IWbemObjectSink::Indicate has failed. The provider may choose to refire the event.

Table 5.2 *The wbemErrorEnum Values (continued)*

wbemErrInvalidFlavor	WBEM_E_INVALID_FLAVOR	0x80041046	The specified flavor was invalid.
wbemErrCircularReference	WBEM_E_CIRCULAR_REFERENCE	0x80041047	An attempt has been made to create a reference that is circular (for example
wbemErrUnsupportedClassUpdate	WBEM_E_UNSUPPORTED_CLASS_UPDATE	0x80041048	The specified class is not supported.
wbemErrCannotChangeKeyInheritance	WBEM_E_CANNOT_CHANGE_KEY_INHERITANCE	0x80041049	An attempt was made to change a key when instances or subclasses are already using the key.
wbemErrCannotChangeIndexInheritance	WBEM_E_CANNOT_CHANGE_INDEX_INHERITANCE	0x80041050	An attempt was made to change an index when instances or subclasses are already using the index.
wbemErrTooManyProperties	WBEM_E_TOO_MANY_PROPERTIES	0x80041051	An attempt was made to create more properties than the current version of the class supports.
wbemErrUpdateTypeMismatch	WBEM_E_UPDATE_TYPE_MISMATCH	0x80041052	A property was redefined with a conflicting type in a derived class.
wbemErrUpdateOverrideNotAllowed	WBEM_E_UPDATE_OVERRIDE_NOT_ALLOWED	0x80041053	An attempt was made in a derived class to override a non-overrideable qualifier.
wbemErrUpdatePropagatedMethod	WBEM_E_UPDATE_PROPAGATED_METHOD	0x80041054	A method was redeclared with a conflicting signature in a derived class.
wbemErrMethodNotImplemented	WBEM_E_METHOD_NOT_IMPLEMENTED	0x80041055	An attempt was made to execute a method not marked with [implemented] in any relevant class.
wbemErrMethodDisabled	WBEM_E_METHOD_DISABLED	0x80041056	An attempt was made to execute a method marked with [disabled].
wbemErrRefresherBusy	WBEM_E_REFRESHER_BUSY	0x80041057	The refresher is busy with another operation.
wbemErrUnparsableQuery	WBEM_E_UNPARSABLE_QUERY	0x80041058	The filtering query is syntactically invalid.
wbemErrNotEventClass	WBEM_E_NOT_EVENT_CLASS	0x80041059	The FROM clause of a filtering query refere class that is not an event class (not derived from __Event).
wbemErrMissingGroupWithin	WBEM_E_MISSING_GROUP_WITHIN	0x8004105a	A GROUP BY clause was used without the corresponding GROUP WITHIN clause.
wbemErrMissingAggregationList	WBEM_E_MISSING_AGGREGATION_LIST	0x8004105b	A GROUP BY clause was used. Aggregation on all properties is not supported.

Table 5.2 *The wbemErrorEnum Values (continued)*

wbemErrPropertyNotAnObject	WBEM_E_PROPERTY_NOT_AN_OBJECT	0x8004105c	Dot notation was used on a property that is not an embedded object.
wbemErrAggregatingByObject	WBEM_E_AGGREGATING_BY_OBJECT	0x8004105d	A GROUP BY clause references a property that is an embedded object without using dot notation.
wbemErrUninterpretableProviderQuery	WBEM_E_UNINTERPRETABLE_PROVIDER_QUERY	0x8004105f	An event provider registration query (__EventProviderRegistration) did not specify the classes for which events were provided.
wbemErrBackupRestoreWinmgtRunning	WBEM_E_BACKUP_RESTORE_WINMGMT_RUNNING	0x80041060	An request was made to back up or restore the repository while WinMgmt.exe was using it.
wbemErrQueueOverflow	WBEM_E_QUEUE_OVERFLOW	0x80041061	The asynchronous delivery queue overflowed due to the event consumer being too slow.
wbemErrPrivilegeNotHeld	WBEM_E_PRIVILEGE_NOT_HELD	0x80041062	The operation failed because the client did not have the necessary security privilege.
wbemErrInvalidOperator	WBEM_E_INVALID_OPERATOR	0x80041063	The operator is not valid for this property type.
wbemErrLocalCredentials	WBEM_E_LOCAL_CREDENTIALS	0x80041064	The user specified a username/password/authority on a local connection. The user must use a blank username/password and rely on default security.
wbemErrCannotBeAbstract	WBEM_E_CANNOT_BE_ABSTRACT	0x80041065	The class was made abstract when its superclass is not abstract.
wbemErrAmendedObject	WBEM_E_AMENDED_OBJECT	0x80041066	An amended object was PUT without the WBEM_FLAG_USE_AMENDED_QUALIFIERS flag being specified.
	WBEM_E_CLIENT_TOO_SLOW		The client was not retrieving objects quickly enough from an enumeration.
wbemErrRegistrationTooBroad	WBEMESS_E_REGISTRATION_TOO_BROAD	0x80042001	The provider registration overlaps with the system event domain.
wbemErrRegistrationTooPrecise	WBEMESS_E_REGISTRATION_TOO_PRECISE	0x80042002	A WITHIN clause was not used in this query.
wbemErrTimedout	WBEM_E_RETRY_LATER	0x80043001	Reserved for future use.
wbemErrResetToDefault	WBEM_E_RESOURCE_CONTENTION	0x80043002	Reserved for future use.

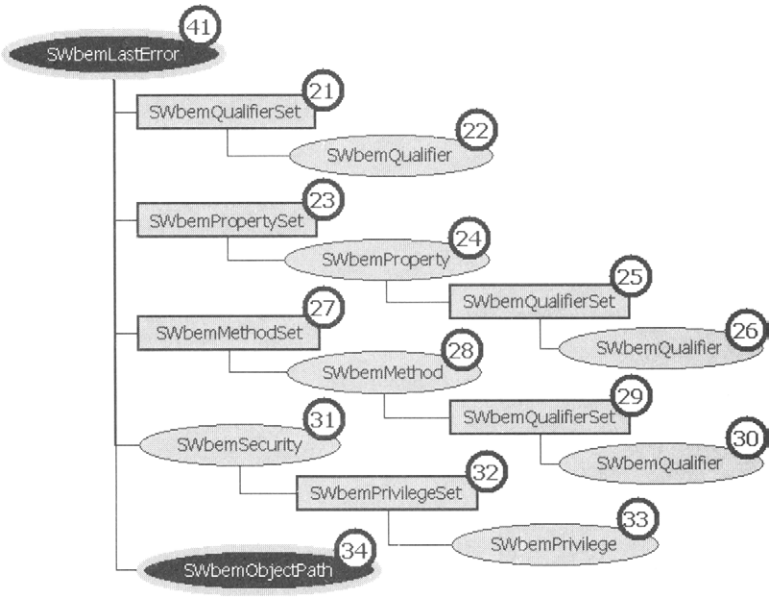


Figure 5.5
Using the WMI object model to examine a WMI last error object instance.

retrieved. If the method is executed asynchronously, the error will be retrieved from the **SWbemLastError** object available in the SINK_OnCompleted() sink routine. The **SWbemLastError** is an **SWbemObject** in its structure and contains information about the error. The **SWbemLastError** object available from Figure 5.5 is represented by the number 41 in the object model. You can compare it with the **SWbemObject** object available from Figure 4.4, represented by the number 20. As an **SWbemLastError** object has the same object model as an **SWbemObject**, all methods and properties exposed by this object are the same.

- **Errors from the execution of a method exposed by a CIM class definition.** In this case, the return code is specific to the CIM class definition method execution and may vary from one CIM class to another. For instance, Table 5.3 contains the error codes returned when executing a method of a *Win32_Service* class instance.

If you execute the **LoadCIMinXL.wsf** script developed in the previous chapter (Sample 4.32, "A Windows Script File self-documenting the CIM repository classes in an Excel sheet"), the description qualifier related to each CIM class method contains a description of the return codes.

We can start with an easy sample. If we reuse Sample 4.1 and retrieve an instance of a *Win32_Service*, we can include some error checking when cre-

Table 5.3 *The Win32_Service Return Codes*

Methods ReturnCode		
0x0	0	The request was accepted.
0x1	1	The request is not supported.
0x2	2	The user did not have the necessary access.
0x3	3	The service cannot be stopped because other services that are running are dependent on it.
0x4	4	The requested control code is not valid, or it is unacceptable to the service.
0x5	5	The requested control code cannot be sent to the service because the state of the service (Win32_BaseService State property) is equal to 0, 1, or 2.
0x6	6	The service has not been started.
0x7	7	The service did not respond to the start request in a timely fashion.
0x8	8	Unknown failure when starting the service.
0x9	9	The directory path to the service executable file was not found.
0xA	10	The service is already running.
0xB	11	The database to add a new service is locked.
0xC	12	A dependency for which this service relies on has been removed from the system.
0xD	13	The service failed to find the service needed from a dependent service.
0xE	14	The service has been disabled from the system.
0xF	15	The service does not have the correct authentication to run on the system.
0x10	16	This service is being removed from the system.
0x11	17	There is no execution thread for the service.
0x12	18	There are circular dependencies when starting the service.
0x13	19	There is a service running under the same name.
0x14	20	There are invalid characters in the name of the service.
0x15	21	Invalid parameters have been passed to the service.
0x16	22	The account which this service is to run under is either invalid or lacks the permissions to run the service.
0x17	23	The service exists in the database of services available from the system.
0x18	24	The service is currently paused in the system.

ating the instance. This logic is implemented in Sample 5.8 (lines 36 to 43). Notice the traditional VBScript error management at line 36. After the display of the VBScript run-time errors (line 37), the script creates an **SWbemLastError** object to retrieve more information about the WMI error (lines 38 to 41).

Sample 5.8 *Trapping error when creating a single instance*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    Const cComputerName = "LocalHost"
21:    Const cWMINameSpace = "root/cimv2"
22:    Const cWMIClass = "Win32_Service"
23:    Const cWMIInstance = "UNKNOWNSVC"
24:
25:    On Error Resume Next
..:
31:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
32:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
33:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
34:    Set objWMIInstance = objWMIServices.Get (cWMIClass & "='" & cWMIInstance & "'")
35:
36:    If Err.Number Then
37:       WScript.Echo "0x" & Hex(Err.Number) & " - " & Err.Description
38:       Set objWMILastError = CreateObject("wbemscripting.swbemlasterror")
39:       WScript.Echo "Operation: " & objWMILastError.Operation
40:       WScript.Echo "ParameterInfo: " & objWMILastError.ParameterInfo
41:       WScript.Echo "ProviderName: " & objWMILastError.ProviderName
42:       WScript.Quit (1)
43:    End If
44:
45:    WScript.Echo objWMIInstance.Name & " (" & objWMIInstance.Description & ")"
46:    WScript.Echo "   AcceptPause=" & objWMIInstance.AcceptPause
47:    WScript.Echo "   AcceptStop=" & objWMIInstance.AcceptStop
..:
69:    WScript.Echo "   TagId=" & objWMIInstance.TagId
70:    WScript.Echo "   WaitHint=" & objWMIInstance.WaitHint
..:
75:    ]]>
76:    </script>
77:  </job>
78:</package>
```

Sample 5.8 references an invalid *Win32_Service* instance (line 23); as a result, the script fails during the instance retrieval (line 34). The obtained output is as follows:

```
C:\>"GetSingleInstanceWithAPI (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

0x80041002 - Not found
Operation: GetObject
ParameterInfo: Win32_Service.Name="UNKNOWNSVC"
ProviderName: CIMWin32
```

Let's look at a more complex script sample that we are familiar with: Sample 5.7, which asynchronously changes the startup mode of a *Win32_Service* instance to Automatic if it was set on Manual and vice versa. This script is a nice application for an error management case because it uses many aspects of WMI, such as the following:

- The WMI connection operation
- The asynchronous *Win32_Service* instance retrieval
- The WMI method invocation to execute asynchronously the CIM class method *ChangeStartMode*
- The assignment of a parameter for the asynchronous CIM class method execution
- Some sink routine contexts

Let's see what the script looks like when we include some error management code. The modifications made to Sample 5.7 are in boldface and shown in Sample 5.9.

Sample 5.9 *Using one method with parameters of a Win32_Service instance asynchronously with context and error management from VBScript*

```
1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <script language="VBScript" src="..\Functions\TinyErrorHandler.vbs" />
14:
15:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
16:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMIMethodSinkContext" reference="true"/>
17:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMIInstanceSinkContext" reference="true"/>
18:
19:    <script language="VBscript">
20:    <![CDATA[
..:
28:    ' ---------------------------------------------------------------------------
29:    Const cComputerName = "LocalHost"
30:    Const cWMINameSpace = "root/cimv2"
31:    Const cWMIClass = "Win32_Service"
32:    Const cWMIInstance = "SNMP"
..:
37:    On Error Resume Next
38:
39:    Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
40:
41:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
42:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
43:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
44:    ' In case of WMI Connection problem
45:    If Err.Number Then ErrorHandler (Err)
46:
```

```
47:    objWMIInstanceSinkContext.Add "WMIMethod", "GetAsync"
48:    objWMIServices.GetAsync objWMISink, _
49:                            cWMIClass & "='" & cWMIInstance & "'", _
50:                            , _
51:                            , _
52:                            objWMIInstanceSinkContext
53:
54:    WScript.Echo "Continuing script execution ..."
55:    PauseScript "Click on 'Ok' to terminate the script ..."
56:
57:    objWMISink.Cancel
..:
63:
64:    ' ------------------------------------------------------------------------------
65:    Sub SINK_OnCompleted (intHResult, objWMILastError, objWMIAsyncContext)
..:
70:        On Error Resume Next
71:
72:        Wscript.Echo
73:        Wscript.Echo "BEGIN - OnCompleted."
74:
75:        Set objContextItem = objWMIAsyncContext.Item ("WMIMethod")
76:        ' In case of wrong context reference
77:        If Err.Number Then ErrorHandler (Err)
78:
79:        If intHResult = 0 Then
80:            WScript.Echo "'" & objContextItem.Value & _
81:                         "' WMI Scripting API Asynchronous call successful " & _
82:                         "(0x" & Hex(intHResult) & ")."
83:        Else
84:            WScript.Echo "-------------------------------------------------------------"
85:
86:            WScript.Echo "'" & objContextItem.Value & _
87:                         "' WMI Scripting API Asynchronous call failed " & _
88:                         "(0x" & Hex(intHResult) & ")."
89:
90:            WScript.Echo vbCRLF & "SWbemLastError content:"
91:            Set objWMIPropertySet = objWMILastError.Properties_
92:            For Each objWMIProperty In objWMIPropertySet
93:                WScript.Echo "  " & objWMIProperty.Name & "=" & objWMIProperty.Value
94:            Next
95:            Set objWMIPropertySet = Nothing
96:            WScript.Echo "-------------------------------------------------------------"
97:        End If
98:
99:        Set objContextItem = Nothing
100:
101:       Wscript.Echo "END   - OnCompleted."
102:
103:   End Sub
...:
116:   ' ------------------------------------------------------------------------------
117:   Sub SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
...:
123:       On Error Resume Next
124:
125:       Wscript.Echo
126:       Wscript.Echo "BEGIN - OnObjectReady."
127:
128:       Set objContextItem = objWMIAsyncContext.Item ("WMIMethod")
```

```
129:        ' In case of wrong context reference
130:        If Err.Number Then ErrorHandler (Err)
131:
132:        Select Case objContextItem.Value
133:               Case "GetAsync"
134:                    WScript.Echo "'" & objWMIInstance.DisplayName & _
135:                                 "' startup is currently '" & _
136:                                 objWMIInstance.StartMode & "'."
137:
138:                    Set objWMIMethod = objWMIInstance.Methods_("ChangeStartMode")
139:                    ' In case of wrong method reference
140:                    If Err.Number Then ErrorHandler (Err)
141:
142:                    Set objWMIInParameters = objWMIMethod.InParameters
143:
144:                    If objWMIInstance.StartMode = "Manual" Then
145:                       objWMIMethodSinkContext.Add "WMIMethod", "ExecMethodAsync"
146:                       objWMIMethodSinkContext.Add "CIMMethod", "ChangeStartMode (Automatic)"
147:                       objWMIInParameters.Properties_.Item("StartMode") = "Automatic"
148:                       ' In case of wrong parameter reference
149:                       If Err.Number Then ErrorHandler (Err)
150:
151:                       objWMIInstance.ExecMethodAsync_ objWMISink, _
152:                                                       "ChangeStartMode", _
153:                                                       objWMIInParameters, _
154:                                                       , _
155:                                                       , _
156:                                                       objWMIMethodSinkContext
157:                    End If
158:                    If objWMIInstance.StartMode = "Auto" Then
159:                       objWMIMethodSinkContext.Add "WMIMethod", "ExecMethodAsync"
160:                       objWMIMethodSinkContext.Add "CIMMethod", "ChangeStartMode (Manual)"
161:                       objWMIInParameters.Properties_.Item("StartMode") = "Manual"
162:                       ' In case of wrong parameter reference
163:                       If Err.Number Then ErrorHandler (Err)
164:
165:                       objWMIInstance.ExecMethodAsync_ objWMISink, _
166:                                                       "ChangeStartMode", _
167:                                                       objWMIInParameters, _
168:                                                       , _
169:                                                       , _
170:                                                       objWMIMethodSinkContext
171:                    End If
...:
176:               Case "ExecMethodAsync"
177:                    Set objContextItem = objWMIAsyncContext.Item ("CIMMethod")
178:                    ' In case of wrong context reference
179:                    If Err.Number Then ErrorHandler (Err)
180:
181:                    If objWMIInstance.ReturnValue = 0 Then
182:                       Wscript.Echo "'" & objContextItem.Value & _
183:                                    "' method execution successful " & _
184:                                    "(0x" & Hex(objWMIInstance.ReturnValue) & ")."
185:                    Else
186:                       Wscript.Echo "'" & objContextItem.Value & _
187:                                    "' method execution failed " & _
188:                                    "(0x" & Hex(objWMIInstance.ReturnValue) & ")."
189:                    End If
190:        End Select
...
```

```
194:        Wscript.Echo "END    - OnObjectReady."
195:
196:    End Sub
...:
223:    ]]>
224:    </script>
225:  </job>
226:</package>
```

At line 13, the script starts by including a small function used as an error handler. This function is very basic, because its only purpose is to display the *Err.number* and the *Err.Description* before stopping the script execution. The function's code is as follows:

```
 1:' VB Script to handle errors occuring during a script execution.                '
 2:' This is a very simple error handling. No error recovery is made.               '
 3:' Only the error code number and the associated message is displayed.           '
..:
10:Option Explicit
11:
12:' ---------------------------------------------------------------------
13:Function ErrorHandler (Err)
14:
15:        WScript.Echo "-------------------------------------------------"
16:        Wscript.Echo "Error: 0x" & Hex(Err.Number) & vbCRLF
17:        Wscript.Echo Err.Description
18:        WScript.Echo "-------------------------------------------------"
19:
20:        Wscript.Quit (1)
21:
22:End Function
```

When Sample 5.9 executes, several errors may occur. We may have some of the following cases:

- **A WMI connection error may occur when using the SWbemLocator object to create the SWbemServices object (line 43).** For instance, an error may occur because the system given in the constant at line 29 is not available. Another case could be a set of invalid credentials passed to the **SWbemLocator** *ConnectServer* method. In this case, line 45 of Sample 5.9 traps the error and calls the error handler function. This is typically a run-time error. For an unavailable server, the script output is

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

-------------------------------------------------------------
Error: 0x800706BA

The RPC server is unavailable.
-------------------------------------------------------------
```

For a set of invalid credentials, the script output will be:

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

-----------------------------------------------------------
Error: 0x80070005

Access is denied.
-----------------------------------------------------------
```

- **When creating the SWbemServices object, the referenced CIM class does not exist (line 31).** In this case, the asynchronous method *GetAsync* is executed (line 48), but because the class doesn't exist, no instance is retrieved. Because the WMI **SWbemServices** method uses an asynchronous method, the error is returned in the SINK_OnCompleted() sink routine in the context of the *GetAsync* method execution (line 75). In this case, *intHresult* is not equal to zero (line 79), which means that the **SWbemLastError** object is initialized from the sink routine parameters. Note that the *intHresult* value is displayed with the message (lines 86 to 88). In this case, the script shows the content of the **SWbemLastError** object. Because the **SWbemLastError** object is an **SWbemObject** object, the script enumerates the list of the properties available with their respective values (lines 90 to 95). This type of error is a WMI operational error. In this case, the script output is

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Continuing script execution ...

BEGIN - OnCompleted.
-----------------------------------------------------------
'GetAsync' WMI Scripting API Asynchronous call failed (0x80041010).

SWbemLastError content:
  Description=
  Operation=GetObject
  ParameterInfo=Win32_RequestedClassName='SNMP'
  ProviderName=WinMgmt
  StatusCode=
-----------------------------------------------------------
END   - OnCompleted.
```

- **When creating the SWbemServices object, the requested instance does not exist (line 32).** In this case, the asynchronous method *GetAsync* is executed, but because the referenced instance does not exist,

no instance is retrieved. Since the problem occurs during the *GetAsync* asynchronous method execution, the logic is the same as in the previous case. The error is retrieved during the execution of the SINK_OnCompleted() sink routine. This is a WMI operational error. In this case, the script output is

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Continuing script execution ...

BEGIN - OnCompleted.
------------------------------------------------------------
'GetAsync' WMI Scripting API Asynchronous call failed (0x80041002).

SWbemLastError content:
  Description=
  Operation=GetObject
  ParameterInfo=Win32_Service.Name="RequestedInstanceName"
  ProviderName=CIMWin32
  StatusCode=
------------------------------------------------------------
END   - OnCompleted.
```

- **When retrieving a sink routine context, the expected context is not found.** This can typically be the case if a sink routine looks for a context, but the **SWbemNamedValueSet** collection does not contain the expected context. This error may occur on lines 75, 128, and 177 of Sample 5.9. The code looks for a WMIContext or a CIMContext **SWbemNamedValue** (lines 75, 128, or 177). If another **SWbemNamedValue** is requested (e.g., due to a mistyping of the **SWbemNamedValue** name), this produces the following run-time error output and stops the script execution:

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Continuing script execution ...

BEGIN - OnObjectReady.
------------------------------------------------------------
Error: 0x80041002

Not found
------------------------------------------------------------
```

- **When referencing a CIM method to get the list of its parameters in an SWbemObject object, the expected CIM method is not found.**

This error may occur at line 138 of Sample 5.9 and is probably due to a mistyping of the method name. Basically, even if the error context is a bit different (it is related to a method name reference and not a WMI context name), the run-time error produced is the same as the previous one. In this case the output will be as follows:

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Continuing script execution ...

BEGIN - OnObjectReady.
'SNMP Service' startup is currently 'Auto'.
-----------------------------------------------------------
Error: 0x80041002

Not found
-----------------------------------------------------------
```

- **When referencing a CIM method parameter, the referenced parameter is not found.** Again, this is the same type of run-time error as in the previous case, but it is related to a method parameter name and not to the method name itself. This error may occur at lines 147 and 161 of Sample 5.9.

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Continuing script execution ...

BEGIN - OnObjectReady.
'SNMP Service' startup is currently 'Manual'.
-----------------------------------------------------------
Error: 0x80041002

Not found
-----------------------------------------------------------
```

- **The assigned CIM method parameter is invalid.** As the CIM method is executed asynchronously with the *ExecMethodAsync* method of the **SWbemObject** representing the SNMP *Win32_Service* instance, the error is retrieved in the SINK_OnObjectReady() sink routine, which is called when the *ExecMethodAsync* is executed. As shown previously, when examining the asynchronous event contexts, the *ExecMethodAsync* is invoked at lines 151 and 165, and, with the help of the context, the returned result of this method invocation is verified at line 181. Depending on the execution result, an appropriate mes-

sage is displayed with the result value (lines 182 to 184, 186, and 188). This fault does not cause a run-time error; it is a WMI operational error because the CIM method parameter is wrong. The returned code is related to the CIM class method invoked. In this case, the result is one of the values presented in Table 5.3 for the *Win32_Service*. The error occurs during the SINK_OnCompleted() sink routine execution, but in the context of the *ExecMethodAsync* method invocation. The display output is as follows:

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Continuing script execution ...

BEGIN - OnObjectReady.
'SNMP Service' startup is currently 'Manual'.
END   - OnObjectReady.

BEGIN - OnCompleted.
'GetAsync' WMI Scripting API Asynchronous call successful (0x0).
END   - OnCompleted.

BEGIN - OnObjectReady.
'ChangeStartMode (Automatic)' method execution failed (0x15).
END   - OnObjectReady.

BEGIN - OnCompleted.
'ExecMethodAsync' WMI Scripting API Asynchronous call successful
(0x0).
END   - OnCompleted.
```

- **The executed CIM method name is invalid.** This error may occur at lines 152 and 166. Here the *ExecMethodAsync* invokes a CIM class method that does not exist for the instantiated CIM class. This results in a failure in the execution of the *ExecMethodAsync* method of the **SWbemObject** object. It is important to differentiate this error from the previous one. In the previous case, the *ExecMethodAsync* method invocation is successful, but the execution of the *ChangeStartMode* method fails due to an invalid parameter. In this case, the *ExecMethodAsync* method can't execute the CIM class method because it is not a valid method name. Both are WMI operational errors, but one is at the level of the WMI scripting API object method (this case), and the other is at the level of CIM class method execution (previous case). The display output is as follows:

```
C:\>"SwapServiceStartupAndExecuteMethodAsyncWithContext (Error).wsf"
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Continuing script execution ...

BEGIN - OnObjectReady.
'SNMP Service' startup is currently 'Manual'.
END   - OnObjectReady.

BEGIN - OnCompleted.
'GetAsync' WMI Scripting API Asynchronous call successful (0x0).
END   - OnCompleted.

BEGIN - OnCompleted.
-------------------------------------------------------------
'ExecMethodAsync' WMI Scripting API Asynchronous call failed
(0x80041055).

SWbemLastError content:
  Description=
  Operation=ExecMethod
  ParameterInfo=Win32_Service.Name="SNMP"
  ProviderName=WinMgmt
  StatusCode=
-------------------------------------------------------------
END   - OnCompleted.
```

- **The CIM method execution failed.** Even if all parameters required to execute the *ChangeStartMode* method are correct, it is possible that the CIM class method execution could fail because of external reasons. In this case, the script falls into the same scenario as an invalid method parameter input. So, the WMI scripting API object method *ExecAsyncMethod* works, but the CIM class method execution fails. The script receives a return code of the CIM class method execution during the SINK_OnCompleted() sink routine execution, in the context of the *ExecMethodAsync* method invocation.

Sample 5.9, shows how to handle errors from VBScript. Although the languages are different, the logic to manage the errors from JScript is the same. Only the statements are different. Sample 5.10 shows the same script as Sample 5.9 but written in JScript. Notice the inclusion of the VBScript error handler function previously used (line 13). Sample 5.10 takes advantage of the mixed language possibilities offered by WSH from a Windows Script File. The script is provided for reference. All comments made for the VBScript version remain valid for the JScript version.

Sample 5.10 *Using one method with parameters of a Win32_Service instance asynchronously with context and error management from JScript*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <script language="VBScript" src="..\Functions\TinyErrorHandler.vbs" />
14:
15:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
16:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMIMethodSinkContext" reference="true"/>
17:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMIInstanceSinkContext" reference="true"/>
18:
19:    <script language="Jscript">
20:    <![CDATA[
21:
22:    // ------------------------------------------------------------------------------------
23:    var cComputerName = "W2K-DPEN6400"
24:    var cWMINameSpace = "root/cimv2"
25:    var cWMIClass = "Win32_Service"
26:    var cWMIInstance = "SNMP"
..:
31:    objWMISink = Wscript.CreateObject ("WbemScripting.SWbemSink", "SINK_");
32:
33:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault;
34:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate;
35:    try
36:      {
37:      objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "");
38:      }
39:    // In case of WMI Connection problem
40:    catch (Err)
41:      {
42:      ErrorHandler (Err)
43:      }
44:
45:    objWMIInstanceSinkContext.Add ("WMIMethod", "GetAsync");
46:    objWMIServices.GetAsync (objWMISink, cWMIClass + "='" + cWMIInstance + "'",
47:                             null,
48:                             null,
49:                             objWMIInstanceSinkContext);
50:
51:    Wscript.Echo ("Continuing script execution ...");
52:    PauseScript ("Click on 'Ok' to terminate the script ...");
53:
54:    objWMISink.Cancel;
55:
56:    // ------------------------------------------------------------------------------------
57:    function SINK_OnCompleted (intHResult, objWMILastError, objWMIAsyncContext)
58:        {
..:
64:        Wscript.Echo ();
65:        Wscript.Echo ("BEGIN - OnCompleted.");
66:
67:        try
68:          {
69:          objContextItem = objWMIAsyncContext.Item ("WMIMethod");
```

```
 70:          }
 71:        // In case of wrong context reference
 72:        catch (Err)
 73:          {
 74:          ErrorHandler (Err)
 75:          }
 76:
 77:        if (intHResult == 0)
 78:           {
 79:           Wscript.Echo ("'" + objContextItem.Value +
 80:                         "' WMI Scripting API Asynchronous call successful " +
 81:                         "(" + intHResult + ").");
 82:           }
 83:        else
 84:           {
 85:           Wscript.Echo ("-------------------------------------------------------------");
 86:
 87:           Wscript.Echo ("'" + objContextItem.Value +
 88:                         "' WMI Scripting API Asynchronous call failed " +
 89:                         "(" + intHResult + ").");
 90:
 91:           Wscript.Echo ();
 92:           Wscript.Echo ("SWbemLastError content:");
 93:           objWMIPropertySet = objWMILastError.Properties_;
 94:
 95:           enumWMIPropertySet = new Enumerator (objWMIPropertySet);
 96:           for (;! enumWMIPropertySet.atEnd(); enumWMIPropertySet.moveNext())
 97:               {
 98:               varItem = enumWMIPropertySet.item()
 99:               Wscript.Echo ("  " + varItem.name + "=" + enumWMIPropertySet.item());
100:               }
101:
102:          Wscript.Echo ("-------------------------------------------------------------");
103:          }
104:
105:       Wscript.Echo ("END   - OnCompleted.");
106:
107:       }
108:
109:   // ----------------------------------------------------------------------------------
110:   function SINK_OnObjectPut (objWMIPath, objWMIAsyncContext)
...:
119:   // ----------------------------------------------------------------------------------
120:   function SINK_OnObjectReady (objWMIInstance, objWMIAsyncContext)
121:       {
...:
127:       Wscript.Echo ();
128:       Wscript.Echo ("BEGIN - OnObjectReady.");
129:
130:       try
131:         {
132:         objContextItem = objWMIAsyncContext.Item ("WMIMethod")
133:         }
134:       // In case of wrong context reference
135:       catch (Err)
136:         {
137:         ErrorHandler (Err)
138:         }
139:
140:       switch (objContextItem.Value)
```

```
141:              {
142:              case "GetAsync":
143:                   Wscript.Echo ("'" + objWMIInstance.DisplayName +
144:                                 "' startup is currently '" +
145:                                 objWMIInstance.StartMode + "'.");
146:
147:                   try
148:                     {
149:                     objWMIMethod = objWMIInstance.Methods_("ChangeStartMode");
150:                     }
151:                   // In case of wrong method reference
152:                   catch (Err)
153:                     {
154:                     ErrorHandler (Err)
155:                     }
156:
157:                   objWMIInParameters = objWMIMethod.InParameters;
158:
159:                   if (objWMIInstance.StartMode == "Manual")
160:                      {
161:                      objWMIMethodSinkContext.Add ("WMIMethod", "ExecMethodAsync");
162:                      objWMIMethodSinkContext.Add ("CIMMethod", "ChangeStartMode Automatic)");
163:
164:                      try
165:                        {
166:                        objWMIInParameters.Properties_.Item("StartMode") = "Automatic";
167:                        }
168:                      // In case of wrong parameter reference
169:                      catch (Err)
170:                        {
171:                        ErrorHandler (Err)
172:                        }
173:
174:                      objWMIInstance.ExecMethodAsync_ (objWMISink,
175:                                                       "ChangeStartMode",
176:                                                       objWMIInParameters,
177:                                                       null,
178:                                                       null,
179:                                                       objWMIMethodSinkContext);
180:                      }
181:                   if (objWMIInstance.StartMode == "Auto")
182:                      {
183:                      objWMIMethodSinkContext.Add ("WMIMethod", "ExecMethodAsync");
184:                      objWMIMethodSinkContext.Add ("CIMMethod", "ChangeStartMode (Manual)");
185:
186:                      try
187:                        {
188:                        objWMIInParameters.Properties_.Item("StartMode") = "Manual";
189:                        }
190:                      // In case of wrong parameter reference
191:                      catch (Err)
192:                        {
193:                        ErrorHandler (Err)
194:                        }
195:
196:                      objWMIInstance.ExecMethodAsync_ (objWMISink,
197:                                                       "ChangeStartMode",
198:                                                       objWMIInParameters,
199:                                                       null,
200:                                                       null,
```

```
                                                        objWMIMethodSinkContext);
                    }
                break;

            case "ExecMethodAsync":
                try
                  {
                  objContextItem = objWMIAsyncContext.Item ("CIMMethod");
                  }
                // In case of wrong context reference
                catch (Err)
                  {
                  ErrorHandler (Err)
                  }

                if (objWMIInstance.ReturnValue == 0)
                   {
                   Wscript.Echo ("'" + objContextItem.Value +
                                 "' method execution successful " +
                                 "(" + objWMIInstance.ReturnValue + ").");
                   }
                else
                   {
                   Wscript.Echo ("'" + objContextItem.Value +
                                 "' method execution failed " +
                                 "(" + objWMIInstance.ReturnValue + ").");
                   }
                break;
            }

        Wscript.Echo ("END   - OnObjectReady.");

        }
    // ---------------------------------------------------------------------------
    function SINK_OnProgress (intUpperBound, intCurrent, strMessage, objWMIAsyncContext)
...:
    // ---------------------------------------------------------------------------
    function PauseScript (strMessage)
...:
    ]]>
    </script>
  </job>
</package>
```

By examining the possible error types of Sample 5.9, we clearly see that we have the following:

- Traditional run-time errors that can be addressed with the **On Error Resume Next** statement combined with the *Err.Number* statement from VBScript or with the **try ... catch ... finally** statement from JScript.
- WMI scripting API errors that produce an **SWbemLastError** object available in the SINK_OnCompleted() sink routine if the call is asynchronous. Otherwise, these errors are produced as run-time errors.

- CIM class method execution errors that return specific return codes related to the CIM class method invoked (i.e., *Win32_Service* in Table 5.3).

We see that error handling makes reading the script code more difficult. Although it is a mandatory coding for a production script, these statements are not required during the learning phase. For this reason, subsequent scripts presented in the book, where possible, do not contain error management statements. But keep in mind that as a script developer, you must add the required tests in your production code to make sure that the written logic is reliable and behaves as expected.

5.4 WMI DateTime Helper

5.4.1 Under Windows XP and Windows.NET Server

WMI DateTime Helper is an object present in the WMI object model and added under Windows XP and Windows.NET Server to simplify date/time conversions. For instance, in the previous chapter, Table 4.16 contains the properties of the *Win32_Registry* class. In the property list, we have the *InstallDate* property. The *InstallDate* property contains a date and a time represented as follows:

```
yyyymmddHHMMSS.mmmmmmsUUU
```

This format is explained in Table 5.4.

This format is used in all date and time representations made in the CIM repository. Although this date and time representation is a simple string that can be parsed with the standard VBscript and JScript run-time

Table 5.4 *The DMTF Data and Time Format*

yyyy	Four-digit year (0000 through 9999). Your implementation can restrict the supported range. For example, an implementation can support only the years 1980 through 2099.
mm	Two-digit month (01 through 12).
dd	Two-digit day of the month (01 through 31). This value must be appropriate for the month. For example, February 31 is invalid. However, your implementation does not have to check for valid data.
HH	Two-digit hour of the day using the 24-hour clock (00 through 23).
MM	Two-digit minute in the hour (00 through 59).
SS	Two-digit number of seconds in the minute (00 through 59).

Table 5.4 *The DMTF Data and Time Format (continued)*

mmmmmm	Six-digit number of microseconds in the second (000000 through 999999). You implementation does not have to support evaluation using this field. However, this field must always be present to preserve the fixed-length nature of the string.
s	Plus sign (+) or minus sign (–) to indicate a positive or negative offset from Universal Time Coordinates (UTC).
UUU	Three-digit offset indicating the number of minutes that the originating time zone deviates from UTC. For WMI, it is encouraged, but not required, to convert times to GMT with a UTC offset of zero.

library functions, implementing a WMI object to abstract the string representation facilitates its manipulation and conversion to other formats. The WMI object abstracting the DateTime representation is called the **SWBemDateTime** object. Note that this object is available only under Windows XP and Windows.NET Server. Table 5.5 lists its properties and methods.

Table 5.5 *The SWBemDateTime Object*

Properties	Year	Year component Must be in range 0000-9999
	YearSpecified	Whether Year is significant
	Month	Month component Must be in range 01-12
	MonthSpecified	Whether Month is significant
	Day	Day component Must be in range 01-31 if IsInterval is FALSE, and 0-99999999 otherwise
	DaySpecified	Whether Day is significant
	Hours	Hour component Must be in range 0-23
	HoursSpecified	Whether Hour is significant
	Minutes	Minutes Component Must be in range 0-59
	MinutesSpecified	Whether Minute is significant
	Seconds	Seconds component Must be in range 0-59
	SecondsSpecified	Whether Seconds is significant

Table 5.5 *The SWBemDateTime Object (continued)*

Properties (cont'd.)	Microseconds	Microseconds component Must be in range 0 to 999999
	MicrosecondsSpecified	Whether Microseconds is significant
	UTC	UTC offset Signed number in the range -720,720
	UTCSpecified	Whether UTC is significant
	IsInterval	If TRUE, value represents an interval value. Otherwise it represents a Datetime value.
	Value	The raw DMTF-format string value representation of the date time.
Methods	SetVarDate()	The variant date value to be used to set this object. If TRUE the supplied vDate is interpreted as a local time and is converted internally to the correct UTC format. Otherwise vDate is converted directly into a UTC value with offset 0.
	GetVarDate()	If TRUE then the return value expresses a local time (for the client). Otherwise the return value is to be regarded as a UTC time.
	SetFileTime()	The FILETIME value used to set this object. If TRUE the supplied strFileTime is interpreted as a local time and is converted internally to the correct UTC format. Otherwise strFileTIme is converted directly into a UTC value with offset 0.
	GetFileTime()	If TRUE then the return value expresses a local time (for the client). Otherwise the return value is to be regarded as a UTC time.

To illustrate the use of the **SWBemDateTime** object, Sample 5.11 shows how the DMTF DateTime format of the *Win32_Registry* instance can be converted to other formats.

Sample 5.11 *Converting a DateTime DMTF format to scripting run-time supported formats*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:    <object progid="WbemScripting.SWbemDateTime" id="objWMIDateTime" />
15:
16:    <script language="VBscript">
17:    <![CDATA[
..:
21:    Const cComputerName = "LocalHost"
22:    Const cWMINameSpace = "root/cimv2"
23:    Const cWMIClass = "Win32_Registry"
24:    Const cWMIInstance = "Microsoft Windows Whistler Server|C:\WINNT|\Device\Harddisk0\Partition1"
```

```
..:
30:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
31:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
32:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
33:    Set objWMIInstance = objWMIServices.Get (cWMIClass & "='" & cWMIInstance & "'")
34:
35:    WScript.Echo objWMIInstance.Name & "=" & objWMIInstance.Name
36:    WScript.Echo "   Caption=" & objWMIInstance.Caption
37:    WScript.Echo "   CurrentSize=" & objWMIInstance.CurrentSize
38:    WScript.Echo "   Caption=" & objWMIInstance.Caption
39:    WScript.Echo "   Description=" & objWMIInstance.Description
40:
41:    objWMIDateTime.Value = objWMIInstance.InstallDate
42:    WScript.Echo "   InstallDate (Value)=" & objWMIDateTime.Value
43:    WScript.Echo "   InstallDate (varDate UTC Time)=" & objWMIDateTime.GetVarDate (False)
44:    WScript.Echo "   InstallDate (varDate Local Time)=" & objWMIDateTime.GetVarDate (True)
45:    WScript.Echo "   InstallDate (FileTime UTC Time)=" & objWMIDateTime.GetFileTime (False)
46:    WScript.Echo "   InstallDate (FileTime Local Time)=" & objWMIDateTime.GetFileTime (True)
47:    WScript.Echo "   InstallDate (Year)=" & objWMIDateTime.Year
48:    WScript.Echo "   InstallDate (Month)=" & objWMIDateTime.Month
49:    WScript.Echo "   InstallDate (Day)=" & objWMIDateTime.Day
50:    WScript.Echo "   InstallDate (Hours)=" & objWMIDateTime.Hours
51:    WScript.Echo "   InstallDate (Minutes)=" & objWMIDateTime.Minutes
52:    WScript.Echo "   InstallDate (Seconds)=" & objWMIDateTime.Seconds
53:    WScript.Echo "   InstallDate (MicroSeconds)=" & objWMIDateTime.MicroSeconds
54:    WScript.Echo "   InstallDate (UTC)=" & objWMIDateTime.UTC
55:    WScript.Echo "   InstallDate (IsInterval)=" & objWMIDateTime.IsInterval
56:
57:    WScript.Echo "   MaximumSize=" & objWMIInstance.MaximumSize
58:    WScript.Echo "   ProposedSize=" & objWMIInstance.ProposedSize
59:    WScript.Echo "   Status=" & objWMIInstance.Status
..:
64:    ]]>
65:    </script>
66:  </job>
67:</package>
```

The script instantiates the **SWBemDateTime** object at line 14. Next, it retrieves the instance of the *Win32_Registry* class (lines 21 to 33) and lists the associated properties. The particularity resides at line 41, where the *InstallDate* property is assigned to the **SWBemDateTime** object *Value* property. Once loaded in the **SWBemDateTime** object, the script displays all the properties available from this object. The result is a display of the DMTF DateTime in various forms shown as follows:

```
C:\>ShowWin32_RegistryDateTime.wsf
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Microsoft Windows Whistler Server|C:\WINNT|\Device\Harddisk0\Partition1
   Caption=Registry
   CurrentSize=2
   Caption=Registry
   Description=Registry
   InstallDate (Value)=20010503021847.000000+120
```

```
InstallDate (varDate UTC Time)=03-05-2001 02:18:47
InstallDate (varDate Local Time)=03-05-2001 04:18:47
InstallDate (FileTime UTC Time)=126333299270000000
InstallDate (FileTime Local Time)=126333371270000000
InstallDate (Year)=2001
InstallDate (Month)=5
InstallDate (Day)=3
InstallDate (Hours)=2
InstallDate (Minutes)=18
InstallDate (Seconds)=47
InstallDate (MicroSeconds)=0
InstallDate (UTC)=120
InstallDate (IsInterval)=False
MaximumSize=51
ProposedSize=51
Status=OK
```

Of course, the **SWBemDateTime** object can be used with any properties containing a DMTF DateTime format. It is not dedicated to the *Win32_Registry* class. Because it allows the DMTF date/time to be represented in a more readable way, it was already used in the previous chapter at Sample 4.30 ("A generic routine to display the SWbemPropertySet object") to display the **SWbemPropertySet** object content. In the next chapter, we use this object again when we work with Absolute Timer events which are events scheduled to occur at a precise date and time.

5.4.2 What about Windows 2000 and before?

If your system runs under Windows 2000 or any previous platforms, the **SWBemDateTime** object is not available from the WMI COM object model. As most scripts included in this book are written for Windows XP or Windows.NET Server and make use of the **SWBemDateTime** object, this represents a compatibility problem. It is possible to work around this limitation without impacting the existing scripts with the use of the Windows Script Components (WSC) discussed in Chapter 1. As a reminder, a WSC is nothing but a COM implementation where the COM logic is coded in a script. If we write a WSC that emulates the behavior of the **SWBemDateTime** object, it will be possible to run any scripts making use of this object under Windows 2000 by simply modifying one single line. For instance, with Sample 5.11, we can replace line 14

```
14:    <object progid="WbemScripting.SWbemDateTime" id="objWMIDateTime" />
```

with:

```
14:    <object progid="SWbemDateTime.WSC" id="objWMIDateTime" />
```

If the **SWwbemDateTime.wsc** exposes the same properties and methods as the native **SWBemDateTime** object, this will guarantee the same level of method and properties invocations. However, consider the **SWBemDateTime.wsc** example given as a framework as it implements only some of the functionalities offered by the native **SWBemDateTime** object. Currently, This WSC component supports all **SWBemDateTime** object features with the following restrictions:

- It cannot set and get a *FileTime*.
- The *SetVarDate* method always interprets the given time as a local time.
- There is no interval management capability.
- There is no support for wildcards in the DMTF Date/Time format.

Despite these restrictions, the **SWBemDateTime.wsc** offers a level of functionality sufficient to run on Windows 2000 the scripts using the **SWBemDateTime** object of Windows XP or Windows.NET Server. The **SWBemDateTime.wsc** code is available in Sample 5.12.

Sample 5.12 *The SWbemDateTime Windows Script Component*

```
 1:<?xml version="1.0"?>
 .:
 8:<component>
 9:
10:  <registration
11:          description="SWbemDateTime"
12:          progid="SWbemDateTime.WSC"
13:          version="1.00"
14:          classid="{46818027-a0d8-4729-9750-74640dd14dc8}"
15:  >
16:  </registration>
17:
18:  <!--
..:
48:  <public>
49:    <method name="SetVarDate" internalname="SetVarDate">
50:      <parameter name="strVarDate" />
51:    </method>
52:
53:    <method name="GetVarDate" internalname="GetVarDate">
54:      <parameter name="boolIsLocal" />
55:    </method>
..:
65:    <property name="Day" internalname="strDay">
66:      <get/>
67:      <put/>
68:    </property>
69:
```

```
 70:    <property name="DaySpecified" internalname="boolDaySpecified">
 71:      <get/>
 72:    </property>
 73:
 74:    <property name="Month" internalname="strMonth">
 75:      <get/>
 76:      <put/>
 77:    </property>
 78:
 79:    <property name="MonthSpecified" internalname="boolMonthSpecified">
 80:      <get/>
 81:    </property>
 82:
 83:    <property name="Year" internalname="strYear">
 84:      <get/>
 85:      <put/>
 86:    </property>
...:
146:  </public>
147:
148:  <script language="VBScript">
149:  <![CDATA[
...:
194:  ' ---------------------------------------------------------------------------------------
195:  Function SetVarDate(strVarDate)
...:
200:          If Len (strVarDate) Then
201:             strDay = Right ("00" & Day (strVarDate),
202:             strMonth = Right ("00" & Month (strVarDate), 2)
203:             strYear = Right ("0000" & Year (strVarDate), 4)
204:             strHours = Right ("00" & Hour (strVarDate), 2)
205:             strMinutes = Right ("00" & Minute (strVarDate), 2)
206:             strSeconds = Right ("00" & Second (strVarDate), 2)
207:             strMicroseconds = "000000"
208:
209:             Set objWMIServices = GetObject ("Winmgmts:Root\CIMv2")
210:             Set objWMIInstances = objWMIServices.InstancesOf ("Win32_ComputerSystem")
211:             For Each objWMIInstance In objWMIInstances
212:                 strUTC = Right ("000" & objWMIInstance.CurrentTimeZone, 3)
213:             Next
214:             Set objWMIInstances = Nothing
215:
216:             If strUTC > 0 Then
217:                strUTC = "+" & strUTC
218:             End If
219:
220:             strValue = strYear & _
221:                        strMonth & _
222:                        strDay & _
223:                        strHours & _
224:                        strMinutes & _
225:                        strSeconds & "." & _
226:                        strMicroSeconds & _
227:                        strUTC
228:          Else
229:             MsgBox "SetVarDate(): Invalid DateTime value", _
230:                    cExclamationMarkIcon, _
231:                    "SWBemDateTime.WSC"
232:          End If
```

```
233:
234:  End Function
...:
...:
...:
673:
674:  ]]>
675:  </script>
676:
677:</component>
```

Sample 5.12 contains only a portion of the complete code because it shows only the code of the *SetVarDate* method. Basically, the logic is always the same as the code consists of the manipulation of a string to extract and concatenate date and time items (day, month, year, hours, minutes, seconds). From line 1 to 146, we recognize the WSC component declaration as we saw in Chapter 1. From line 194, the various functions implementing the desired properties and methods are available. From line 195 to 234, Sample 5.12 shows the *SetVarDate* method logic. The *SetVarDate* method uses WMI to determine the time zone of the system with a class not yet used until now: the *Win32_ComputerSystem*. Basically, the instance of this class exposes the *CurrentTimeZone* property, which gives the time zone of the computer.

5.5 WMI extension for ADSI

Although we do not focus on Active Directory System Interfaces (ADSI) in this book, there is a very interesting particularity related to the combination of ADSI and WMI. ADSI comes with a collection of objects exposing their own properties and methods. As the ADSI implementation allows the object model to be extended, it means that new objects can be added to the model and also existing objects can expose some new properties and methods. This is exactly where the WMI extension comes into the game. With the WMI extension for ADSI, it is possible to manage systems represented by ADSI computer objects retrieved from Active Directory. For instance, when an application performs an LDAP query in Active Directory to retrieve all computers matching a specific name and performs an LDAP bind operation on the retrieved computer, the extension enables the software to obtain an **SWbemServices** object, a WMI object path, or a *Win32_ComputerSystem* instance directly from the ADSI **Computer** object. Table 5.6 contains the properties and methods directly available from the ADSI interface that represent a **Computer** object.

Table 5.6 *The ADSI IADsComputer Interface Properties*

ComputerID	Access: read The globally unique identifier assigned to each machine.
Department	Access: read/write The department (such as OU or organizational unit) within a company that this computer belongs to.
Description	Access: read/write The description of this computer.
Division	Access: read/write The division (such as organization) within a company that this computer belongs to.
Location	Access: read/write The physical location of the machine where this machine is typically located.
MemorySize	Access: read/write The size of random access memory (RAM) in megabytes (MB).
Model	Access: read/write The make/model of this machine.
NetAddresses	Access: read/write An array of NetAddress fields that represent the addresses by which this computer can be reached. NetAddress is a provider-specific **BSTR** composed of two substrings separated by a colon (":"). The left-hand substring indicates the address type, and the right-hand substring is a string representation of an address of that type. For example, TCP/IP addresses are of the form "IP:100.201.301.45. IPX type addresses are of the form "IPX:10.123456.80".
OperatingSystem	Access: read/write The name of the operating system used on this machine.
OperatingSystemVersion	Access: read/write The operating system version number.
Owner	Access: read/write The name of the person who typically uses this machine and has a license to run the installed software.
PrimaryUser	Access: read/write The name of the contact person, in case something needs to be changed on this machine.
Processor	Access: read/write The type of processor.
ProcessorCount	Access: read/write The number of processors.

Table 5.6 *The ADSI IADsComputer Interface Properties (continued)*

Role	Access: read/write The role of this machine, for example, workstation, server, domain controller.
Site	Access: read/write The globally unique identifier identifying the site this machine was installed in. A site represents a physical region of good connectivity in a network.
StorageCapacity	Access: read/write The disk space in megabytes.

Note: At writing time, this extension is available from WMI under Windows NT 4.0, Windows 2000, and Windows XP. However, the security push started at Microsoft early in 2002 has made the future of this extension uncertain. Microsoft removed the extension from Windows.NET Server and does not plan, at writing time, to make it available as a separate download from its Web site. This choice is driven by security reasons because this feature can be quite powerful in some circumstances and is rarely used. As we will see further, Sample 5.16 ("Using the WMI ADSI extension to reboot computers retrieved from an LDAP query") illustrates how easy to use, but powerful, this WMI extension for ADSI can be. We will revisit the Microsoft security push initiative and its impact on WMI in the second book dedicated to WMI, *Leveraging Windows Management Instrumentation (WMI) Scripting* (ISBN 1555582990).

When WMI is installed and ADSI is already present (which is the case for Windows 2000, Windows XP), WMI adds two methods and one property to the ADSI **Computer** object. Table 5.7 contains these new extensions.

Table 5.7 *The WMI ADSI Extension Property and Methods*

Properties	WMIObjectPath	A string value that contains a moniker. The moniker identifies the object path for the WMI computer object.
Methods	GetWMIObject	Returns the WMI computer object identified by the ADSI computer object.
	GetWMIServices	Returns the WMI services object that you can use to perform other WMI actions, such as executing methods or invoking queries.

5.5.1 The *WMIObjectPath* extension

Using the *WMIObjectPath* property, it is possible to retrieve the corresponding WMI moniker of an ADSI **Computer** object. This facilitates the instantiation of the *Win32_ComputerSystem* class by referencing the corresponding moniker. Sample 5.13 shows how to proceed.

Sample 5.13 *Retrieving the corresponding WMI moniker of an ADSI Computer object*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <script language="VBscript">
14:    <![CDATA[
..:
18:    ' ------------------------------------------------------------------------------------
19:    Const cComputerADsPath = "cn=NET-DPEN6400A,ou=Domain Controllers,dc=LissWare,dc=net"
..:
26:    Set objComputer = GetObject("LDAP://" & cComputerADsPath)
27:
28:    WScript.Echo objComputer.WMIObjectPath
29:    Set objWMIInstance = GetObject (objComputer.WMIObjectPath)
30:    Set objWMIPropertySet = objWMIInstance.Properties_
31:
32:    For Each objWMIProperty In objWMIPropertySet
33:        If IsArray (objWMIProperty.Value) Then
34:           For Each varElement In objWMIProperty.Value
35:                   WScript.Echo "  " & objWMIProperty.Name & " (" & varElement & ")"
36:           Next
37:        Else
38:           WScript.Echo "  " & objWMIProperty.Name & " (" & objWMIProperty.Value & ")"
39:        End If
40:    Next
..:
45:    ]]>
46:    </script>
47:  </job>
48:</package>
```

At line 19, the script defines the *distinguishedName* of the ADSI **Computer** object in a constant. As the script is for purely academic purposes, we proceed this way for convenience, but keep in mind that it can be any ADSI **Computer** object from Active Directory (i.e., issued by an LDAP query, as we will see in a subsequent example). Next, line 26 performs the LDAP bind operation to connect to the **Computer** object stored in Active Directory. At line 28, the script uses the WMI ADSI property extension to echo the corresponding WMI moniker. At line 29, the script uses the moniker to obtain the *Win32_ComputerSystem* instance and enumerates its properties (lines 30 to 40). The output is as follows:

```
C:\>WMIObjectPath.wsf
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

WINMGMTS:{impersonationLevel=impersonate}!
          //NET-DPEN6400A/root/cimv2:Win32_ComputerSystem.Name="NET-DPEN6400A"
   AdminPasswordStatus (3)
   AutomaticResetBootOption (True)
   AutomaticResetCapability (True)
   BootOptionOnLimit ()
   BootOptionOnWatchDog ()
   BootROMSupported (True)
   BootupState (Normal boot)
   Caption (NET-DPEN6400A)
   ChassisBootupState (3)
   CreationClassName (Win32_ComputerSystem)
   CurrentTimeZone (120)
   DaylightInEffect (True)
   Description (AT/AT COMPATIBLE)
   Domain (LissWare.Net)
   DomainRole (5)
   EnableDaylightSavingsTime (True)
   FrontPanelResetStatus (3)
   InfraredSupported (False)
   InitialLoadInfo ()
   InstallDate ()
   KeyboardPasswordStatus (3)
   LastLoadInfo ()
   Manufacturer (Compaq)
   Model (Deskpro EN Series)
   Name (NET-DPEN6400A)
   NameFormat ()
   NetworkServerModeEnabled (True)
   NumberOfProcessors (1)
   OEMLogoBitmap ()
   OEMStringArray (CDT v. 1.0)
...
...
...
```

5.5.2 The *GetWMIObject* extension

As with the *GetWMIObject* method, it is possible to directly retrieve the corresponding *Win32_ComputerSystem* instance of the examined **Computer** object without creating an **SWbemServices** object and without referring to the WMI moniker. For example, Sample 5.14 lists all the WMI properties of the same **Computer** object by retrieving the *Win32_ComputerSystem* instance directly from the ADSI **Computer** object.

Sample 5.14 *Retrieving the WMI properties of a Win32_ComputerSystem instance from an ADSI Computer object*

```
1:<?xml version="1.0"?>
.:
8:<package>
9:  <job>
...:
```

```
13:    <script language="VBscript">
14:    <![CDATA[
..:
18:    ' ------------------------------------------------------------------------------
19:    Const cComputerADsPath = "cn=NET-DPEN6400A,ou=Domain Controllers,dc=LissWare,dc=net"
..:
26:    Set objComputer = GetObject("LDAP://" & cComputerADsPath)
27:
28:    Set objWMIInstance = objComputer.GetWMIObject
29:    Set objWMIPropertySet = objWMIInstance.Properties_
30:
31:    For Each objWMIProperty In objWMIPropertySet
32:        If IsArray (objWMIProperty.Value) Then
33:           For Each varElement In objWMIProperty.Value
34:               WScript.Echo "  " & objWMIProperty.Name & " (" & varElement & ")"
35:           Next
36:        Else
37:           WScript.Echo "  " & objWMIProperty.Name & " (" & objWMIProperty.Value & ")"
38:        End If
39:    Next
..:
44:    ]]>
45:    </script>
46:  </job>
47:</package>
```

The script logic is exactly the same as Sample 5.13, but with the *GetWMIObject* method at line 28, the script retrieves the **SWbemObject** representing the *Win32_ComputerSystem* instance directly from the ADSI **Computer** object.

5.5.3 The *GetWMIServices* extension

The last method brought by WMI to ADSI is the *GetWMIServices* method. In the two previous samples, to retrieve the *Win32_ComputerSystem* class instance we used the WMI moniker or we retrieved the instance directly from the ADSI **Computer** object. With the *GetWMIServices* method, it is possible to retrieve the **SWbemServices** object directly from the ADSI **computer** object. As soon as the script has this object created, it is possible to retrieve any manageable instances available from the **Root\CIMv2** namespace of this computer (i.e., the *Win32_Service* instances). See Sample 5.15.

Sample 5.15 *Retrieving the Win32_Service instances from an ADSI Computer object*

```
1:<?xml version="1.0"?>
.:
8:<package>
9:  <job>
..:
13:    <script language="VBscript">
14:    <![CDATA[
```

```
..:
18:    ' ----------------------------------------------------------------------------------
19:    Const cComputerADsPath = "cn=W2K-DPEN6400,ou=Domain Controllers,dc=MyW2KDomain,dc=com"
20:
21:    ' ----------------------------------------------------------------------------------
22:    Const cWMIClass  = "Win32_Service"
..:
28:    Set objComputer = GetObject("LDAP://" & cComputerADsPath)
29:
30:    Set objWMIServices = objComputer.GetWMIServices
31:    Set objWMIInstances = objWMIServices.InstancesOf(cWMIClass)
32:
33:    For Each objWMIInstance in objWMIInstances
34:        WScript.Echo "'" & objWMIInstance.DisplayName & "' is currently " & _
35:                     LCase (objWMIInstance.State) & "."
36:    Next
..:
41:    ]]>
42:    </script>
43:  </job>
44:</package>
```

Again, the logic is exactly the same as the two previous samples, but now the **SWbemServices** object is created (line 30) with the usage of the *GetWMIServices* method associated with the ADSI **Computer** object. Once the **SWbemServices** object available, it represents a connection to the **Root\CIMv2** namespace of the real computer represented by the ADSI **Computer** object.

With the created **SWbemServices** object, the script instantiates a collection (line 31) containing the *Win32_Service* instances available from this computer. Once created, the script enumerates the Windows services available in the collection (lines 33 to 36).

The use of the *GetWMIServices* or *GetWMIObject* methods implies that the default security context of the script allows the connection to the computer name that comes from ADSI. As the ADSI **Computer** object comes from Active Directory, it can be any computer in the forest, and it is likely that some of the computer names are remote.

In the previous samples, it is important to note that these two methods use the default security context (the security context of the user running the script) to obtain the WMI objects:

- An **SWbemServices** for the *GetWMIServices* method
- An **SWbemObject** for the *GetWMIObject* method

If the ADSI bind operation is executed with the **IADsOpenDSObject** interface, it is likely that different credentials are used, in which case the ADSI connection is executed under a different security context than the default security context. In this example, it is fair to think that the **SWbem-**

Services and **SWbemObject** objects obtained from the WMI ADSI extension are also created under the same security context as the ADSI security context. Unfortunately, this is not the case, and the default security context is used despite the credentials provided for the ADSI bind operation. This limitation must be taken into consideration when writing a script that uses the WMI ADSI extension, especially when the remote machine names that come from an ADSI query are accessed because the current security context is used to perform the WMI connection. This implies that the user running the script must have the rights to access the remote machines.

In Sample 5.15, the script retrieves the *Win32_Service* instances available, but it can be any other instances available from this computer. To demonstrate the power of the WMI and ADSI integration, Sample 5.16 implements logic directly applicable for your day-to-day management tasks. Imagine a script that queries Active Directory for a list of computers and, based on that list, reboots the computers. If you use a naming convention in your company that allows the identification of a computer role based on its name, it is quite easy to create an LDAP query filter to retrieve a list of all the print servers in the company or any other server type. Once the LDAP query completes, the script binds to the corresponding ADSI **Computer** object (supposing that you have the necessary rights), from the ADSI object retrieves a WMI connection, and reboots the system. Let's see how Sample 5.16 works.

Sample 5.16 *Using the WMI ADSI extension to reboot computers retrieved from an LDAP query*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <runtime>
14:      <unnamed name="LDAPFilter" helpstring="enter a list of computers as follow
                 '(name=DCUK056)(name=DCBE017)(name=syst*)'" required="true" type="string" />
15:    </runtime>
16:
17:    <reference object="WbemScripting.SWbemLocator"/>
18:
19:    <script language="VBScript" src="..\Functions\ADSearchFunction2.vbs"/>
20:
21:    <script language="VBscript">
22:    <![CDATA[
..:
26:    ' ---------------------------------------------------------------------------------------
27:    ' Defaults used when only a LDAP filter is given for the query
28:    Const cLDAPClassFilter        = "(objectCategory=computer)"
29:    Const cLDAPPropertiesToReturn = "ADsPath"
30:    Const cHowDeepToSearch        = "subTree"
..:
45:    On Error Resume Next
46:
```

```
47:    ' ----------------------------------------------------------------------------
48:    ' Parse the command line parameters
49:    If WScript.Arguments.Unnamed.Count = 0 Then
50:       WScript.Arguments.ShowUsage()
51:       WScript.Quit
52:    End If
53:
54:    strLDAPNameFilter = WScript.Arguments.Unnamed.Item(0)
55:    strLDAPNameFilter = "(&" & cLDAPClassFilter & "(|" & strLDAPNameFilter & "))"
56:
57:    ' ----------------------------------------------------------------------------
58:    Set objRoot = GetObject("LDAP://RootDSE")
59:    strDefaultDomainNC = objRoot.Get("DefaultNamingContext")
60:    Set objRoot = Nothing
61:
62:    WScript.Echo "Applied LDAP filter is:"
63:    WScript.Echo "  (&" & cLDAPClassFilter & strLDAPNameFilter & ")"
64:
65:    WScript.Echo "Querying 'computer' class in Active Directory ..."
66:
67:    Set objResultList = ADSearch ("LDAP://" & strDefaultDomainNC, _
68:                                  strLDAPNameFilter, _
69:                                  cLDAPPropertiesToReturn, _
70:                                  cHowDeepToSearch, _
71:                                  False, _
72:                                           False)
73:
74:    WScript.Echo "Number of 'computer' class objects found is " & _
75:                 objResultList.Item ("RecordCount")
76:    WScript.Echo
77:
78:    intSuccess = 0
79:
80:    For Each objResult in objResultList
81:        intColumnPosition = InStr (objResult, ":")
82:        If intColumnPosition Then
83:           ' Query Active Directory for your computer.
84:           Set objComputer = GetObject(objResultList.Item (objResult))
85:
86:           intIndice = intIndice + 1
87:
88:           ' Use the GetWMIServices method on the DS computer object to
89:           ' retrieve the WMI services object for this computer.
90:           Set objWMIServices = objComputer.GetWMIServices
91:           If Err.Number Then
92:              Wscript.Echo intIndice & _
93:                           ": Failed to get WMI Service object " & _
94:                           "for 'computer' class object : '" & _
95:                           objComputer.Name & "'."
96:              Err.Clear
97:           Else
98:              Wscript.Echo intIndice & _
99:                           ": Getting 'Win32_OperatingSystem' instance " & _
100:                          "for 'computer' class object : '" & _
101:                          objComputer.Name & "'."
102:
103:             Set objWMIInstances = objWMIServices.InstancesOf("Win32_OperatingSystem")
104:             If Err.Number Then
105:                Wscript.Echo intIndice & _
106:                             ": Failed to get 'Win32_OperatingSystem' " & _
107:                             "instance for 'computer' class object : '" & _
108:                             objComputer.Name & "'."
109:                Err.Clear
```

```
110:              Else
111:                 Wscript.Echo intIndice & _
112:                              ": Creation of Win32_OperatingSystem instance successful."
113:                 intSuccess = intSuccess + 1
114:              End If
115:           End If
116:
117:           ' Add the necessary WMI privileges to reboot.
118:           objWMIServices.Security_.Privileges.Add wbemPrivilegeShutdown, True
119:           For Each objWMIInstance in objWMIInstances
120:               objWMIInstance.Reboot
121:               If Err.Number Then
122:                  WScript.Echo intIndice & _
123:                               ": Failed to reboot '" & _
124:                               objWMIInstance.CSName & _
125:                               "' running " & objWMIInstance.Caption & _
126:                               " - Build " & objWMIInstance.BuildNumber & _
127:                               " (" & objWMIInstance.CSDVersion & ")."
128:                  Err.Clear
129:               Else
130:                  WScript.Echo intIndice & _
131:                               ": Successfully rebooted '" & _
132:                               objWMIInstance.CSName & _
133:                               "' running " & objWMIInstance.Caption & _
134:                               " - Build " & objWMIInstance.BuildNumber & _
135:                               " (" & objWMIInstance.CSDVersion & ")."
136:               End If
137:           Next
...:
143:       End If
144:    Next
145:
146:    Wscript.Echo
147:    Wscript.Echo intSuccess & " sucessful reboot on a total of " & intIndice & " computers."
148:    Wscript.Echo
149:
150:    ]]>
151:    </script>
152:  </job>
153:</package>
```

The script uses an ADSI function (line 19) performing the LDAP query and the Windows Script File facilities to read the parameters on the command line (lines 13 to 15 and lines 49 to 52). The command-line parameters are nothing more than the LDAP name of the computers to search in Active Directory. For instance, the script can be used with the following command line:

```
C:\>QueryAndReboot "(name=MyServer01)(name=MyServer02)(name=MyServer03)(name=MyExchange*)"
```

The script reads this line as one parameter and concatenates it with Active Directory *computer* class to create a valid LDAP filter (lines 54 and 55). Since the LDAP filter considers the computer list with an OR logical operator, every Active Directory computer name matching the LDAP name list returns a computer *distinguishedName*. After reading Active Directory default naming context (lines 58 to 60), the script performs the LDAP query (line 67) with the help of the ADSI function included at line 19.

Once the query completes, the script enumerates the results in a "For Each" loop (lines 80 to 144). For each **Computer** object matching the selection criteria, the script binds to Active Directory object (line 84) and retrieves the **SWbemServices** object (line 90). Next, the script performs the instantiation of the *Win32_OperatingSystem* class for the connected computer (line 103) by using the *InstancesOf* method. This method returns a collection that contains one instance of the operating system currently running in the system. Note that it is possible to reference the *Key* property of the *Win32_OperatingSystem* instance. The *Key* is a *Win32_OperatingSystem* class property called *name*, and it contains the name of the booted operating system, the path of the Windows NT directory, and the location of the partition hosting the operating system. Basically, these are the parameters contained in the **BOOT.ini** file of the operating system booted. For instance, the key may contain something like:

```
Microsoft Windows .NET Enterprise Server|J:\WINDOWS|\Device\Harddisk2\Partition3
```

Since this kind of string can make it quite difficult to determine whether all systems in the enterprise are installed in the same way, the script uses the enumeration technique to find the operating system booted. In any case, the enumeration contains only one instance of the operating system, as there is only one operating system running at the time. Doing so, it makes the code more independent of the Windows platforms setup. If this operation is successfully completed, the required privileges to reboot a computer are added to the **SWbemServices** object (line 118). As soon as the code enters the operating system enumeration loop (lines 119 to 137), the considered computer is rebooted (line 120). The rest of the code (lines 121 to 136) contains some error handling and counters to determine the number of rebooted computers of the total found by the LDAP query. This information can be useful as computers found in Active Directory are not necessarily up and running to perform the WMI connection.

5.6 Representing WMI data in XML

Many times throughout this chapter, the WMI instance data is represented from the properties exposed by the WMI instance. This forces us to use a coding technique that lists all properties available to extract the data. As we know, a WMI instance is contained in an **SWbemObject**. Under Windows XP and Windows.NET Server, it is possible to use an **SWbemObject** method called *GetText_* that extracts the data instance in XML. This object is not available under Windows 2000 and before. Sample 5.17 shows how to proceed.

Sample 5.17 *Representing WMI data in XML*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <script language="VBScript" src="..\Functions\WriteData.vbs" />
14:
15:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
16:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMINamedValueSet" />
17:    <object progid="Microsoft.XMLDom" id="objXML" />
18:
19:    <script language="VBscript">
20:    <![CDATA[
..:
24:    Const cComputerName = "LocalHost"
25:    Const cWMINameSpace = "root/cimv2"
26:    Const cWMIClass = "Win32_Service"
27:    Const cWMIInstance = "SNMP"
..:
33:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
34:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
35:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
36:    Set objWMIInstance = objWMIServices.Get (cWMIClass & "='" & cWMIInstance & "'")
37:
38:    objWMINamedValueSet.Add "LocalOnly", False
39:    objWMINamedValueSet.Add "PathLevel", 0
40:    objWMINamedValueSet.Add "IncludeQualifiers", False
41:    objWMINamedValueSet.Add "ExcludeSystemProperties", True
42:
43:    ' wbemObjectTextFormatCIMDTD20 or wbemObjectTextFormatWMIDTD20
44:    strXMLText = objWMIInstance.GetText_(wbemObjectTextFormatWMIDTD20, , objWMINamedValueSet)
45:
46:    WriteData "WMI.XML", strXMLText
47:
48:    objXML.Async = False
49:    objXML.LoadXML strXMLText
50:
51:    WScript.Echo strXMLText
52:    WScript.Echo
53:    WScript.Echo objXML.XML
..:
58:    ]]>
59:
60:    </script>
61:  </job>
62:</package>
```

The script retrieves a single instance of the *Win32_Service* class (lines 33 to 36). As soon as the WMI instance is created, it is possible to extract the data in an XML format by invoking the *GetText_* method (line 44). To perform this operation several parameters can be specified. Some parameters are directly passed with the method invocation (line 44), some others via an **SWbemNamedValueSet** object (lines 38 to 41). The *GetText_* method

requires one value that determines the type of Document Type Definition (DTD) to use for the XML representation. Under Windows XP and Windows.NET Server, WMI comes with two DTDs: one that contains a CIM DTD 2.0 definition and one that contains WMI DTD 2.0 definition. Using a WMI DTD 2.0 definition enables a few WMI-specific extensions, such as embedded objects (important when working with events; see Chapter 6) and scope (domain range of properties and methods declaration in a class; see Chapter 2). The DTD files (**cim20.dtd** and **wmi20.dtd**) are located in %SystemRoot%\System32\wbem\xml. For instance, to get a default representation of the WMI instance retrieved by Sample 5.17, line 44 can be coded as follows:

```
44:    strXMLText = objWMIInstance.GetText_(wbemObjectTextFormatWMIDTD20)
```

Executed this way, the XML representation contains the system properties, the class inheritance, and the values of all properties defined for the *Win32_Service* class. The XML data is assigned to a variable called *strXMLText* (line 44). The content of this variable is saved to a file (line 46) by using a function included in the beginning of the script (line 13). The file created is called **WMI.xml** and can be viewed with any XML viewer. Figure 5.6 shows the created XML file with XML Spy 3.5.

Of course, it is likely that this information is not what is required in the XML representation. In this case, it is necessary to specify another set of parameters. These parameters must be passed to the *GetText_* method via the **SWbemNamedValueSet** object (lines 38 to 41). This object must contain four explicit parameters, listed in Table 5.8.

Table 5.8 *The Context Object Values for an XML Representation*

LocalOnly	When TRUE, only properties and methods locally defined in the class are present in the resulting XML. The default value is FALSE.
IncludeQualifiers	When TRUE, class, instance, properties, and method qualifiers are included in the resulting XML. The default value is FALSE.
ExcludeSystemProperties	When TRUE, WMI system properties are filtered out of the output. The default value is FALSE.
PathLevel	0 = A <CLASS> or <INSTANCE> element is generated. 1 = A <VALUE.NAMEDOBJECT> element is generated. 2 = A <VALUE.OBJECTWITHLOCALPATH> element is generated. 3 = A <VALUE.OBJECTWITHPATH> is generated. The default is 0.

Based on the parameters provided in the script (line 44), the XML file contains only the data of the *Win32_Service* instance: no qualifiers (line 40), no system properties (line 41). The XML representation contains all properties available from the instance. We previously showed that properties can be inherited from a superclass; line 38 specifies that all properties (not only the locally defined) must be included in the XML representation. Because it uses a *PathLevel* equal to zero (line 39), the instance is included without its object path. In this case, the produced XML file looks as follows:

```
 1:<INSTANCE CLASSNAME="Win32_Service">
 2: <PROPERTY NAME="AcceptPause" CLASSORIGIN="Win32_BaseService" TYPE="boolean">
 3:  <VALUE>FALSE</VALUE>
 4: </PROPERTY>
 5: <PROPERTY NAME="AcceptStop" CLASSORIGIN="Win32_BaseService" TYPE="boolean">
 6:  <VALUE>FALSE</VALUE>
 7: </PROPERTY>
 8: <PROPERTY NAME="Caption" CLASSORIGIN="CIM_ManagedSystemElement" TYPE="string">
 9:  <VALUE>SNMP</VALUE>
10: </PROPERTY>
11: <PROPERTY NAME="CheckPoint" CLASSORIGIN="Win32_Service" TYPE="uint32">
12:  <VALUE>0</VALUE>
13: </PROPERTY>
14: <PROPERTY NAME="CreationClassName" CLASSORIGIN="CIM_Service" TYPE="string">
15:  <VALUE>Win32_Service</VALUE>
16: </PROPERTY>
..:
..:
..:
61: <PROPERTY NAME="Status" CLASSORIGIN="CIM_ManagedSystemElement" TYPE="string">
62:  <VALUE>OK</VALUE>
63: </PROPERTY>
64: <PROPERTY NAME="SystemCreationClassName" CLASSORIGIN="CIM_Service" TYPE="string">
65:  <VALUE>Win32_ComputerSystem</VALUE>
66: </PROPERTY>
67: <PROPERTY NAME="SystemName" CLASSORIGIN="CIM_Service" TYPE="string">
68:  <VALUE>XP-DPEN6400</VALUE>
69: </PROPERTY>
70: <PROPERTY NAME="TagId" CLASSORIGIN="Win32_BaseService" TYPE="uint32">
71:  <VALUE>0</VALUE>
72: </PROPERTY>
73: <PROPERTY NAME="WaitHint" CLASSORIGIN="Win32_Service" TYPE="uint32">
74:  <VALUE>0</VALUE>
75: </PROPERTY>
76:</INSTANCE>
```

The XML Spy representation shown in Figure 5.6 differs considerably from the representation shown in Figure 5.7.

Other *PathLevel* values can be used to get different WMI object path information (line 39). The script loads the XML representation issued by WMI in a **document object model XML** (DOMXML) object at line 49. The **DOMXML** object is created at line 17. Once loaded in DOMXML, the scripts displays the XML representation contained in the *strXMLtext* variable (line 51) and in DOMXML (line 53). This validates the WMI

Figure 5.6
An instance representation in XML.

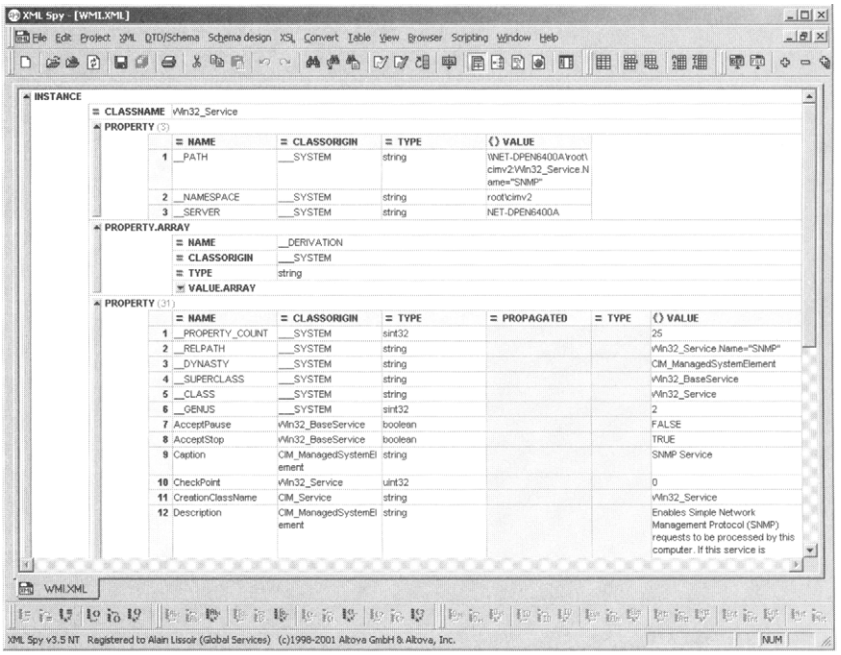
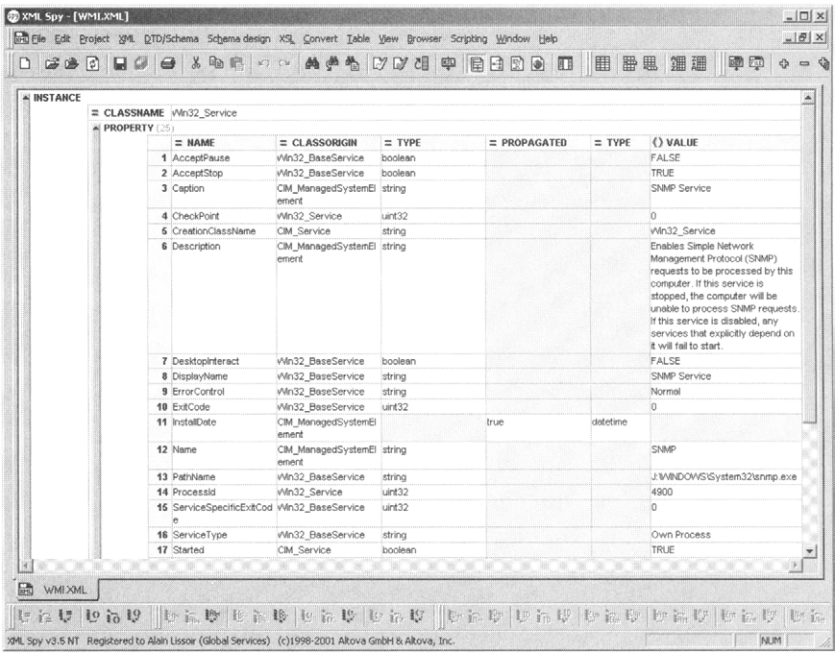


Figure 5.7
Another instance representation in XML.



XML representation against DOMXML. From this point, any XML transformation is applicable (i.e., generating an HTML file of the WMI data with the help of an XML stylesheet).

5.7 Detecting changes with the Refresher object

Given the problem where a WMI script must monitor modifications on a real-world manageable entity, the first idea that may come to mind is to detect modifications by polling the monitored instances. For instance, with our current WMI knowledge, we can imagine the script shown in Sample 5.18.

Sample 5.18 *Looping in a script to detect a change*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    Const cComputerName = "LocalHost"
21:    Const cWMINameSpace = "root/cimv2"
22:    Const cWMIClass = "Win32_Service"
23:    Const cWMIInstance = "SNMP"
..:
29:    Set objWMIInstance = GetObject("WinMgmts:{impersonationLevel=impersonate, " & _
30:                                   "AuthenticationLevel=default}!\\" & _
31:                                   cComputerName & "\" & cWMINameSpace).Get _
32:                                   (cWMIClass & "='" & cWMIInstance & "'")
33:
34:    WScript.Echo "Set the service start mode to 'Disable' to stop the script ..."
35:    WScript.Echo "Watching '" & objWMIInstance.Name & "' service." & vbCRLF
36:
37:    strCurrentState = objWMIInstance.State
38:
39:    For intIndice = 0 To 1000
..:
49:        WScript.Echo Now & " - " & objWMIInstance.State
..:
57:        WScript.Sleep 2000
58:    Next
..:
62:    ]]>
63:   </script>
64:  </job>
65:</package>
```

Unfortunately, this script won't work because it instantiates the instance before the loop (lines 29 to 32). At no time during the loop (between lines

39 to 58) is the state of the instance refreshed. Of course, it is always possible to create the WMI instance inside the loop, but this is not a nice solution as it generates additional resource usage and useless network traffic. Creating or recreating the instance every loop is certainly not a best practice. To get an updated version of the created instance, the **SWbemObject** exposes a method called *Refresh_* (available only under Windows XP and Windows.NET Server). Sample 5.19 shows how to use this method.

Sample 5.19 *Looping in a script to detect a change with the SWbemObject Refresh_ method*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    Const cComputerName = "LocalHost"
21:    Const cWMINameSpace = "root/cimv2"
22:    Const cWMIClass = "Win32_Service"
23:    Const cWMIInstance = "SNMP"
..:
29:    Set objWMIInstance = GetObject("WinMgmts:{impersonationLevel=impersonate, " & _
30:                                   "AuthenticationLevel=default}!\\" & _
31:                                   cComputerName & "\" & cWMINameSpace).Get _
32:                                   (cWMIClass & "='" & cWMIInstance & "'")
33:
34:    WScript.Echo "Set the service start mode to 'Disable' to stop the script ..."
35:    WScript.Echo "Watching '" & objWMIInstance.Name & "' service." & vbCRLF
36:
37:    strCurrentState = objWMIInstance.State
38:
39:    For intIndice = 0 To 1000
40:        objWMIInstance.Refresh_
41:
42:        If strCurrentState <> objWMIInstance.State Then
43:           WScript.Echo "'" & objWMIInstance.Name & "' service state has change from '" & _
44:                        strCurrentState & "' to '" & objWMIInstance.State & "'."
45:        End If
46:
47:        strCurrentState = objWMIInstance.State
48:
49:        WScript.Echo Now & " - " & objWMIInstance.State
50:
51:        If objWMIInstance.StartMode = "Disabled" Then
52:           WScript.Echo "Ending the script because the '" & _
53:                        objWMIInstance.Name & "' service is 'Disabled'."
54:           Exit For
55:        End If
56:
57:        WScript.Sleep 2000
58:    Next
..:
```

```
62:    ]]>
63:   </script>
64:  </job>
65:</package>
```

Basically, it uses the same logic as Sample 5.18. The most important modification concerns the addition of line 40. This line invokes the *Refresh_* method. Beside this new line, some extra logic is added to do the following:

- Display a message when a service state change is detected (lines 37 and 42 to 45)
- Terminate the script if the startup mode of the monitored service is disabled (lines 51 to 55)

This method simply refreshes the created instance (line 40) without the need to reinstantiate the object. If the state of the service instance is modified (i.e., by using the Services snap-in), the script compares the current service state with the previous one (line 42). If there is a difference between the two states, the script shows the difference (lines 43 to 44). Next, the script verifies the service current startup mode and terminates if it is disabled (lines 51 to 55). If the service instance is not disabled, the script continues its loop to repeat the same process. Below is a sample output of the script execution that occurs when the SNMP service is started and stopped.

```
C:\>WatchInstanceStateWithRefreshMethod.wsf
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Set the service start mode to 'Disable' to stop the script ...
Watching 'SNMP' service.

19-07-2001 13:49:09 - Stopped
19-07-2001 13:49:11 - Stopped
'SNMP' service state has change from 'Stopped' to 'Start Pending'.
19-07-2001 13:49:13 - Start Pending
'SNMP' service state has change from 'Start Pending' to 'Running'.
19-07-2001 13:49:15 - Running
19-07-2001 13:49:17 - Running
19-07-2001 13:49:19 - Running
'SNMP' service state has change from 'Running' to 'Stop Pending'.
19-07-2001 13:49:21 - Stop Pending
19-07-2001 13:49:23 - Stop Pending
'SNMP' service state has change from 'Stop Pending' to 'Stopped'.
19-07-2001 13:49:25 - Stopped
19-07-2001 13:49:27 - Stopped
19-07-2001 13:49:29 - Stopped
19-07-2001 13:49:31 - Stopped
19-07-2001 13:49:33 - Stopped
19-07-2001 13:49:35 - Stopped
Ending the script because the 'SNMP' service is 'Disabled'.
```

In Sample 5.19, we poll only one instance. How can we poll several instances? Of course, an easy way would be to create all required instances and refresh each of them in the loop. However, WMI offers a better way to perform this by using an **SWbemRefresher** object. Basically, it is possible to encapsulate different object instances or object collections to refresh in an **SWbemRefresher** object. At a certain point of the script execution, it is necessary only to refresh the **SWbemRefresher** object with its *Refresh* method to refresh all objects that it contains. Sample 5.20 shows how to use the **SWbemRefresher** object.

Sample 5.20 *Refreshing different instances or a collection of instances with the SWbemRefresher object*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemRefresher" id="objWMIRefresher" />
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    Const cComputerName = "LocalHost"
21:
22:    Const cWMINameSpaceItem1 = "Root/CIMv2"
23:    Const cWMIClassItem1  = "Win32_WMISetting"
24:    Const cWMIInstanceItem1 = "@"
25:
26:    Const cWMINameSpaceItem2 = "Root/CIMv2"
27:    Const cWMIClassItem2 = "Win32_Service"
..:
40:    Set objWMIServices1 = GetObject("WinMgmts:{impersonationLevel=impersonate, " & _
41:                                    "AuthenticationLevel=default}!\\" & _
42:                                    cComputerName & "\" & cWMINameSpaceItem1)
43:
44:    Set objWMIServices2 = GetObject("WinMgmts:{impersonationLevel=impersonate, " & _
45:                                    "AuthenticationLevel=default}!\\" & _
46:                                    cComputerName & "\" & cWMINameSpaceItem2)
47:
48:    Set objWMIRefresherItem1 = objWMIRefresher.Add _
49:                (objWMIServices1, cWMIClassItem1 & "=" & cWMIInstanceItem1)
50:    Set objWMIInstance = objWMIRefresherItem1.Object
51:
52:    Set objWMIRefresherItem2 = objWMIRefresher.AddEnum _
53:                (objWMIServices2, cWMIClassItem2)
54:    Set objWMIInstances = objWMIRefresherItem2.ObjectSet
55:
56:    WScript.Echo "Number of items in the refresher is " & objWMIRefresher.Count & "."
57:
58:    objWMIRefresher.AutoReconnect = True
59:
60:    For Each objWMIRefreshableItem In objWMIRefresher
61:        If objWMIRefreshableItem.IsSet Then
```

```
 62:          WScript.Echo "  Item" & objWMIRefreshableItem.Index & " is a collection."
 63:       Else
 64:          WScript.Echo "  Item" & objWMIRefreshableItem.Index & " is a single object."
 65:       End if
 66:    Next
...:
 70:    For intIndice = 0 to 1000
 71:       objWMIRefresher.Refresh
 72:       WScript.Echo vbCRLF & "--------- Pass " & intIndice & ":"
 73:
 74:       Wscript.Echo "Currently WMI Logging level is " & objWMIInstance.LoggingLevel
 75:       WScript.Echo
 76:
 77:       For Each objItem In objWMIInstances
 78:          WScript.Echo "'" & objItem.DisplayName & "' is " & objItem.State & "."
 79:
 80:          If objItem.StartMode = "Disabled" And objItem.Name = "SNMP" Then
 81:             WScript.Echo vbCRLF & "Ending the script because the '" & _
 82:                          objItem.Name & "' service is 'Disabled'."
 83:             boolExitFlag = True
 84:             Exit For
 85:          End If
 86:       Next
 87:
 88:       If boolExitFlag Then
 89:          Exit For
 90:       End If
 91:
 92:       WScript.Sleep (2000)
 93:    Next
...:
103:    ]]>
104:   </script>
105:  </job>
106:</package>
```

Compared with Sample 5.19, Sample 5.20 works differently. The first part performs the connections to the WMI namespaces where classes of the monitored instance reside (lines 40 to 46). The two created objects are **SWbemServices** objects (lines 40 and 44).

To make the script more generic, the script creates the WMI connection twice: once for each class instance or instance collection to monitor. Although not mandatory when the classes are in the same namespace, this makes the script easier to reuse for classes spread among different namespaces (lines 40 to 42 and 44 to 46). It is possible that the two classes are not located in the same WMI namespace, in which case it is necessary to establish the connection to each specific namespace. Once connected to the different namespaces, the **SWbemRefresher** object created at line 13 is initialized with the *Win32_WMISetting* instance (lines 48 and 49) and the *Win32_Service* collection instances (lines 52 and 53). To add *Win32_WMISetting* instance in the **SWbemRefresher**, the *Add* method is

used. To add the *Win32_Service* collection of instances in the **SWbemRefresher**, the *AddEnum* method is used. Note that to initialize the **SWbemRefresher** object, the **SWbemServices** objects representing the connection to the WMI namespaces are also referenced (line 49 and 53). Once initialization is complete, the **SWbemRefresher** object contains a collection of items. Each item is an **SWbemRefreshableItem** object (lines 48 and 52), which may contain a single instance or a collection of instances.

From each **SWbemRefreshableItem** object, it is possible to retrieve the instance or the collection of instances stored in the **SWbemRefresher** object:

- The instance of the *Win32_WMISetting* class can be retrieved from the first **SWbBemRefreshableItem** object with the *Object* property (line 50). This instance is stored in an **SWbemObject** object represented by the *objWMIInstance* variable, as it is a single object instance.
- The instances of the *Win32_Service* class can be retrieved from the second **SWbemRefreshableItem** object with the *ObjectSet* property (line 54). This collection of instances is stored in an **SWbemObjectset** object represented by the *objWMIInstances* variable, as it is a collection of instances.

These two variables are used later in the script to reference the refreshed version of the instances contained in the **SWbemRefreshableItem** objects. To ensure that the WMI connection is reestablished at refresh time if the WMI connection is broken, the **SWbemRefresher** object property *AutoReconnect* is set to True (line 58).

The script shows the nature of the **SWbemRefreshableItem** objects contained in the **SWbemRefresher** object (lines 62 and 64). The script enumerates each item (lines 60 to 66) and tests whether the examined item is a collection by using the *IsSet* property of the **SWbemRefreshableItem**.

Next, the script performs a loop (lines 70 to 86) to show the state of the instances to be refreshed. Line 74 shows the *Win32_WMISetting* instance, and line 78 displays the *Win32_Service* collection instances states in a loop (lines 77 to 86). The script performs one thousand loops (lines 70 to 93) and exits immediately from the loops as soon as the startup mode of the SNMP service instance is disabled (lines 80 to 85 and 88 to 90). Sample output of the script execution while the logging level of WMI is modified (i.e., with the WMI Control MMC) and the SNMP service is started and stopped (i.e., with the Services MMC) is as follows:

```
C:\>WatchInstanceCollectionStateWithRefresher.wsf
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Number of items in the refresher is 2.
  Item1 is a single object.
  Item2 is a collection.


--------- Pass 0:
Currently WMI Logging level is 1

'Alerter' is Running.
'Application Management' is Stopped.
...
'SNMP' is Stopped.
...
'WMI Performance Adapter' is Stopped.
'Automatic Updates' is Running.
'Wireless Zero Configuration service' is Running.

--------- Pass 1:
Currently WMI Logging level is 2

'Alerter' is Running.
'Application Management' is Stopped.
...
'SNMP' is Running.
...
'WMI Performance Adapter' is Stopped.
'Automatic Updates' is Running.
'Wireless Zero Configuration service' is Running.

--------- Pass 2:
Currently WMI Logging level is 2

'Alerter' is Running.
'Application Management' is Stopped.
...
'SNMP' is Running.
...
'WMI Performance Adapter' is Stopped.
'Automatic Updates' is Running.
'Wireless Zero Configuration service' is Running.

--------- Pass 3:
Currently WMI Logging level is 1

'Alerter' is Running.
'Application Management' is Stopped.
...
'SNMP' is Running.
...
```

```
'WMI Performance Adapter' is Stopped.
'Automatic Updates' is Running.
'Wireless Zero Configuration service' is Running.

--------- Pass 4:
Currently WMI Logging level is 1

'Alerter' is Running.
'Application Management' is Stopped.
...
'SNMP' is Stopped.
...
'WMI Performance Adapter' is Stopped.
'Automatic Updates' is Running.
'Wireless Zero Configuration service' is Running.

Ending the script because the 'SNMP' service is 'Disabled'.
```

Table 5.9 shows the **SWbemRefresher** object properties and methods. Table 5.10 shows the **SWbemRefresherableItem** object properties and methods.

Although the polling techniques of the *Refresh_* method or the **SWbemRefresher** and **SWbemRefresherableItem** objects are technically valid from a scripting point of view, they do not represent the most versatile means to watch system modifications. The objects are refreshed from the created instances, which means that the systems hosting these instances are contacted during each loop. This generates some unnecessary network traffic. Moreover, another inconvenience is the impossibility of executing the monitoring in parallel with other tasks. And last but not least, the script uses some CPU cycles regardless of whether an interesting change occurs. We clearly see the need for another technique to monitor changes. This is where the WMI event notification becomes really interesting. We will see that the **SWbemRefresher** object is useful in such a context too, but before

Table 5.9 *The SWBemRefresherableItem Object Properties and Methods*

Properties	Index	Index of the item in the refresher object.
	IsSet	The SWbemRefresher object in which this item resides.
	Object	Indicates whether the item is a single object or an object set.
	ObjectSet	The SWbemObject that represents the items.
	Refresher	The SWbemObjectSet that represents the items.
Methods	Remove	Removes the item from the refresher object.

Table 5.10 *The SWbemRefresher Object Properties and Methods*

Properties	AutoReconnect	Indicates whether the refresher automatically attemps to reconnect to a remote provider if the connection is broken.
	Count	Number of items in the refresher object.
Methods	Add	Adds a new refreshable object to the refresher object.
	AddEnum	Adds a new enumerator to the refresher.
	DeleteAll	Removes all items from the refresher.
	Item	Returns a specified item from the collection in the Refresher.
	Refresh	Refreshs all items in the refresher.
	Remove	Removes object or object set with a specified index from the refresher.

discussing the WMI event notification scripting, we must learn what WMI event notification is. This is covered in the following chapter.

5.8 Summary

The WMI scripting API offers many possibilities for performing various operations. Its versatility adds to its complexity in the sense that there are so many ways to perform an operation that it can sometimes be confusing. As we have said throughout this chapter, we recommend that you practice this technology to get a better working knowledge of the WMI scripting API.

Throughout our examination of WMI scripting, we have seen that it is possible to instantiate CIM classes or real-world manageable entities (mapped on defined CIM classes) by using a moniker or a series of WMI scripting APIs. Moreover, the security can be set at different object levels of the WMI scripting API. In addition to these two aspects, it is also possible to combine the WMI scripting API with WQL queries. Last but not least, the execution of the WMI operation can be performed synchronously or asynchronously. This provides many features and possibilities that can work together or separately. The ability to execute some WMI scripting API operations asynchronously is an important WMI features because it enhances the responsiveness of a script (or any application using the asynchronous calls). Conceptually, asynchronous operations are a big step in WMI event monitoring. The event monitoring makes extensive usage of the asynchronous features discussed here in combination with the WQL queries. In the following chapter, through different script samples, we combine the power of WQL with WMI asynchronous operations.