Chapter 8

Understanding WMI Classes

Classes provide the core functionality of Windows Management Instrumentation (WMI). But how are classes organized? Most people do not realize that there is a pattern to the way WMI classes are stored in the hierarchy. Once you recognize this pattern, you can open new vistas in your scripting life. You can achieve self-actualization as a seasoned IT pro. You can...OK, you get the idea.

Before You Begin

To work through this chapter, you should be familiar with the following concepts:

- The basics of WMI scripting
- The concept of WMI namespaces
- How WMI providers work

After you complete this chapter, you will be familiar with the following concepts:

- The use of WMI system classes
- The basics of namespace security
- How to set namespace security through scripting
- How to determine effective user rights into a WMI connection

Note All the scripts used in this chapter are located on the CD that accompanies this book in the \Scripts\Chapter08 folder.

Using the System Classes

The system classes in WMI are all based on the Common Information Model (CIM) discussed in Chapter 1. As you might recall, the CIM classes are designed to be copied—like a book of patterns or templates. Some system classes always reside in every WMI namespace. These system classes are part of the core WMI functionality and are therefore not described in a Man-

aged Object Format (MOF) file. If you create a new WMI namespace, these core system classes are copied there when the namespace is built. The core system classes are used to provide some basic functionality for WMI. They perform the following activities:

- Event and provider registration
- Security
- Event notification

If you are wondering how you will be able to identify a system class, it is actually very easy—all system classes are preceded with a double underscore (__). If you ever write your own WMI class, make sure you do not give it a name that is preceded by a double underscore because Mofcomp.exe ignores class names that begin with a double underscore. WMI reserves this naming convention for WMI system class names.

Abstract Base Classes

The abstract base classes are classes that can serve only as the basis for a new class. You cannot create an instance of an abstract class. If a class is an abstract class, the abstract qualifier is set. You never use these classes in creating another WMI class. The only class that could even be used to derive another class is the *__NotifyStatus* class, and if you need to do notification actions, you are better off using the *__ExtendedStatus* class instead because it has far more properties. The *__SystemClass* is used as the basis for all of the system classes that are not in the following list. You cannot directly derive a class from the *__SystemClass*, but that is not really an issue because if you are interested in the properties available from *__SystemClass*, you can derive a class from a system class already derived from *__SystemClass*. (The *__PARAMETERS* class shows up as all caps in the WMI namespaces, so I followed that convention.) The following are the abstract system base classes:

- *__NotifyStatus*
- *__PARAMETERS*
- *__SecurityRelatedClass*
- *__SystemClass*
- *__SystemSecurity*

Using System Classes as Base Classes

Some system classes built into WMI perform vital day-to-day functions behind the scenes. Most of these system classes stand alone—you cannot inherit properties from them, and you cannot use their methods. A few, indeed, a very small subset of system WMI classes, are willing to share their properties, methods, and events. These system classes can be used as base classes to enable you easily to create new event consumer types, event types, or error object types. Additionally, you can use the *__Namespace* system class to create a new namespace in

WMI. This might be useful if you were creating a number of new WMI classes used to manage a network. It might be a good idea to keep all these classes together in their own namespace. That way, if things get out of hand, you can easily delete the entire namespace and roll back to a previous level of functionality in your WMI infrastructure.

Identifying the Version of WMI

Most of the system classes are not usable as base classes for derived classes. This means you simply cannot use them when you are trying to derive additional classes. These are classes that seem to form core WMI functionality and would not make sense to be used as base classes. This does not prevent you from using the classes, however.

The *__CIMOMIdentification* class provides good troubleshooting information about WMI on your machine. It tells you when WMI was installed, the version that is currently running, and even the version that was used to create the database. You can use this class in your script just as you would use any other class. This is illustrated in the script CimomIdentification.vbs. Because there is only one instance of WMI running on a machine at a time, you can use the shorthand name @ to tell the script you want to retrieve the current running instance of WMI. This shortcut refers to the current instance of WMI, making it very easy to use the *Get* method instead of *ExecQuery* and having to loop the instance. To make the script a little easier to use, I build a single variable *strOut* to use for the output. This is better than having to use *Wscript.echo* many times to return the information retrieved from the query.

CimomIdentification.vbs

```
strComputer = "."
wmiNS = "\root\default"
wmiQuery = "__CIMOMIdentification=@"'only one instance of cimom
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set objItem = objWMIService.get(wmiQuery)

With objItem
    strOut = "setupTime: " & .setupTime
    strOut = strOut & vbcrlf &"setupDate: " & .setupDate
    strOut = strOut & vbcrlf &"VersionCurrentlyRunning: " & .VersionCurrentlyRunning
    strOut = strOut & vbcrlf &"versionUsedToCreateDB: " & .versionUsedToCreateDB
    strOut = strOut & vbcrlf &"WorkingDirectory: " & .WorkingDirectory
end With

WScript.echo strOut
```

Working with System Security

Another useful base class is the *__SystemSecurity* class. Once again, this class is not usable as a base class for other classes, but it does provide valuable information. Several methods are exposed by the *__SystemSecurity* class. These methods enable you to set security access, get security permissions, and identify security privileges held by the user trying to make the WMI connection.

Displaying the Security Information

In the script DisplaySecurityDescriptor.vbs, I use the *GetSD* method from the *__SystemSecurity* class to obtain the security descriptor of a WMI namespace to which I am connected. In this case, I am connected to the *root\wmi* namespace, but the script works with any namespace that exists on your computer. The interesting thing about this particular script is you can use it to set the security descriptor on another namespace or on another machine. This is actually the easiest way to set namespace security in WMI.

DisplaySecurityDescriptor.vbs

```
strComputer = "."
wmiNS = "\root\wmi"
wmiQuery = "__SystemSecurity=@"
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set objItem = objWMIService.Get(wmiQuery)
intRTN = objItem.getSD(arrSD)
For I = 0 To UBound(arrSD)
intSD = intSD & arrSD(i)
    If I <> Ubound(arrSD) Then
      intSD = intSD & ","
       End If
Next
WScript.echo intSD
```

As shown in Figure 8-1, the Security tab can be accessed from the WMI Control Properties dialog box. Once you select the Security tab, the Security For dialog box appears. This enables you to add users or modify security already in effect for groups or individuals.

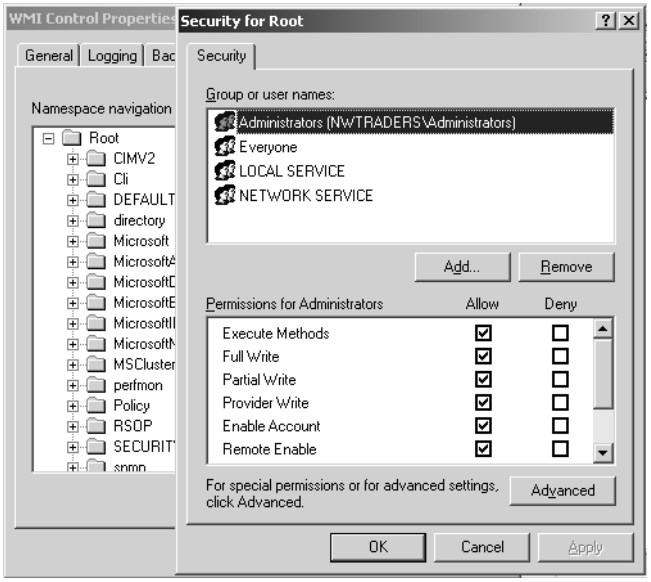

Figure 8-1 Manually setting security on namespaces in the WMI Security dialog box

Setting the Security Information

Once you have retrieved the security descriptor by using the DisplaySecurityDescriptor.vbs script, you can use the returned value to set the security on another WMI namespace—either on the same computer or on another computer—if the same users exist. The procedure is something like the following:

1. Use the WMI Control Properties dialog box to configure security on the namespace in the manner you wish it to be. Add users and grant rights as required.
2. Use the DisplaySecurityDescriptor.vbs script to retrieve the security descriptor.
3. Use the SetSecurityDescriptor.vbs script to set security on the target namespace or computer.
4. Open the WMI Control Properties dialog box on the target computer or in the namespace to ensure that rights are as expected.

Keep in mind the security descriptor is returned as an array data type. You have to use the *Array* command when you set it using the *SetSD* method. You can combine the two scripts, retrieve the security descriptor from one computer, and use it to set the security descriptor on another computer so that you are able to share the security descriptor value between the two computers and not have to type it in (or paste it in) as shown in the SetSecurityDescriptor.vbs script.

SetSecurityDescriptor.vbs

```
strComputer = "."
wmiNS = "\root\cimv2"
wmiQuery = "__SystemSecurity=@"
arSD= array(1,0,4,128,184,0,0,0,200,0,0,0,0,0,0,0,20,0,0,0,2,0,164,0,6,0,0,0,0,0,36,
0,35,0,0,0,1,5,0,0,0,0,0,5,21,0,0,0,182,68,228,35,192,133,56,93,22,192,234,50,248,
3,0,0,0,2,36,0,32,0,2,0,1,5,0,0,0,0,0,5,21,0,0,0,160,101,207,126,120,75,155,95,231,
124,135,112,149,89,0,0,0,18,24,0,63,0,6,0,1,2,0,0,0,0,0,5,32,0,0,0,32,2,0,0,0,18,
20,0,19,0,0,0,1,1,0,0,0,0,0,1,0,0,0,0,0,18,20,0,19,0,0,0,1,1,0,0,0,0,0,5,20,0,0,
0,0,18,20,0,19,0,0,0,1,1,0,0,0,0,0,5,19,0,0,0,1,2,0,0,0,0,0,5,32,0,0,0,32,2,0,0,
1, 2,0,0,0,0,0,5,32,0,0,0,32,2,0,0)
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.Get(wmiQuery) ' note using Get not ExecQuery

WScript.Echo "Preparing to change the SD"

SubChangeSD

Sub SubChangeSD
errReturn = colItems.SetSD(arSD)
If Err <> 0 Then
   WScript.Echo "Method returned error " & errReturn
Else
WScript.Echo "SD was changed"
End If
End sub
```

Identifying the Caller's Rights

To identify the rights a user has in a namespace, you can use the *GetCallerAccessRights* method. All users have the right to call this method because it is required to enable them to determine whether they are allowed into the namespace. The rights are returned in hexadecimal and are additive. Table 8-1 lists the rights and the hexadecimal values that come back from the *GetCallerAccessRights* method. Because the values are additive, if you had only the *WBEM_ENABLE* and the *WBEM_METHOD_EXECUTE* access rights, for instance, *GetCallerAccessRights* would return 3.

Table 8-1 System Security Access Rights

Name	Value	Meaning
WBEM_ENABLE	0x1	Enables the account and grants the user read permissions; default access right for all users
WBEM_METHOD_EXECUTE	0x2	Allows the execution of methods
WBEM_FULL_WRITE_REP	0x4	Allows write to classes and instances except for system classes
WBEM_PARTIAL_WRITE_REP	0x8	Allows write to provider instances but not static classes or static instances to the repository
WBEM_WRITE_PROVIDER	0x10	Allows write to classes and instances to providers
WBEM_REMOTE_ACCESS	0x20	Allows remote operations granted by the permissions set by other bits
READ_CONTROL	0x20000	Allows read access to the security descriptors
WRITE_DAC	0x40000	Allows write access to discretionary access control lists (DACLs)

The script GetCallerRights.vbs uses the *ConnectServer* method of the *SWbemLocator* object. If you try to use this method with the moniker, the only thing that returns is a 0 (no problem, but no answer, either). The other thing that is a little strange about the GetCallerRights.vbs script is the use of an output variable. In the line *errRTN = objItem.GetCallerAccessRights(intRights)*, you use the variable *intRights* to hold the output from running the *GetCallerAccessRights* command. The variable *errRTN* holds the actual return code that comes back from running the command. In most instances, it should be 0 (no errors) because all users should have the ability to run *GetCallerAccessRights*.

GetCallerRights.vbs

```
strComputer = "."
wmiNS = "\root\cimv2"
wmiQuery = "__SystemSecurity"
strUsr =""'Blank for current security. Domain\Username
strPWD = ""'Blank for current security.
strLocl = "MS_409" 'U.S. English. Can leave blank for current language
strAuth = ""'if specify domain in strUsr, this must be blank
iFlag = "0" 'only two values allowed here: 0 (wait for connection),
128 (wait max two min)
```

```
Set objLocator = CreateObject("WbemScripting.SWbemLocator")
Set objWMIService = objLocator.ConnectServer(strComputer, _
     wmiNS, strUsr, strPWD, strLocl, strAuth, iFlag)
Set objItem = objWMIService.get(wmiQuery)
errRTN = objItem.GetCallerAccessRights(intRights)

If errRTN = 0 then
WScript.Echo "Calling users rights: " & intRights
Else
WScript.echo "error occurred. It was: " & errRTN
End if
```

Quick Check

Q: When using the *__CIMOMIdentification* class to retrieve information about the version of WMI, what does *__CIMOMIdentification*=@ mean?

A: When using the *__CIMOMIdentification* class to retrieve information about the version of WMI, *__CIMOMIdentification*=@ means to use the current version of the installed instance of WMI.

Q: Why might you simply receive a zero when trying to retrieve the effective calling user rights by using the *GetCallerAccessRights* method from the *__SystemSecurity* class?

A: You might receive a zero when trying to retrieve the effective calling user rights by using the *GetCallerAccessRights* method from the *__SystemSecurity* class for one of three reasons: you are using the WMI moniker instead of the *SWbemLocator* method; you are echoing out the return code instead of the output variable value; or the caller has no rights.

Understanding the CIM Classes

There are many CIM classes. For example, there are 286 CIM classes in the *root\cimv2* namespace in a Microsoft Windows XP workstation installation. In many cases, CIM classes look exactly the same as Win32 classes. These are the base classes upon which WMI in Microsoft Windows is built. In this regard, they are organized in much the same way as the Win32 classes are. The interesting thing is the way they are used. If you run a script that queries *CIM_Card*, such as the script CIM_Card.vbs, you might see such information as the serial number and other data about the cards installed on the computer—provided the hardware maker supports the class.

CIM_Card.vbs

```
strComputer = "."
wmiNS = "\root\cimv2"
wmiQuery = "Select * from cim_card"
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.ExecQuery(wmiQuery)

For Each objItem in colItems
   Wscript.echo "Caption:"  & objItem.Caption
Wscript.echo "CreationClassName:"  & objItem.CreationClassName
```

```
Wscript.echo "Depth:"  & objItem.Depth
Wscript.echo "Description:"  & objItem.Description
Wscript.echo "Height:"  & objItem.Height
Wscript.echo "HostingBoard:"  & objItem.HostingBoard
Wscript.echo "HotSwappable:"  & objItem.HotSwappable
Wscript.echo "InstallDate:"  & objItem.InstallDate
Wscript.echo "Manufacturer:"  & objItem.Manufacturer
Wscript.echo "Model:"  & objItem.Model
Wscript.echo "Name:"  & objItem.Name
Wscript.echo "OtherIdentifyingInfo:"  & objItem.OtherIdentifyingInfo
Wscript.echo "PartNumber:"  & objItem.PartNumber
Wscript.echo "PoweredOn:"  & objItem.PoweredOn
Wscript.echo "Removable:"  & objItem.Removable
Wscript.echo "Replaceable:"  & objItem.Replaceable
Wscript.echo "RequirementsDescription:"  & objItem.RequirementsDescription
Wscript.echo "RequiresDaughterBoard:"  & objItem.RequiresDaughterBoard
Wscript.echo "SerialNumber:"  & objItem.SerialNumber
Wscript.echo "SKU:"  & objItem.SKU
Wscript.echo "SlotLayout:"  & objItem.SlotLayout
Wscript.echo "SpecialRequirements:"  & objItem.SpecialRequirements
Wscript.echo "Status:"  & objItem.Status
Wscript.echo "Tag:"  & objItem.Tag
Wscript.echo "Version:"  & objItem.Version
Wscript.echo "Weight:"  & objItem.Weight
Wscript.echo "Width:"  & objItem.Width

Next
```

Close examination of the results will reveal that the query is actually fulfilled by the *Win32_BaseBoard* class. The output tells you this in the *CreationClassName* property value returned by the script. To verify the results, look at another script. In the Win32_Baseboard.vbs script, there are 29 properties. The *CIM_Card* class has only 27 properties. The output for CIM_Card.vbs returns values only for 27 properties—even though the query is actually serviced by the *Win32_BaseBoard* class. The two unique properties supplied by the *Win32_BaseBoard* class, *ConfigOptions* and *Product*, are returned only if the query is made directly against the *Win32_BaseBoard* class as shown in the Win32_Baseboard.vbs script. The *ConfigOptions* property is returned as an array, so you need to use the *Join* function to turn it into a string that you can easily print out.

Win32_Baseboard.vbs

```
strComputer = "."
wmiNS = "\root\cimv2"
wmiQuery = "Select * from win32_BaseBoard"
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.ExecQuery(wmiQuery)

For Each objItem in colItems
Wscript.echo "Caption:"  & objItem.Caption
Wscript.echo "ConfigOptions:"  & JOIN (objItem.ConfigOptions)
Wscript.echo "CreationClassName:"  & objItem.CreationClassName
Wscript.echo "Depth:"  & objItem.Depth
Wscript.echo "Description:"  & objItem.Description
```

```
Wscript.echo "Height:"  & objItem.Height
Wscript.echo "HostingBoard:"  & objItem.HostingBoard
Wscript.echo "HotSwappable:"  & objItem.HotSwappable
Wscript.echo "InstallDate:"  & objItem.InstallDate
Wscript.echo "Manufacturer:"  & objItem.Manufacturer
Wscript.echo "Model:"  & objItem.Model
Wscript.echo "Name:"  & objItem.Name
Wscript.echo "OtherIdentifyingInfo:"  & objItem.OtherIdentifyingInfo
Wscript.echo "PartNumber:"  & objItem.PartNumber
Wscript.echo "PoweredOn:"  & objItem.PoweredOn
Wscript.echo "Product:"  & objItem.Product
Wscript.echo "Removable:"  & objItem.Removable
Wscript.echo "Replaceable:"  & objItem.Replaceable
Wscript.echo "RequirementsDescription:"  & objItem.RequirementsDescription
Wscript.echo "RequiresDaughterBoard:"  & objItem.RequiresDaughterBoard
Wscript.echo "SerialNumber:"  & objItem.SerialNumber
Wscript.echo "SKU:"  & objItem.SKU
Wscript.echo "SlotLayout:"  & objItem.SlotLayout
Wscript.echo "SpecialRequirements:"  & objItem.SpecialRequirements
Wscript.echo "Status:"  & objItem.Status
Wscript.echo "Tag:"  & objItem.Tag
Wscript.echo "Version:"  & objItem.Version
Wscript.echo "Weight:"  & objItem.Weight
Wscript.echo "Width:"  & objItem.Width
Next
```

CIM Classes Are Really DMTF Classes

The CIM classes were devised by the Distributed Management Task Force (DMTF), and they form the basis of the WMI schema you use in the Windows operating system. In many cases, a CIM class is used as a basis for a Win32 class that performs a similar function within the schema. However, three-quarters of the CIM classes do not have a direct relation with a Win32 counterpart.

Consider the *MonitorResolution* Class

Suppose you are perusing the CIM classes and you run across the *CIM_MonitorResolution* class. You think it looks nice, and you whip out the cimMonitorResolution.vbs script. You are excited with the prospects of being able to utilize the information directly. When you run the script, what to your wondering eyes should appear? Nothing!

cimMonitorResolution.vbs

```
strComputer = "."
wmiNS = "\root\cimv2"
strFile = "'%boot.ini%'"
wmiQuery = "Select * from CIM_MonitorResolution"
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.ExecQuery(wmiQuery)

For Each objItem in colItems
Wscript.echo "Caption:"  & objItem.Caption
Wscript.echo "Description:"  & objItem.Description
```

```
Wscript.echo "HorizontalResolution:"  & objItem.HorizontalResolution
Wscript.echo "MaxRefreshRate:"  & objItem.MaxRefreshRate
Wscript.echo "MinRefreshRate:"  & objItem.MinRefreshRate
Wscript.echo "RefreshRate:"  & objItem.RefreshRate
Wscript.echo "ScanMode:"  & objItem.ScanMode
Wscript.echo "SettingID:"  & objItem.SettingID
Wscript.echo "VerticalResolution:"  & objItem.VerticalResolution
Next
```

To confirm your suspicions, you use the Windows Management Instrumentation Tester (Wbemtest.exe) and look up the *CIM_MonitorResolution* class in the *root/cimv2* namespace. The steps to do this are as follows:

1. Click Start, click Run, and type **WbemTest.exe**.
2. Click Connect.
3. Change the namespace from *root\default* to *root\cimv2*, and then click Connect.
4. Click Open Class, and type the name of the WMI class: **CIM_MonitorResolution**.
5. Click Instances.

The window shown in Figure 8-2 confirms you have no instances of the *CIM_Monitor-Resolution* class implemented on your system.

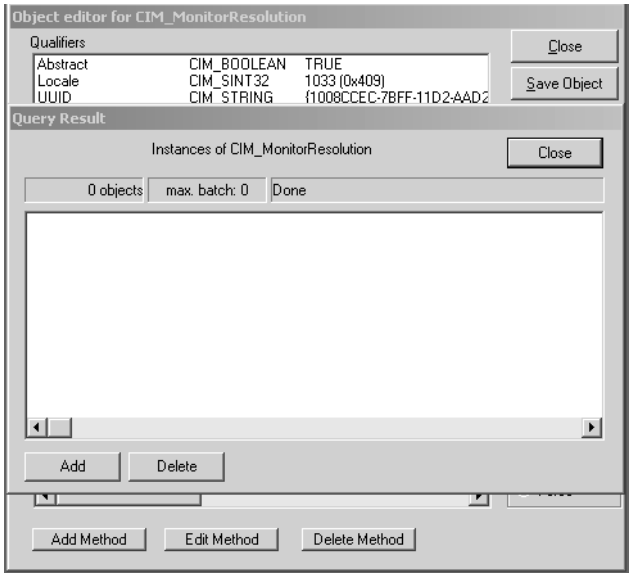

Figure 8-2 Determining whether instances of a class exist on the computer

If you want to see whether another class is using *CIM_MonitorResolution* as a base class and is inheriting all those wonderful properties, you can again turn to Wbemtest.exe. Open the *CIM_MonitorResolution* class as indicated in the preceding steps. Once it is open, click the Derived button on the right side of the screen, and the window shown in Figure 8-3 appears.

It is immediately obvious your investigation has come to an end—there are no derived classes. *CIM_MonitorResolution* is not implemented in any way, shape, or form on your system.

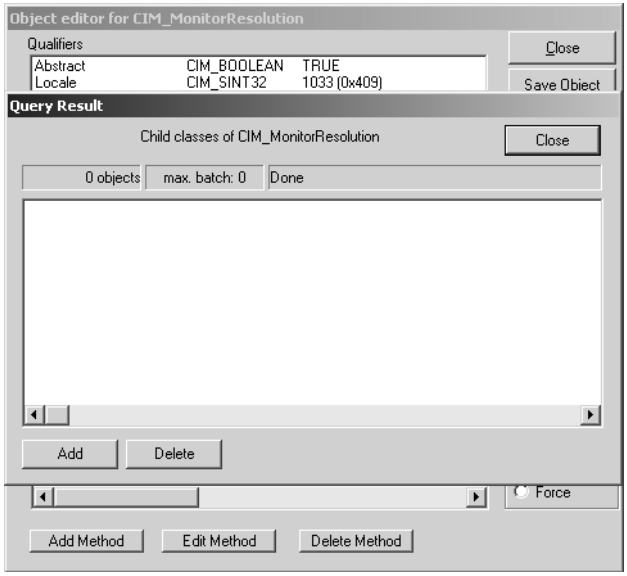

Figure 8-3 Determining whether derived classes are in use on a computer

Quick Check

Q: What tool can you use to determine quickly whether an instance of a class exists on your system?

A: The tool you can use to determine quickly whether an instance of a class exists on your system is the Windows Management Instrumentation Tester (Wbemtest.exe). It is always available on your computer.

Q: Why is it important to determine whether an instance of a class exists on your computer?

A: It is important to determine whether an instance of a class exists on your computer because, if there are no instances, you are not able to perform a query of that class.

Summary

In this chapter, we looked at the way WMI classes are put together. We examined abstract base classes and saw how they can be used in the formation of other WMI classes. We discussed the system classes that WMI uses as the building blocks of WMI, as well as those that are used to control the behavior of WMI. We determined which of these system classes you can use to build other classes as required. Finally, we examined the concept of instances.

Quiz Yourself

Q: What is the difference between an abstract WMI class and a regular WMI class?

A: The difference between an abstract WMI class and a regular WMI class is that an abstract WMI class cannot have any instances.

Q: What does it mean for a WMI class to have instances?

A: When a WMI class has instances, it means the class is active on the computer and you can query the properties of the class and use any methods it has implemented.

Q: What does it mean if a WMI class does not have instances?

A: If a WMI class does not have any instances, it might exist in the schema as an abstract concept, but it has not been implemented into reality. If the class does not have instances, you are not able to query against it.

Q: You want to use the *__CIMOMIdentification* class to determine the version of WMI that is running. How can you use the *Get* method to retrieve this information?

A: If you want to use the *__CIMOMIdentification* class to determine the version of WMI that is running using the *Get* method, you will need to use a query that specifies *__CIMOMIdentification=@*.

Q: If you want to identify the security privileges a user has when making a WMI connection, which WMI class could you use?

A: To determine the security privileges a user has when making a WMI connection, you could use the *__SystemSecurity* class.

On Your Own

Lab 17 Exploring Abstract Classes

In this lab, you will build a script that generates a list of abstract classes in a particular WMI namespace.

1. Open the ConnectorServerTemplate.vbs script from the Lab 17 folder. Save it as StudentLab17.vbs.
2. In the header section of the script, add some additional variables to be used in the script. You need five additional variables: *colClasses*, *strClass*, *a*, *strMsg*, and *strTab*.
3. Just under the variable declarations, make *strTab* equal to a carriage return line feed and a tab stop. Your code should look like the following:

```
strTab = VBcrlf & vbtab
```

4. At the bottom of the reference section where you assign values to the variables, just below the *iFlag="0"* line, assign the value for *strMsg*. It will print out a line that heads the list of abstract classes.

```
strMsg = "The following are abstract classes in the "_
    & wmiNS & " namespace"
```

5. Open the SeparatorLinefunction.vbs script, and copy the *FunLine* function from the bottom of that script. The function looks like the following:

```
Function funLine(lineOfText)
Dim numEQs, separator, i
numEQs = Len(lineOfText)
For i = 1 To numEQs
    separator = separator & "="
Next
 FunLine = lineOfText & vbcrlf & separator
End function
```

6. Go back to the StudentLab17.vbs script, and paste the *FunLine* function at the bottom of your script.

7. Now that you have the separator line function in place, go back to the *strMsg* line and use the function to underline the title of the report. The modified line of code looks like the following:

```
strMsg = funLine("The following are abstract classes in the "_
    & wmiNS & " namespace")
```

8. Set the *colClasses* variable to hold the collection of subclasses that comes back from using the *SubClassesOf* method of the *SWbemObject*. It looks like the following:

```
Set colClasses = objWMIService.SubclassesOf()
```

9. Change the *For Each objItem in ColItems* line to read *For Each strClass in colClasses*.

10. Delete all the *Wscript.echo* commands.

11. Save and run the script. At this point, you should not see any errors. If you do, you need to resolve them before continuing. (You can always look at the solution if you need to.)

12. Set the variable *objItem* equal to what comes back from using the *Get* method of the *SWbemObject* to get the *Class* property from *Path_*. This code looks like the following:

```
Set objItem = objWMIService.get(strClass.path_.class)
```

13. Use *For Each Next* to walk through the collection of qualifiers. Use the variable *a* as the counter. You get the collection of qualifiers by using the *Qualifiers_* property of *objItem* retrieved in the previous step. The code looks like the following:

```
For Each a In objItem.Qualifiers_
```

14. Use the *Instr* command to filter out the word *Abstract*. This will be part of an *If Then End If* command. It looks like the following:

```
If InStr(1,a.name,"abstract",1) Then
End If
```

15. You want to build an output variable so that the results come out in a single command. Use *strMSG*, which at this point contains the title. Now add it to itself, use *strTab*, and pick up the class name from the *Path* object. The line that does this goes between the *If Then* command you used for the *Instr* command.

```
strMsg= strMsg & strTab & objItem.path_.class
```

16. Make sure you have closed all the *For Next* loops you used. The bottom section of the code looks like the following:

```
For Each strClass in colClasses
  Set objItem = objWMIService.get(strClass.path_.class)
  For Each a In objItem.Qualifiers_
If InStr(1,a.name,"abstract",1) Then
strMsg= strMsg & strTab & objItem.path_.class
End if
Next
Next
```

17. Save and run the script. It should work just fine.

Lab 18 Examining WMI Classes

In this lab, you will use the Windows Management Instrumentation Tester (Wbemtest.exe) to examine several WMI classes.

1. Launch Wbemtest.exe from a command-prompt window.
2. Once Wbemtest.exe is running, you need to connect to a WMI namespace. Click Connect.
3. The Connect dialog box appears. Root\Default is highlighted as the default namespace. Unfortunately, setting a new default is not configurable. Change to the *root\cimv2* namespace, and click Connect. You can leave all the other parameters for the connection set to the default values.
4. Run the Lab17solution.vbs script to generate a list of abstract classes in the *root\cimv2* namespace.
5. Find *CIM_PhysicalMemory* on the list. This indicates it has the abstract qualifier set.
6. In Wbemtest.exe, click Open Class, and type in the **CIM_PhysicalMemory** class, and click OK.

7. The Object Editor For *CIM_PhysicalMemory* dialog box appears. In the upper pane, examine the qualifiers that are set for this class. The qualifier you are looking for is called *Abstract*. Is it present? It is. What is the value assigned to the qualifier? It is set to *True*.
8. Look through the properties of the class. They are enumerated in the middle pane.
9. Click Instances on the right side of the Object Editor dialog box. Are there any instances listed? No.
10. Close that dialog box, and click Derived. Are there any classes derived from *CIM_PhysicalMemory*? Yes. This indicates the class is used as a base class for *Win32_PhysicalMemory*.
11. Now you are going to explore two related classes in more detail. You can choose any association class from the earlier list and see if it has a class derived from it by using Wbemtest.exe. If you cannot find something, and your computer has a modem, you can use *CIM_PotsModem* and *Win32_PotsModem*.
12. Open the CompareClasses.vbs script. Run it, and type in **CIM_PotsModem,Win32_PotsModem**. This line needs to be exact because no error checking is included in the script to filter out the input. Basically, this script compares two classes that are derived from each other and prints out the unique properties.
13. Notice at the top of the output how many properties *CIM_PotsModem* has and how many properties *Win32_PotsModem* has (36 versus 79).
14. Open the script PropertyExplorer.vbs, and then open and make a copy of the WmiTemplate.vbs script.
15. Save the copy of the WmiTemplate.vbs script as StudentLab18A.vbs.
16. Go back to the PropertyExplorer.vbs script and run it. At the prompt, type **CIM_PotsModem**, and click OK. A list of 36 *Wscript.Echo* commands will be returned.
17. Copy all the *Wscript.Echo* commands, and paste them into the StudentLab18A.vbs script in the middle of the *For Next* loop. Replace the existing *Wscript.Echo* commands with this text.
18. Change the *wmiQuery* command so that it is selecting everything from *CIM_PotsModem*.
19. Run the script. It should run fine and return a decent amount of information.
20. Determine where the StudentLab18A.vbs script is getting its information. Examine the output for the *CreationClassName* property. You will see something that looks like the following:

```
CreationClassName:Win32_PotsModem
```

21. By using this class, you are really using only a subset of the *Win32_PotsModem* class.

22. Write another script that uses all of the available properties of the class. Open another copy of the WmiTemplate.vbs script, and save it as StudentLab18B.vbs.
23. Run the PropertyExplorer.vbs script. This time feed in *Win32_PotsModem*.
24. Copy all the *Wscript.Echo* commands from the output.
25. Go back to the StudentLab18b.vbs script, and replace the existing *Wscript.Echo* commands from the template with the ones copied from the output of the PropertyExplorer.vbs script.
26. Change the *wmiQuery* so that it is pointing to the *Win32_PotsModem* class.
27. Save and run the script. You should see quite a bit more information from this script.
28. If you look at the top of the script, you will see you have *On Error Resume Next* turned on. Turn off this command, and run the script. Now you will notice there is a problem—three, to be exact. Three of the properties are stored as an array.
29. Two of these properties are right next to one another: *DCB* and *Default*. Use the *Join* command so you can easily print out their values. The modified lines look like the following:

```
Wscript.echo "DCB:"  & join(objItem.DCB)
Wscript.echo "Default:"  & join(objItem.Default)
```

30. The third array property in this class is the *Properties* property. Again, use the *Join* command. The modified line of code now looks like the following:

```
Wscript.echo "Properties:"  & Join(objItem.Properties)
```

31. Run the script again. Look for the output of the *CreationClassName* property. See where it is pointing to: the *Win32_PotsModem* class.

This concludes this lab.