13

Managing Windows Server with Window Management Instrumentation (WMI)

This chapter covers the following recipes:

- Exploring WMI Architecture in Windows
- Exploring WMI Namespaces
- Exploring WMI Classes
- Obtaining WMI Class Instances
- Using WMI Methods
- Using WMI Events
- Implementing Permanent WMI Eventing

Introduction

Windows Management Instrumentation (WMI) is a Windows component you use to help manage Windows systems. WMI is Microsoft's proprietary implementation of the standards-based **Web-Based Enterprise Management (WBEM)**. WBEM is an open standard promulgated by the **Distributed Management Task Force (DMTF)** that aims to unify the management of distributed computing environments by using open standards-based internet technologies.

In addition to WMI, there are other implementations of WBEM, including OpenWBEM. You can read more about the DMTF and WBEM at `https://www.dmtf.org/about/faq/wbem_faq`, and check out OpenWBEM over at `http://openwbem.sourceforge.net/`. That said, WMI is most useful today on Windows.

Microsoft first introduced WMI as an add-on component for Windows NT 4. They later integrated WMI as an essential component of the Windows client from Windows XP onward, and Windows Server versions since Windows Server 2000. Subsequently, several feature teams inside the Windows group heavily used WMI. For example, the storage and networking stacks within Windows use WMI. Many of the cmdlets, such as `Get-NetAdapter`, are based directly on WMI.

The WBEM standards originally specified that the components of WMI communicate using HTTP. To improve performance, Microsoft instead used **Component Object Model** (**COM**) technology, which was a popular environment in the early days of Windows NTG.

Internally, WMI itself remains largely based on COM.

PowerShell 1.0 came with a set of cmdlets that you could use to access WMI. These were basic cmdlets that worked in all subsequent versions of Windows PowerShell. However, there is no support for these cmdlets directly in PowerShell 7. You can invoke older WMI cmdlet-based scripts on a machine using PowerShell remoting if you really need to.

With PowerShell V3, Microsoft did some significant work on WMI, resulting in a new module, `CimCmdlets`. There were several reasons behind the new module and the associated updates to some of the WMI internals to assist developers. In this chapter, you use the `CimCmdlets` module to access the features of WMI. You can read more about why the team built new cmdlets in a blog post at `https://devblogs.microsoft.com/powershell/introduction-to-cim-cmdlets/`. If you have scripts that use the older WMI cmdlets, consider upgrading them to use the later `CimCmdlets` module instead. These newer cmdlets are faster, which is always a nice benefit.

WMI architecture

WMI is a complex subject with a lot of parts. As an IT professional, it is useful to understand the WMI architecture within Windows before getting to know the **Common Information Model** (**CIM**) cmdlets. The runtime architecture of WMI is the same in Windows 10/11 and Windows Server 2022. The following diagram shows the conceptual view of the WMI architecture:

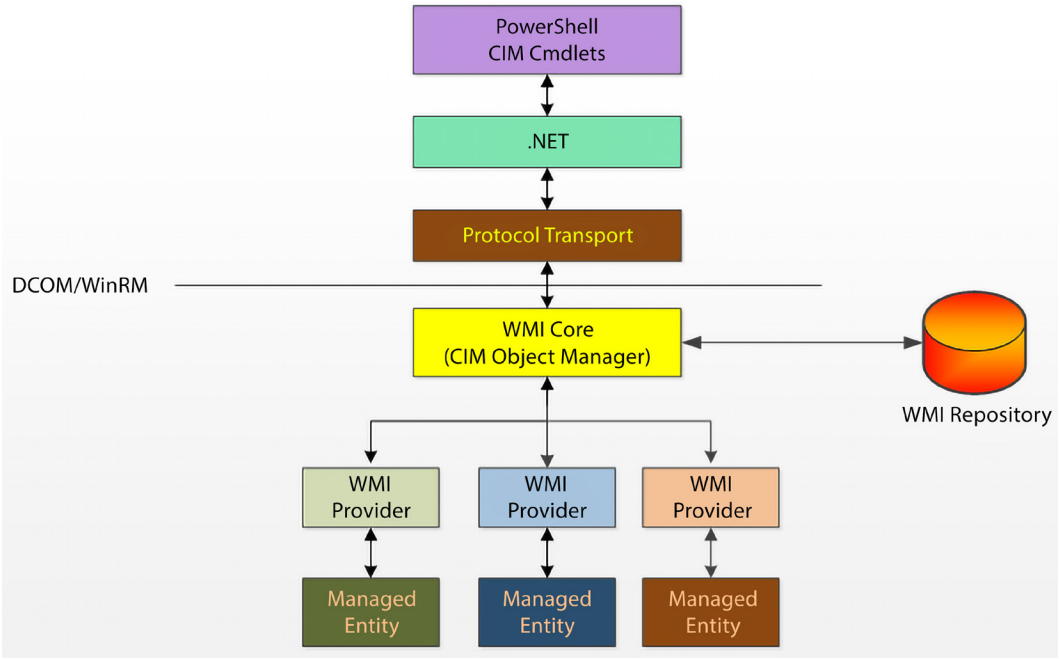

Figure 13.1: WMI architecture

As an IT pro, you use the cmdlets in the `CimCmdlets` module to access WMI. These cmdlets use .NET to communicate, via an underlying transport protocol (such as WinRM and Named Pipes), with the WMI core and the CIM Object Manager. The connection can be to either a local or remote host. The core components of WMI, particularly the **CIM Object Manager** (**CIMOM**), are all COM components you find on every Windows host.

The CIMOM stores information in the WMI repository, sometimes called the (CIM) or the CIM database. This database is, in effect, a subset of an ANSI-SQL database. The CIM cmdlets enable you to access the information within the database. The CIM database organizes the data into namespaces of classes. .NET also uses namespaces to collect .NET classes. However, .NET classes include the namespace name as part of the class name. For example, you can create a new email message using the .NET class `System.Net.Mail.Mailmessage`, where the namespace is `System.Net.Mail`. Thus, the full class name contains the namespace and class names. With WMI, namespace names and class names are separate – you supply them to the CIM cmdlets using different cmdlet parameters.

In the WMI database, WMI classes contain data instances that hold information about managed entities. For example, the WMI namespace `Root\CimV2` has the class `Win32_Share`. Each instance within this WMI class represents one of the **Server Message Block (SMB)** shares within your host. With PowerShell, you normally use the SMB cmdlets to manage SMB shares. There is often useful information in other WMI classes for which there is no cmdlet support.

Strictly speaking, inside PowerShell, a WMI object instance is an instance of a specialized .NET class with data returned from WMI. For this reason, you can treat WMI objects using the same mechanisms you employ with other .NET objects and the output from PowerShell cmdlets. When you retrieve instances of, for example, the `Win32_Share` class, .NET gets the instance information and returns it in a .NET wrapper object. You can then manipulate that share detail like any other object.

Many WMI classes have methods you can invoke that perform some operation on either a given WMI instance or statically based on the class. The `Win32_Share` class, for example, has a static `Create()` method that you can use to create a new share. Each instance of the `Win32_Share` class has a dynamic `Delete()` method, which deletes the SMB share.

An important architectural feature of WMI is the WMI provider. A WMI provider is an add on to WMI that implements WMI classes inside a given host. The `Win32` WMI provider, for example, implements hundreds of WMI classes, including `Win32_Share` and `Win32_Bios`. A provider also implements class methods and class events. For example, the `Win32` provider performs the `Delete()` method to delete an SMB share and the `Create()` method to create a new SMB share.

In production, you are more likely to manage SMB shares using the SMB cmdlets and less likely to use WMI directly. Since SMB shares should be very familiar, they make a great example to help you understand more about WMI, and this chapter's recipes use the class `Win32_Share`. That being said, using WMI directly to create SMB shares is a little faster than using `New-SMBShare`.

WMI and WMI providers both provide a rich eventing system. WMI and WMI provider classes can implement events to which you can subscribe. When the event occurs, the eventing system notifies you, and you can take some action to handle the event occurrence. For example, you can register for a WMI event when someone changes an AD group's membership. When this happens, WMI eventing allows you to take some actions, such as emailing a security administrator to inform them of the group's membership change. WMI also implements permanent WMI events. This feature allows you to configure WMI to trap and handle events with no active PowerShell session. Permanent events even survive a reboot of the host, which is extremely powerful in a lights-off environment.

There is much more detail about WMI than can fit in this chapter. For more information about WMI and how you can interact with it in more detail, consult Richard Siddaway's PowerShell and WMI book (© Manning, Aug 2012 - `https://www.manning.com/books/powershell-and-wmi`). Richard's book details WMI, but all the code samples use the older WMI cmdlets. You should be able to translate the samples to use the CIM cmdlets. A key value of the book is the discussion of WMI features and how they work. The basic functioning of WMI has not changed significantly since that book was published.

The systems used in the chapter

This chapter primarily uses the server `SRV1`, a domain-joined server in the `Reskit.Org` domain. You have used this server, and the domain, in previous chapters of this book. You also need access to the domain's two DCs (`DC1` and `DC2`) and another member server, `SRV2`.

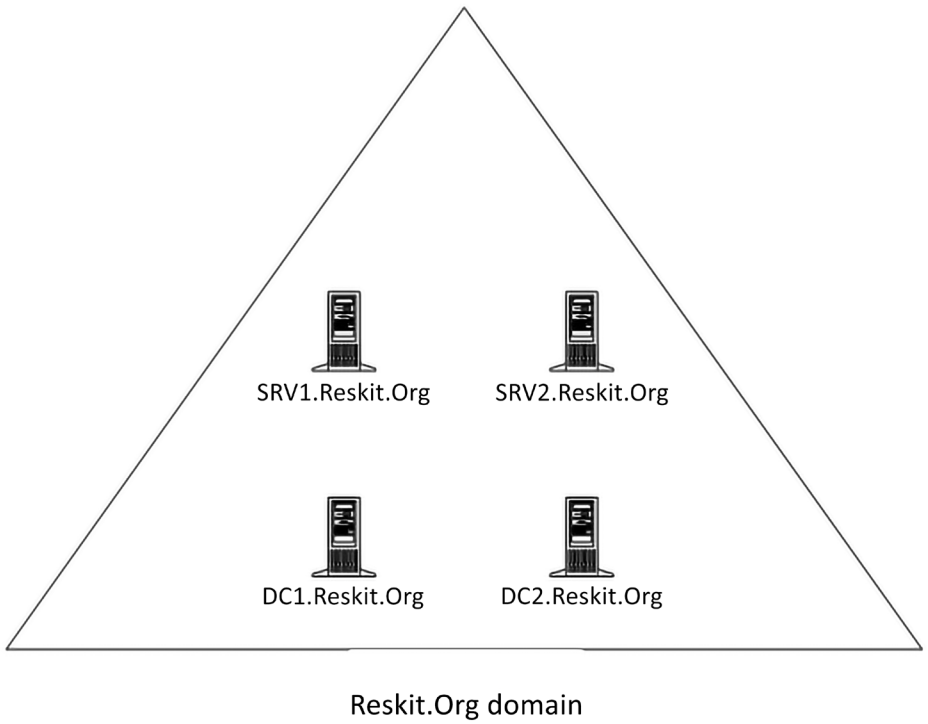

Figure 13.2: Host in use for this chapter

Exploring WMI Architecture in Windows

Windows installs WMI during the installation of the OS. The installation process puts most of the WMI components, including the repository, tools, and the WMI providers, into a folder `C:\Windows\System32\WBEM`.

Inside a running Windows host, WMI runs as a service, the winmgmt service (`WinMgmt.exe`). Windows runs this service inside a shared service process (`svchost.exe`). In the early versions of WMI in Windows, WMI loaded all the WMI providers into the winmgmt service. The failure of a single provider could cause the entire WMI service to fail. Later, with Windows XP and beyond, Microsoft improved WMI to load providers in a separate process, `Wmiprvse.exe`. WMI loads individual providers as needed.

In this recipe, you examine the contents of the WBEM folder, the WMI service, and the runtime components of WMI.

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS code on this host.

How to do it...

1. Viewing the `WBEM` folder

```
$WBEMFOLDER = "$Env:windir\system32\wbem"
Get-ChildItem -Path $WBEMFOLDER |
  Select-Object -First 20
```

2. Viewing the WMI repository folder

```
Get-ChildItem -Path $WBEMFOLDER\Repository
```

3. Viewing the WMI service details

```
Get-Service -Name Winmgmt  |
  Format-List -Property *
```

4. Getting process details

```
$Service = tasklist.exe /svc /fi "SERVICES eq winmgmt" |
             Select-Object -Last 1
$Process = [int] ($Service.Substring(30,4))
```

```
Get-Process -Id $|Process
```

5. Examining DLLs loaded by the WMI service process

```
Get-Process -Id $Process |
  Select-Object -ExpandProperty Modules |
    Where-Object ModuleName -match 'wmi' |
      Format-Table -Property FileName, Description, FileVersion
```

6. Discovering WMI providers

```
Get-ChildItem -Path $WBEMFOLDER\*.dll |
  Select-Object -ExpandProperty Versioninfo |
    Where-Object FileDescription -match 'prov' |
      Format-Table -Property Internalname,
                             FileDescription,
                             ProductVersion
```

7. Examining the `WmiPrvSE` process

```
Get-CimInstance -ClassName Win32_Bios | Out-Null
Get-Process -Name WmiPrvSE
```

8. Finding the WMI event log

```
$Log = Get-WinEvent -ListLog *wmi*
$Log
```

9. Looking at the event types in the WMI log

```
$Events = Get-WinEvent -LogName $Log.LogName
$Events | Group-Object -Property LevelDisplayName
```

10. Examining WMI event log entries

```
$Events |
  Select-Object -First 5 |
    Format-Table -Wrap
```

11. Viewing executable programs in the `WBEM` folder

```
$Files = Get-ChildItem -Path $WBEMFOLDER\*.exe
"{0,15}  {1,-40}" -f 'File Name','Description'
Foreach ($File in $Files){
```

```
 $Name = $File.Name
 $Desc = ($File |
          Select-Object -ExpandProperty VersionInfo).FileDescription
"{0,15}  {1,-40}" -f $Name,$Desc
}
```

12. Examining the `CimCmdlets` module

```
Get-Module -Name CimCmdlets |
  Select-Object -ExcludeProperty Exported*
    Format-List -Property *
```

13. Finding cmdlets in the `CimCmdlets` module

```
Get-Command -Module CimCmdlets
```

14. Examining the .NET type returned from `Get-CimInstance`

```
Get-CimInstance -ClassName Win32_Share | Get-Member
```

How it works...

The WMI service and related files are in the Windows installation folder's `System32\WBEM` folder. In *step 1*, you view part of the contents of that folder, with output like this:

```
PS C:\Foo> # 1. Viewing the WBEM folder
PS C:\Foo> $WBEMFOLDER = "$Env:windir\system32\wbem"
PS C:\Foo> Get-ChildItem -Path $WBEMFOLDER |
             Select-Object -First 20

    Directory: C:\Windows\System32\wbem

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        23/09/2022     16:35                AutoRecover
d-----        08/05/2021     10:41                en
d-----        22/09/2022     16:24                en-US
d-----        08/05/2021     09:20                Logs
d-----        22/09/2022     13:49                MOF
d-----        26/10/2022     17:23                Performance
d-----        24/09/2022     16:09                Repository
d-----        08/05/2021     09:20                tmf
d-----        08/05/2021     09:20                xml
-a----        08/05/2021     09:14           2852 aeinv.mof
-a----        08/05/2021     10:41          17510 AgentWmi.mof
-a----        08/05/2021     09:15            693 AgentWmiUninstall.mof
-a----        08/05/2021     09:14            852 appbackgroundtask_uninstall.mof
-a----        08/05/2021     09:14          65536 appbackgroundtask.dll
-a----        08/05/2021     09:14           2902 appbackgroundtask.mof
-a----        08/05/2021     09:15           1112 AttestationWmiProvider_Uninstall.mof
-a----        08/05/2021     09:15           3376 AttestationWmiProvider.mof
-a----        08/05/2021     09:14           1724 AuditRsop.mof
-a----        08/05/2021     09:14           1092 authfwcfg.mof
-a----        08/05/2021     09:14          12120 bcd.mof
```

Figure 13.3: Examining the WBEM folder

WMI stores the CIM repository in a separate folder. In *step 2*, you examine the files that make up the database, with output like this:

```
PS C:\Foo> # 2. Viewing the WMI repository folder
PS C:\Foo> Get-ChildItem -Path $WBEMFOLDER\Repository

    Directory: C:\Windows\System32\wbem\Repository

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        05/11/2022     22:20        5750784 INDEX.BTR
-a----        05/11/2022     19:32         105640 MAPPING1.MAP
-a----        05/11/2022     22:20         105640 MAPPING2.MAP
-a----        05/11/2022     06:52         105640 MAPPING3.MAP
-a----        05/11/2022     22:20       31899648 OBJECTS.DATA
```

Figure 13.4: Examining the files making up the CIM repository

In *step 3*, you use the `Get-Service` cmdlet to examine the WMI service, with a console output that looks like this:

```
PS C:\Foo> # 3. Viewing the WMI service details
PS C:\Foo> Get-Service -Name Winmgmt  |
             Format-List -Property *

UserName            : localSystem
Description         : Provides a common interface and object model to access management
                      information about operating system, devices, applications and
                      services. If this service is stopped, most Windows-based software will
                      not function properly. If this service is disabled, any services that
                      explicitly depend on it will fail to start.
DelayedAutoStart    : False
BinaryPathName      : C:\WINDOWS\system32\svchost.exe -k netsvcs -p
StartupType         : Automatic
Name                : Winmgmt
RequiredServices    : {RPCSS}
CanPauseAndContinue : True
CanShutdown         : True
CanStop             : True
DisplayName         : Windows Management Instrumentation
DependentServices   : {UALSVC, HgClientService}
MachineName         : .
ServiceName         : Winmgmt
ServicesDependedOn  : {RPCSS}
StartType           : Automatic
ServiceHandle       : Microsoft.Win32.SafeHandles.SafeServiceHandle
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
Site                :
Container           :
```

Figure 13.5: Viewing the WMI service

In *step 4*, you examine the Windows process that runs the WMI service, with output like this:

```
PS C:\Foo> # 4. Getting process details
PS C:\Foo> $Service = tasklist.exe /svc /fi "SERVICES eq winmgmt" |
                      Select-Object -Last 1
PS C:\Foo> $Process = [int] ($Service.Substring(30,4))
PS C:\Foo> Get-Process -Id $Process

 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
     17    16.25      36.81      90.09    3392   0 svchost
```

Figure 13.6: Viewing the WMI service

In *step 5*, you look at the DLLs loaded by the WMI service process with the following output:

```
PS C:\Foo> # 5. Examining DLLs loaded by the WMI service process
PS C:\Foo> Get-Process -Id $Process |
             Select-Object -ExpandProperty Modules |
               Where-Object ModuleName -match 'wmi' |
                 Format-Table -Property FileName, Description, FileVersion

FileName                                Description    FileVersion
--------                                -----------    -----------
c:\windows\system32\wbem\wmisvc.dll     WMI            10.0.20348.1 (WinBuild.160101.0800)
C:\WINDOWS\SYSTEM32\WMICLNT.dll         WMI Client API 10.0.20348.1 (WinBuild.160101.0800)
C:\WINDOWS\system32\wbem\wmiutils.dll   WMI            10.0.20348.1 (WinBuild.160101.0800)
C:\WINDOWS\system32\wbem\wmiprvsd.dll   WMI            10.0.20348.1 (WinBuild.160101.0800)
```

Figure 13.7: Viewing the DLLs loaded by the WMI service process

Each WMI provider is a DLL that the WMI service can use. In *step 6*, you look at the WMI providers on SRV1, with output like this:

```
DhcpServerPsProvider.dll      DHCP WMIv2 Provider                              10.0.20348.1
DMWmiBridgeProv.dll           DM WMI Bridge Provider                           10.0.20348.1
DMWmiBridgeProv1.dll          DM WMI Bridge Provider                           10.0.20348.1
DnsClientPsProvider.dll       DNS Client WMIv2 Provider                        10.0.20348.1
DnsServerPsProvider.dll       DNS WMIv2 Provider                               10.0.20348.1
dsprov.dll                    WMI DS Provider                                  10.0.20348.1
"EventTracingManagement.dll"  WMI Provider for ETW                             10.0.20348.1
ipmiprr.dll                   IPMI Provider Resource                           10.0.20348.1
IPMIPRV.dll                   WMI IPMI PROVIDER                                10.0.20348.1
MDMAppProv.dll                MDM Application Provider                         10.0.20348.1
MDMSettingsProv.dll           MDM Settings Provider                            10.0.20348.1
mgmtprovider.dll              Server Manager Managment Provider                10.0.20348.1
mistreamprov.dll              SIL Composable Streams Provider                  10.0.20348.1
DtcCIMProvider.dll            DTC WMIv2 Provider                               10.0.20348.1
msiprov.dll                   WMI MSI Provider                                 10.0.20348.1
MspsProv.exe                  Msps Prov                                        10.0.20348.1
mttmprov.dll                  Mangement Tools Task Manager CIM Provider        10.0.20348.1
NCObjAPI                      Non-COM WMI Event Provision APIs                 10.0.20348.1
ndisimplatwmi.DLL             NDIS IM Platform WMI Provider                    10.0.20348.1
NetAdapter.dll                Network Adapter WMI Provider                     10.0.20348.1
netdacim.dll                  Microsoft Direct Access WMI Provider             10.0.20348.1
NetEventPacketCapture.dll     NetEvent Packet Capture Provider                 10.0.20348.1
NetNat.dll                    Windows NAT WMI Provider                         10.0.20348.1
netnccim.dll                  DA Network Connectivity WMI Provider             10.0.20348.1
NetPeerDistCim.dll            BranchCache WMI Provider                         10.0.20348.1
netswitchteamcim.DLL          VM Switch Teaming WMI Provider                   10.0.20348.1
NetTCPIP.dll                  TCPIP WMI Provider                               10.0.20348.1
nlmcim.dll                    Network Connection Profiles WMI Provider         10.0.20348.1
ntevt.dll                     WMI Event Log Provider                           10.0.20348.1
PLATID.DLL                    WMIv2 Provider to Retrieve Platform Identifiers  10.0.20348.1
PrintManagementProvider.DLL   Print WMI Provider                              10.0.20348.1
RacWmiProv.dll                Reliability Metrics WMI Provider                 10.0.20348.1
RAMgmtPSProvider.dll          RAMgmtPSProvider                                 10.0.20348.1
regprov.dll                   Registry Provider                                10.0.20348.1
schedprov.dll                 Task Scheduler WMIv2 Provider                    10.0.20348.1
SDNDiagnosticsProvider.dll    SDNDiagnosticsProvider                           10.0.20348.1
servercompprov.dll            Windows Server Feature WMI Provider              10.0.20348.1
sllprovider.dll               Server License Logging CIM Provider              10.0.20348.1
vdswmi.dll                    WMI Provider for VDS                             10.0.20348.1
viewprov.dll                  WMI View Provider                                10.0.20348.1
VpnClientPsProvider.dll       VPN Client WMIv2 Provider                        10.0.20348.1
VSSPROV.DLL                   WMI Provider for VSS                             10.0.20348.1
WdacWmiProv.dll               WDAC WMI Providers                               10.0.20348.1
Win32_EncryptableVolume.DLL   Win32_Encryptable Volume Provider               10.0.20348.143
Win32_Tpm.DLL                 TPM WMI Provider                                 10.0.20348.1
WMIPCIMA.dll                  WMI Win32Ex Provider                             10.0.20348.1
wmipdfs.dll                   WMI DFS Provider                                 10.0.20348.1
WMIPDskQ.dll                  WMI Provider for Disk Quota Information          10.0.20348.1
WbemPerfClass.dll             WbemPerf V2 Class Provider                       10.0.20348.1
WbemPerfInst.dll              WbemPerf V2 Instance Provider                    10.0.20348.1
wmipicmp.dll                  WMI ICMP Echo Provider                           10.0.20348.1
WMIPIpRt.dll                  WBEM Provider for IP4 Routes                     10.0.20348.1
wmipjobj.dll                  WMI Windows Job Object Provider                  10.0.20348.1
WMIPSess.dll                  WMI Provider for Sessions and Connections        10.0.20348.1
```

Figure 13.8: Viewing WMI provider DLLs

In *step 7*, you examine the `WmiPrvSE` process, with output like this:

```
PS C:\Foo> # 7. Examining the WmiPrvSE process
PS C:\Foo> Get-CimInstance -ClassName Win32_Bios | Out-Null
PS C:\Foo> Get-Process -Name WmiPrvSE

 NPM(k)     PM(M)       WS(M)     CPU(s)       Id  SI ProcessName
 ------     -----       -----     ------       --  -- -----------
     11      2.75        9.10       0.03    10012   0 WmiPrvSE
```

Figure 13.9: Viewing the WmiPrvSE process

Like other Windows services, WMI writes details of events to an event log, which can help troubleshoot WMI issues. In *step 8*, you look for the WMI event log, with output like this:

```
PS C:\Foo> # 8. Finding the WMI event log
PS C:\Foo> $Log = Get-WinEvent -ListLog *wmi*
PS C:\Foo> $Log

LogMode   MaximumSizeInBytes RecordCount LogName
-------   ------------------- ----------- -------
Circular              1052672        1457 Microsoft-Windows-WMI-Activity/Operational
```

Figure 13.10: Viewing the WMI event log

In *step 9*, you get the events from the log to view the different log levels, with output like this:

```
PS C:\Foo> # 9. Looking at the Event Types in the WMI log
PS C:\Foo> $Events = Get-WinEvent -LogName $Log.LogName
PS C:\Foo> $Events | Group-Object -Property LevelDisplayName

Count Name           Group
----- ----           -----
  728 Error          {System.Diagnostics.Eventing.Reader.EventLogRecord, System.Diagnostics.Eventing.Reader.EventLogRecor...
  729 Information    {System.Diagnostics.Eventing.Reader.EventLogRecord, System.Diagnostics.Eventing.Reader.EventLogRecor...
```

Figure 13.11: Discovering WMI event types

In *step 10*, you view the first five WMI event log entries on `SRV1`. The output looks like this:

```
PS C:\Foo> # 10. Examining WMI event log entries
PS C:\Foo> $Events |
             Select-Object -First 5 |
               Format-Table -Wrap

   ProviderName: Microsoft-Windows-WMI-Activity

TimeCreated           Id LevelDisplayName Message
-----------           -- ---------------- -------
06/11/2022 20:51:39 5857 Information      CIMWin32 provider started with result code 0x0. HostProcess = wmiprvse.exe; ProcessID
                                          = 10012; ProviderPath = %systemroot%\system32\wbem\cimwin32.dll
06/11/2022 20:50:53 5857 Information      WMIProv provider started with result code 0x0. HostProcess = wmiprvse.exe; ProcessID =
                                          8304; ProviderPath = %systemroot%\system32\wbem\wmiprov.dll
06/11/2022 20:50:52 5860 Information      Namespace = ROOT\CIMV2; NotificationQuery = SELECT * FROM Win32_ProcessStartTrace
                                          WHERE ProcessName = 'wsmprovhost.exe'; UserName = NT AUTHORITY\SYSTEM; ClientProcessID
                                          = 6780, ClientMachine = SRV1; PossibleCause = Temporary
06/11/2022 20:50:52 5857 Information      WMI Kernel Trace Event Provider provider started with result code 0x0. HostProcess =
                                          wmiprvse.exe; ProcessID = 3392; ProviderPath = C:\Windows\System32\wbem\krnlprov.dll
06/11/2022 20:50:51 5857 Information      WmiPerfClass provider started with result code 0x0. HostProcess = wmiprvse.exe;
                                          ProcessID = 8304; ProviderPath = C:\Windows\System32\wbem\WmiPerfClass.dll
```

Figure 13.12: Viewing WMI event log entries

In *step 11*, you view the executable programs in the `WBEM` folder, with output like this:

```
PS C:\Foo> # 11. Viewing executable programs in WBEM folder
PS C:\Foo> $Files = Get-ChildItem -Path $WBEMFOLDER\*.exe
PS C:\Foo> "{0,15}  {1,-40}" -f 'File Name','Description'
PS C:\Foo> Foreach ($File in $Files){
             $Name = $File.Name
             $Desc = ($File |
                      Select-Object -ExpandProperty VersionInfo).FileDescription
           "{0,15}  {1,-40}" -f $Name,$Desc
           }

      File Name  Description
     mofcomp.exe  The Managed Object Format (MOF) Compiler
     scrcons.exe  WMI Standard Event Consumer - scripting
   unsecapp.exe  Sink to receive asynchronous callbacks for WMI client application
   wbemtest.exe  WMI Test Tool
    WinMgmt.exe  WMI Service Control Utility
    WMIADAP.exe  WMI Reverse Performance Adapter Maintenance Utility
   WmiApSrv.exe  WMI Performance Reverse Adapter
       WMIC.exe  WMI Commandline Utility
   WmiPrvSE.exe  WMI Provider Host
```

Figure 13.13 Viewing the executable programs in the WBEM folder

With PowerShell 7 (and, optionally, with Windows PowerShell), you access WMI's functionality using the cmdlets in the `CimCmdlets` module. The Windows installation program installed a version of this module when you installed the host OS, and the PowerShell 7 installation process adds an updated version of this module.

In *step 12*, you examine the properties of this module, with output like this:

```
PS C:\Foo> # 12. Examining the CimCmdlets module
PS C:\Foo> Get-Module -Name CimCmdlets |
             Select-Object -ExcludeProperty Exported*

LogPipelineExecutionDetails : False
Name                        : CimCmdlets
Path                        : C:\Program Files\PowerShell\7\Microsoft.Management.Infrastructure.CimCmdlets.dll
ImplementingAssembly        : Microsoft.Management.Infrastructure.CimCmdlets, Version=7.2.6.500, Culture=neutral,
                              PublicKeyToken=31bf3856ad364e35
Definition                  :
Description                 :
Guid                        : fb6cc51d-c096-4b38-b78d-0fed6277096a
HelpInfoUri                 : https://aka.ms/powershell72-help
ModuleBase                  : C:\program files\powershell\7\Modules\CimCmdlets
PrivateData                 :
ExperimentalFeatures        : {}
Tags                        : {}
ProjectUri                  :
IconUri                     :
LicenseUri                  :
ReleaseNotes                :
RepositorySourceLocation    :
Version                     : 7.0.0.0
ModuleType                  : Binary
Author                      : PowerShell
AccessMode                  : ReadWrite
ClrVersion                  :
CompanyName                 : Microsoft Corporation
Copyright                   : Copyright (c) Microsoft Corporation.
DotNetFrameworkVersion      :
Prefix                      :
FileList                    : {}
CompatiblePSEditions        : {Core}
ModuleList                  : {}
NestedModules               : {}
PowerShellHostName          :
PowerShellHostVersion       :
PowerShellVersion           : 3.0
ProcessorArchitecture       : None
Scripts                     : {}
RequiredAssemblies          : {Microsoft.Management.Infrastructure.CimCmdlets.dll, Microsoft.Management.Infrastructure.Dll}
RequiredModules             : {}
RootModule                  : Microsoft.Management.Infrastructure.CimCmdlets
SessionState                : System.Management.Automation.SessionState
OnRemove                    :
```

Figure 13.14: Viewing the CimCmdlets module details

In *step 13*, you use the `Get-Command` cmdlet to discover the cmdlets within the `CimCmdlets` module, which looks like this:

```
PS C:\Foo> # 13. Finding cmdlets in the CimCmdlets module
PS C:\Foo> Get-Command -Module CimCmdlets

CommandType Name                           Version Source
----------- ----                           ------- ------
Cmdlet      Get-CimAssociatedInstance      7.0.0.0 CimCmdlets
Cmdlet      Get-CimClass                   7.0.0.0 CimCmdlets
Cmdlet      Get-CimInstance                7.0.0.0 CimCmdlets
Cmdlet      Get-CimSession                 7.0.0.0 CimCmdlets
Cmdlet      Invoke-CimMethod               7.0.0.0 CimCmdlets
Cmdlet      New-CimInstance                7.0.0.0 CimCmdlets
Cmdlet      New-CimSession                 7.0.0.0 CimCmdlets
Cmdlet      New-CimSessionOption           7.0.0.0 CimCmdlets
Cmdlet      Register-CimIndicationEvent    7.0.0.0 CimCmdlets
Cmdlet      Remove-CimInstance             7.0.0.0 CimCmdlets
Cmdlet      Remove-CimSession              7.0.0.0 CimCmdlets
Cmdlet      Set-CimInstance                7.0.0.0 CimCmdlets
```

Figure 13.15: Viewing the cmdlets in the `CimCmdlets` *module*

In *step 14*, you examine the properties of an object returned from WMI after using the `Get-CimInstance` command. The output from this step looks like this:

```
PS C:\Foo> # 14. Examining the .NET type returned from Get-CimInstance
PS C:\Foo> Get-CimInstance -ClassName Win32_Share | Get-Member

   TypeName: Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_Share

Name                       MemberType  Definition
----                       ----------  ----------
Dispose                    Method      void Dispose(), void IDisposable.Dispose()
Equals                     Method      bool Equals(System.Object obj)
GetCimSessionComputerName  Method      string GetCimSessionComputerName()
GetCimSessionInstanceId    Method      guid GetCimSessionInstanceId()
GetHashCode                Method      int GetHashCode()
GetType                    Method      type GetType()
ToString                   Method      string ToString()
AccessMask                 Property    uint AccessMask {get;}
AllowMaximum               Property    bool AllowMaximum {get;}
Caption                    Property    string Caption {get;}
Description                Property    string Description {get;}
InstallDate                Property    CimInstance#DateTime InstallDate {get;}
MaximumAllowed             Property    uint MaximumAllowed {get;}
Name                       Property    string Name {get;}
Path                       Property    string Path {get;}
PSComputerName             Property    string PSComputerName {get;}
Status                     Property    string Status {get;}
Type                       Property    uint Type {get;}
PSStatus                   PropertySet PSStatus {Status, Type, Name}
```

Figure 13.16: Examining the output from Get-CimInstance

There's more...

In *step 1*, you viewed the first 20 files/folders in the `WBEM` folder. There are a lot more files than you see in the figure. These include the DLL files for the WMI providers available on your system.

In *step 7*, you view the `WmiPrvSE` process. This process hosts WMI providers. Depending on the actions WMI is currently doing, you may see zero, one, or more occurrences of this process on your hosts.

In *step 9* and *step 10*, you discover and examine the WMI system event log. On `SRV1`, you can see both `Error` and `Information` event log entries. As you can see in *step 10*, the information entries mostly indicate that WMI has loaded and invoked a particular provider. In most cases, the error event messages you see are transient or benign.

In *step 14*, you looked at the data returned by `Get-CimInstance`. As you can see from the output, the cmdlet returns the data obtained from the WMI class. This data is wrapped in a .NET object and has a class of `Microsoft.Management.Infrastructure.CimInstance`, with a suffix indicating the path to the WMI class, in this case, the `Win32_Share` class in the `ROOT/CIMV2` WMI namespace.

As you can see from the output, the returned object and its contents differ from that produced by the `Get-WMIObject` cmdlet. This could be an issue if you upgrade Windows PowerShell scripts that previously used the older WMI cmdlets.

Exploring WMI Namespaces

The PowerShell CIM cmdlets enable you to retrieve, update, and remove information from the CIM database and subscribe to and handle WMI events. The CIM database organizes its data into sets of classes within a hierarchical set of namespaces. A namespace is, in effect, a container holding WMI classes.

The name of the root WMI namespace is `ROOT`, although WMI is not overly consistent about capitalization, as you may notice. A namespace can contain classes as well as additional child namespaces. For example, the root namespace has a child namespace, `CIMV2`, which you refer to as `ROOT\CIMV2`. This namespace also has child namespaces.

Every namespace in the CIM database, including `ROOT`, has a special system class called `__NAMESPACE`. This class contains the names of child namespaces within the current namespaces. Thus, in the namespace `ROOT`, the `__NAMESPACE` class includes an instance for the `CIMV2` child namespace. Since this class exists inside every namespace, it is straightforward to discover all the namespaces on your system.

There are many namespaces and classes within WMI on any given system. The specific namespaces and classes depend on what applications and Windows features you run on a host. Additionally, not all the namespaces or classes are useful to the IT pro. Other classes or namespaces may contain classes useful for IT professionals, while most are typically only useful for developers implementing WMI components or WMI providers. The `Win32` WMI provider implements the `ROOT\CimV2` namespace. This namespace contains classes of interest to IT pros.

Another less commonly used namespace is `ROOT\directory\LDAP`, which contains classes related to the Active Directory. While you perform most of the AD management using the AD cmdlets, there are features of this namespace, specifically eventing, that are not available with the AD cmdlets and that you may find useful. And these classes can be utilized even if you do not have the AD **Remote Server Administration Tools (RSAT)** tools loaded.

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Viewing WMI classes in the root namespace

```
Get-CimClass -Namespace 'ROOT' |
  Select-Object -First 10
```

2. Viewing the `__NAMESPACE` class in `ROOT`

```
Get-CimInstance -Namespace 'ROOT' -ClassName __NAMESPACE |
  Sort-Object -Property Name
```

3. Getting and counting classes in `ROOT\CIMV2`

```
$Classes = Get-CimClass -Namespace 'ROOT\CIMV2'
"There are $($Classes.Count) classes in ROOT\CIMV2"
```

4. Discovering all the namespaces on SRV1

```
$EAHT = @{ErrorAction = 'SilentlyContinue'}
Function Get-WMINamespaceEnum {
  [CmdletBinding()]
  Param($NS)
  Write-Output $NS
  Get-CimInstance "__Namespace" -Namespace $NS @EAHT |
  ForEach-Object { Get-WMINamespaceEnum "$ns\$($_.name)"   }
}  # End of function
$Namespaces = Get-WMINamespaceEnum 'ROOT' |
  Sort-Object
"There are $($Namespaces.Count) WMI namespaces on SRV1"
```

5. Viewing first 25 namespaces on SRV1

```
$Namespaces |
  Select-Object -First 25
```

6. Creating a script block to count namespaces and classes

```
$ScriptBlock = {
 Function Get-WMINamespaceEnum {
   [CmdletBinding()]
   Param(
     $NameSpace
    )
   Write-Output $NameSpace
   $EAHT = @{ErrorAction = 'SilentlyContinue'}
   Get-CimInstance "__Namespace" -Namespace $NameSpace @EAHT |
     ForEach-Object {
       Get-WMINamespaceEnum "$NameSpace\$($_.Name)"
     }
   }  # End of function
   $Namespaces = Get-WMINamespaceEnum 'ROOT' | Sort-Object
   $WMIClasses = @()
```

```
    Foreach ($WMINameSpace in $Namespaces) {
    $WMIClasses += Get-CimClass -Namespace $WMINameSpace
   }
  "There are $($Namespaces.Count) WMI namespaces on $(hostname)"
  "There are $($WMIClasses.Count) classes on $(hostname)"
 }
```

7. Running the script block locally on `SRV1`

```
Invoke-Command -ComputerName SRV1 -ScriptBlock $ScriptBlock
```

8. Running the script block on `SRV2`

```
Invoke-Command -ComputerName SRV2 -ScriptBlock $ScriptBlock
```

9. Running the script block on `DC1`

```
Invoke-Command -ComputerName DC1 -ScriptBlock $ScriptBlock
```

How it works...

In *step 1*, you view the WMI classes in the WMI root namespace on `SRV1`, with output like this:

```
PS C:\Foo> # 1. Viewing WMI classes in the root namespace
PS C:\Foo> Get-CimClass -Namespace 'ROOT' |
             Select-Object -First 10

   NameSpace: ROOT

CimClassName                        CimClassMethods  CimClassProperties
------------                        ---------------  ------------------
__SystemClass                       {}               {}
__thisNAMESPACE                     {}               {SECURITY_DESCRIPTOR}
__CacheControl                      {}               {}
__EventConsumerProviderCacheControl {}               {ClearAfter}
__EventProviderCacheControl         {}               {ClearAfter}
__EventSinkCacheControl             {}               {ClearAfter}
__ObjectProviderCacheControl        {}               {ClearAfter}
__PropertyProviderCacheControl      {}               {ClearAfter}
__NAMESPACE                         {}               {Name}
__ArbitratorConfiguration           {}               {OutstandingTasksPerUser, OutstandingTasksTotal…
```

Figure 13.17: WMI classes in the ROOT namespace

In *step 2*, you examine the instances of the __NAMESPACE class in the ROOT WMI namespace. The output looks like this:

```
PS C:\Foo> # 2. Viewing the __NAMESPACE class in ROOT
PS C:\Foo> Get-CimInstance -Namespace 'ROOT' -ClassName __NAMESPACE |
             Sort-Object -Property Name

Name             PSComputerName
----             --------------
AccessLogging
Appv
CIMV2
Cli
DEFAULT
directory
Hardware
Interop
InventoryLogging
Microsoft
msdtc
MSPS
PEH
Policy
RSOP
SECURITY
ServiceModel
StandardCimv2
subscription
WMI
```

Figure 13.18: Examining the __NAMESPACE class in the ROOT namespace

With *step 3*, you get and then count the classes in the ROOT\CIMV2 namespace, with output like this:

```
PS C:\Foo> # 3. Getting and counting classes in ROOT\CIMV2
PS C:\Foo> $Classes = Get-CimClass -Namespace 'ROOT\CIMV2'
PS C:\Foo> "There are $($Classes.Count) classes in ROOT\CIMV2"
There are 1196 classes in ROOT\CIMV2
```

Figure 13.19: Getting and counting the classes in Root\CIMV2

In *step 4*, you define and then use a function to discover all the namespaces in WMI on this host, sorted alphabetically, and then display a count of the namespaces found. The output of this step looks like this:

```
PS C:\Foo> # 4. Discovering all the namespaces on SRV1
PS C:\Foo> $EAHT = @{ErrorAction = 'SilentlyContinue'}
PS C:\Foo> Function Get-WMINamespaceEnum {
             [CmdletBinding()]
             Param($NS)
             Write-Output $NS
             Get-CimInstance "__Namespace" -Namespace $NS @EAHT |
             ForEach-Object { Get-WMINamespaceEnum "$ns\$($_.name)"   }
           }  # End of function
PS C:\Foo> $Namespaces = Get-WMINamespaceEnum 'ROOT' |
             Sort-Object
PS C:\Foo> "There are $($Namespaces.Count) WMI namespaces on SRV1"

There are 126 WMI namespaces on SRV1
```

Figure 13.20: Discovering all the WMI namespaces on SRV1

In *step 5*, you view the first 25 namespace names, with output like this:

```
PS C:\Foo> # 5. Viewing first 25 namespaces on SRV1
PS C:\Foo> $Namespaces |
             Select-Object -First 25
ROOT
ROOT\AccessLogging
ROOT\Appv
ROOT\CIMV2
ROOT\CIMV2\mdm
ROOT\CIMV2\mdm\dmmap
ROOT\CIMV2\mdm\MS_409
ROOT\CIMV2\ms_409
ROOT\CIMV2\power
ROOT\CIMV2\power\ms_409
ROOT\CIMV2\Security
ROOT\CIMV2\Security\MicrosoftTpm
ROOT\CIMV2\Security\MicrosoftVolumeEncryption
ROOT\CIMV2\TerminalServices
ROOT\CIMV2\TerminalServices\ms_409
ROOT\Cli
ROOT\Cli\MS_409
ROOT\DEFAULT
ROOT\DEFAULT\ms_409
ROOT\directory
ROOT\directory\LDAP
ROOT\directory\LDAP\ms_409
ROOT\Hardware
ROOT\Hardware\ms_409
ROOT\Interop
```

Figure 13.21: Listing the first 25 namespaces in WMI on SRV1

In *step 6*, you create a script that counts WMI namespaces and classes for a given host. This step generates no console output. In *step 7*, you run this function against SRV1, with output like this:

```
PS C:\Foo> # 7. Running the script block locally on SRV1
PS C:\Foo> Invoke-Command -ComputerName SRV1 -ScriptBlock $ScriptBlock
There are 126 WMI namespaces on SRV1
There are 17213 classes on SRV1
```

Figure 13.22: Counting the namespaces and classes in WMI on SRV1

In *step 8*, you run the script block on SRV2, with output like this:

```
PS C:\Foo> # 8. Running the script block on SRV2
PS C:\Foo> Invoke-Command -ComputerName SRV2 -ScriptBlock $ScriptBlock
There are 102 WMI namespaces on SRV2
There are 14150 classes on SRV2
```

Figure 13.23: Counting the namespaces and classes in WMI on SRV2

In the final step, *step 9*, you run the script block on a domain controller, DC1. The output of this step is as follows:

```
PS C:\Foo> # 9. Running the script block on DC1
PS C:\Foo> Invoke-Command -ComputerName DC1 -ScriptBlock $$ScriptBlock
There are 114 WMI namespaces on DC1
There are 16492 classes on DC1
```

Figure 13.24: Counting the namespaces and classes in WMI on DC1

There's more...

In *step 2*, you determine the child namespaces of the root namespaces. Each instance contains a string with the child's namespace name. The first entry is `AccessLogging`. Therefore the full namespace name of this child namespace is `ROOT\AccessLogging`.

In *step 3*, you count the classes in the `ROOT\CIMV2`. As mentioned, not all of these classes are useful to an IT pro, although many are. You can use your search engine to find classes that might be useful.

In *step 4*, you define a recursive function. When you call this function, specifying the `ROOT` namespace, the function retrieves the child namespace names from the `__NAMESPACE` class in the `ROOT` namespace. Then, for each child's namespace of the root, the function calls itself with a child namespace name. Eventually, this function returns the names of every namespace in WMI. You then sort this alphabetically by namespace name. Note that you can sort the output of `GET-WMINamespaceEnum` without specifying a property – you are sorting on the contents of the strings returned from the function.

In *step 5*, you view some of the namespaces in WMI on `SRV1`. Two important namespaces are the `ROOT\CIMV2` and the `ROOT\directory\LDAP`. The former contains classes provided by the `Win32` WMI provider. These classes include details about the software and hardware on your system, including the bios, the OS, files, and much more.

Step 7, *step 8*, and *step 9* run the function (defined in *step 6*) remotely. These steps count and display a count of the namespaces and classes on all three systems. For this reason, you should expect that the number of classes and namespaces differs.

Exploring WMI Classes

A WMI class defines a WMI-managed object such as a file share, a disk file, etc. All WMI classes live within a namespace. WMI classes, like .NET classes, contain members that include properties, methods, and events. An example class is `Win32_Share`, which you find in the `root\CIMV2` namespace. This class defines an SMB share on a Windows host. Within WMI, the `Win32` WMI provider implements this class (along with multiple other OS and host-related classes).

As mentioned, you typically use the SMB cmdlets to manage SMB shares (as discussed in *Chapter 8, Managing Shared Data*, including the *Creating and securing SMB shares* recipe). Likewise, you carry out most AD management activities using AD cmdlets rather than accessing the information via WMI. Nevertheless, you can do things with WMI, such as event handling, that can be very useful to the IT professional.

A WMI class contains one or more properties that are attributes of an occurrence of a WMI class. Classes can also include methods that act on a WMI occurrence. For example, the `Win32_Share` class contains a `Name` property that holds that share's name. The `Win32_Share` class also has a `Create()` method to create a new SMB share and a `Delete()` method to remove a specific SMB share. A WMI method can be either dynamic (or instance-based) or static (class-related). The `Win32_Share's Delete()` method is a dynamic method you use to delete a particular SMB share. The `Create()` method is a static method that the class can perform to create a new SMB share. You can learn more about WMI methods in the *Using WMI Methods* recipe later in this chapter.

In this recipe, you use the CIM cmdlets to discover information about classes and what a class can contain. You first examine a class within the default `root\CIMV2` namespace. You also look at a class in a non-default namespace and discover the objects contained in the class.

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Viewing `Win32_Share` class

```
Get-CimClass -ClassName Win32_Share
```

2. Viewing `Win32_Share` class properties:

```
Get-CimClass -ClassName Win32_Share |
  Select-Object -ExpandProperty CimClassProperties |
    Sort-Object -Property Name |
      Format-Table -Property Name, CimType
```

3. Getting methods of `Win32_Share` class

```
Get-CimClass -ClassName Win32_Share |
  Select-Object -ExpandProperty CimClassMethods
```

4. Getting classes in a non-default namespace

```
Get-CimClass -Namespace root\directory\LDAP |
  Where-Object CimClassName -match '^ds_group'
```

5. Viewing the instances of the `ds_group` class

```
Get-CimInstance -Namespace root\directory\LDAP -Classname 'DS_Group'
|
  Select-Object -First 10 |
    Format-Table -Property DS_name, DS_Member
```

How it works...

In *step 1*, you view a specific class, the `Win32_Share` class, with output like this:

```
PS C:\Foo> # 1. Viewing Win32_Share class
PS C:\Foo> Get-CimClass -ClassName Win32_Share

   NameSpace: ROOT/cimv2

CimClassName CimClassMethods          CimClassProperties
------------ ----------------         ------------------
Win32_Share  {Create, SetShareInfo,   {Caption, Description, InstallDate, Name,
              GetAccessMask, Delete}   Status, AccessMask, AllowMaximum, MaximumAllowed,
                                       Path, Type}
```

Figure 13.25: Viewing the Win32_Share WMI class

In *step 2*, you view the properties of the `Win32_Share` class. The output of this step looks like this:

```
PS C:\Foo> # 2. Viewing Win32_Share class properties
PS C:\Foo> Get-CimClass -ClassName Win32_Share |
             Select-Object -ExpandProperty CimClassProperties |
               Sort-Object -Property Name |
                 Format-Table -Property Name, CimType

Name            CimType
----            -------
AccessMask       UInt32
AllowMaximum    Boolean
Caption          String
Description      String
InstallDate    DateTime
MaximumAllowed   UInt32
Name             String
Path             String
Status           String
Type             UInt32
```

Figure 13.26: Viewing the Win32_Share class properties

In *step 3*, you use the `Get-CimClass` cmdlet to view the methods available with the `Win32_Share` class, with output like this:

```
PS C:\Foo> # 3. Getting methods of Win32_Share class
PS C:\Foo> Get-CimClass -ClassName Win32_Share |
             Select-Object -ExpandProperty CimClassMethods

Name          ReturnType Parameters                                                                  Qualifiers
----          ---------- ----------                                                                  ----------
Create            UInt32 {Access, Description, MaximumAllowed, Name, Password, Path, Type} {Constructor, Implemented, MappingStrings, Static}
SetShareInfo      UInt32 {Access, Description, MaximumAllowed}                               {Implemented, MappingStrings}
GetAccessMask     UInt32 {}                                                                  {Implemented, MappingStrings}
Delete            UInt32 {}                                                                  {Destructor, Implemented, MappingStrings}
```

Figure 13.27: Viewing the methods in the Win32_Share class

In *step 4*, you get group-related classes in the `ROOT\directory\LDAP` namespace. The step returns just those classes that have the name `ds_group`. As you can see, this matches a few of the classes in this namespace, as follows:

```
PS C:\Foo> # 4. Getting classes in a non-default namespace
PS C:\Foo> Get-CimClass -Namespace root\directory\LDAP |
             Where-Object CimClassName -match '^ds_group'

   NameSpace: ROOT/directory/LDAP

CimClassName              CimClassMethods   CimClassProperties
------------              ---------------   ------------------
ds_groupofuniquenames     {}                {ADSIPath, DS_adminDescription, DS_adminDisplayName…
ds_groupofnames           {}                {ADSIPath, DS_adminDescription, DS_adminDisplayName…
ds_group                  {}                {ADSIPath, DS_adminDescription, DS_adminDisplayName…

ds_grouppolicycontainer   {}                {ADSIPath, DS_adminDescription, DS_adminDisplayName…
```

Figure 13.28: Finding AD group-related classes in the LDAP namespace

In *step 5*, you get the first ten instances of the `ds_group` WMI class (ten of the groups in AD). The output, shown here, includes both the AD group's name and the current members of each AD group. The output of this step looks like this:

```
PS C:\Foo> # 5. Viewing the instances of the ds_group class
PS C:\Foo> Get-CimInstance -Namespace root\directory\LDAP -Classname 'DS_Group' |
             Select-Object -First 10 |
               Format-Table -Property DS_name, DS_Member

DS_name                          DS_Member
-------                          ---------
Administrators                   {CN=Domain Admins,CN=Users,DC=Reskit,DC=Org, CN=Enterprise Admins,CN=Users,DC=Reskit,DC=Org,
                                 CN=Administrator,CN=Users,DC=Reskit,DC=Org}
Users                            {CN=Domain Users,CN=Users,DC=Reskit,DC=Org,
                                 CN=S-1-5-11,CN=ForeignSecurityPrincipals,DC=Reskit,DC=Org,
                                 CN=S-1-5-4,CN=ForeignSecurityPrincipals,DC=Reskit,DC=Orgl
Guests                           {CN=Domain Guests,CN=Users,DC=Reskit,DC=Org, CN=Guest,CN=Users,DC=Reskit,DC=Org}
Print Operators
Backup Operators
Replicator
Remote Desktop Users
Network Configuration Operators
Performance Monitor Users
Performance Log Users
```

Figure 13.29: Finding AD groups and group members

There's more...

In *step 3*, you view the methods in the `Win32_Share` class using `Get-CimClass`. Since you did not specify a namespace, the WMI cmdlet assumes you are interested in the `ROOT\CIV2` namespace. Note that, in this step, the `Create()` method has two important qualifiers – `constructor` and `status`. These two constructor qualifiers tell you that you use the static `Create()` method to construct a new instance of this class (and a new SMB share). Likewise, you use the `Delete()` method to remove an instance of the class.

You can see in the output that a provider, in this case, the Win32 provider, is specified. This provider implements all of the methods shown. You may find that some classes have no implementation, although this is not common.

In *step 5*, you view the first instances in the `ds_group` class. This class contains an instance for every group in the `Reskit.Org` domain. This class contains more information for each group returned by your use of the `Get-ADGroup` cmdlet.

Obtaining WMI Class Instances

In the *Exploring WMI Classes* recipe, you discovered that WMI provides many (over 100) namespaces on each host and thousands of WMI classes. You use the `Get-CimInstance` cmdlet to return the instances of a WMI class. These classes can reside on either the local or a remote host, as you can see in the recipe. This cmdlet returns the WMI instances for a specified WMI class wrapped in a .NET object.

With WMI, you have three ways in which you can use `Get-CimInstance`:

- The first way is to use the cmdlet to return all class occurrences and properties.
- The second way is to use the `-Filter` parameter to specify a WMI filter. When used with `Get-CimInstance`, the WMI filter instructs the command to filter and return some instances of the desired class.
- The third method uses a WMI query using the **WMI Query Language** (**WQL**). A WQL query is, in effect, a SQL statement that instructs WMI to return some or all properties of some or all occurrences of the specified WMI class.

When you use WMI across the network, specifying a filter or a full WMI query can reduce the amount of data transiting the wire and improve performance. This happens as there is there less data transmiteed.

As in previous recipes, you use the `Get-CimInstance` to retrieve WMI class instances using each of these three approaches.

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Using `Get-CimInstance` in the default namespace

```
Get-CimInstance -ClassName Win32_Share
```

2. Getting WMI objects from a non-default namespace

```
$Instance1 = @{
  Namespace = 'ROOT\directory\LDAP'
  ClassName = 'ds_group'
}
Get-CimInstance @Instance1 |
  Sort-Object -Property Name |
    Select-Object -First 10 |
      Format-Table -Property DS_name, DS_distinguishedName
```

3. Using a WMI filter

```
$Filter = "ds_Name LIKE '%operator%' "
Get-CimInstance @Instance1  -Filter $Filter |
  Format-Table -Property DS_Name
```

4. Using a WMI query

```
$Query = @"
  SELECT * from ds_group
    WHERE ds_Name like '%operator%'
"@
Get-CimInstance -Query $Query -Namespace 'root\directory\LDAP' |
  Format-Table DS_Name
```

5. Getting a WMI object from a remote system (`DC1`)

```
Get-CimInstance -CimSession DC1 -ClassName Win32_ComputerSystem |
  Format-Table -AutoSize
```

How it works...

In *step 1*, you use `Get-CimInstance` to return all the instances of the `Win32_Share` class on `SRV1`, with output like this:

```
PS C:\Foo> # 1. Using Get-CimInstance in the default namespace
PS C:\Foo> Get-CimInstance -ClassName Win32_Share

Name   Path       Description
----   ----       -----------
ADMIN$ C:\WINDOWS Remote Admin
C$     C:\        Default share
IPC$              Remote IPC
```

Figure 13.30: Retrieving Win32_Share class instances on SRV1

In *step 2*, you use `Get-CimInstance` to retrieve instances of a class in a non-default namespace that you name explicitly, with output like this:

```
PS C:\Foo> # 2. Getting WMI objects from a non-default namespace
PS C:\Foo> $Instance1 = @{
             Namespace = 'ROOT\directory\LDAP'
             ClassName = 'ds_group'
           }
PS C:\Foo> Get-CimInstance @Instance1 |
             Sort-Object -Property Name |
               Select-Object -First 10 |
                 Format-Table -Property DS_name, DS_distinguishedName

DS_name                                  DS_distinguishedName
-------                                  --------------------
Administrators                           CN=Administrators,CN=Builtin,DC=Reskit,DC=Org
Group Policy Creator Owners              CN=Group Policy Creator Owners,CN=Users,DC=Reskit,DC=Org
Pre-Windows 2000 Compatible Access       CN=Pre-Windows 2000 Compatible Access,CN=Builtin,DC=Reskit,DC=Org
Windows Authorization Access Group       CN=Windows Authorization Access Group,CN=Builtin,DC=Reskit,DC=Org
Allowed RODC Password Replication Group  CN=Allowed RODC Password Replication Group,CN=Users,DC=Reskit,DC=Org
Denied RODC Password Replication Group   CN=Denied RODC Password Replication Group,CN=Users,DC=Reskit,DC=Org
Enterprise Read-only Domain Controllers  CN=Enterprise Read-only Domain Controllers,CN=Users,DC=Reskit,DC=Org
Cloneable Domain Controllers             CN=Cloneable Domain Controllers,CN=Users,DC=Reskit,DC=Org
Protected Users                          CN=Protected Users,CN=Users,DC=Reskit,DC=Org
Key Admins                               CN=Key Admins,CN=Users,DC=Reskit,DC=Org
```

Figure 13.31: Retrieving WMI objects in an explicitly named namespace

In *step 3*, you use a WMI filter, specified with the `-Filter` parameter to the `Get-CimInstance` cmdlet. The output looks like this:

```
PS C:\Foo> # 3. Using a WMI filter
PS C:\Foo> $Filter = "ds_Name LIKE '%operator%' "
PS C:\Foo> Get-CimInstance @Instance1  -Filter $Filter |
             Format-Table -Property DS_Name

DS_Name
-------
Network Configuration Operators
Cryptographic Operators
Access Control Assistance Operators
Print Operators
Account Operators
Server Operators
Backup Operators
```

Figure 13.32: Retrieving WMI objects using a WMI filter

In *step 4*, you use a full WMI query that contains the namespace/class you wish to retrieve and details of which properties and instances WMI should return, like in the following output:

```
PS C:\Foo> # 4. Using a WMI query
PS C:\Foo> $Query = @"
             SELECT * from ds_group
               WHERE ds_Name like '%operator%'
           "@
PS C:\Foo> Get-CimInstance -Query $Query -Namespace 'root\directory\LDAP' |
             Format-Table DS_Name

DS_Name
-------
Network Configuration Operators
Cryptographic Operators
Access Control Assistance Operators
Print Operators
Account Operators
Server Operators
Backup Operator
```

Figure 13.33: Retrieving WMI objects using a WMI query

In *step 5*, you retrieve a WMI object from a remote host, `DC1`. The class retrieved by this step, `Win32_ComputerSystem`, holds details of the host, such as hostname, domain name, and total physical memory, as you can see in the following output:

```
PS C:\Foo> # 5. Getting a WMI object from a remote system (DC1)
PS C:\Foo> Get-CimInstance -CimSession DC1 -ClassName Win32_ComputerSystem
             Format-Table -AutoSize

Name PrimaryOwnerName Domain     TotalPhysicalMemory Model           Manufacturer          PSComputerName
---- ---------------- ------     ------------------- -----           ------------          --------------
DC1  Book Readers     Reskit.Org 4331601920          Virtual Machine Microsoft Corporation DC1
```

Figure 13.34: Retrieving WMI information from DC1

There's more...

In *step 4*, you create a WMI query and use it in a call to `Get-CimInstance`. The cmdlet uses this query to return all properties on any class instance whose name contains the character `"operator"`, using WMI's wildcard syntax. This query returns all properties in the groups that include printer operators and server operators, as you can see in the output from this step.

In *step 5*, you return details of the `DC1` host. You used `Get-CimInstance` to return the single occurrence of the `Win32_ComputerSystem` class. The output shows that the `DC1` host has 4 GB of memory. If you are using virtualization to implement this host, you may see a different value depending on how you configured the VM.

Using WMI Methods

In many object-oriented programming languages, including PowerShell, a method is an action that an object can carry out. With WMI, any WMI class can have methods that do something useful in relation to an instance or the class itself. For example, the `Win32_Share` class has a `Delete()` method to delete a given SMB share. The class also has the `Create()` static method, which creates a new SMB share.

In many cases, WMI methods duplicate what you can do with other PowerShell cmdlets. You could, for example, use the `New-SmbShare` cmdlet to create a new SMB share rather than using the `Create()` static method of the `Win32_Share` class.

WMI methods are of two types: instance methods and static methods. An instance method operates on a specific instance – for example, deleting a particular SMB share. Classes also provide static methods, which do not need a reference to any existing class instances. For example, you can use the `Create()` static method to create a new SMB share (and a new occurrence in the `Win32_Share` class).

Getting ready

This recipe uses `SRV1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

How to do it...

1. Reviewing methods of `Win32_Share` class on `SRV1`

```
Get-CimClass -ClassName Win32_Share |
  Select-Object -ExpandProperty CimClassMethods
```

2. Reviewing properties of `Win32_Share` class

```
Get-CimClass -ClassName Win32_Share |
  Select-Object -ExpandProperty CimClassProperties |
    Format-Table -Property Name, CimType
```

3. Creating a new SMB share using the `Create()` static method:

```
$NewShareDetails = @{
  Name        = 'TestShare1'
  Path        = 'C:\Foo'
  Description = 'Test Share'
  Type        = [uint32] 0 # disk
}
$CimShareHT = @{
  ClassName  = 'Win32_Share'
  MethodName = 'Create'
  Arguments  = $NewShareDetails
}
Invoke-CimMethod @CimShareHT
```

4. Viewing the new SMB share

```
Get-SMBShare -Name 'TestShare1'
```

5. Viewing the new SMB share using `Get-CimInstance`

```
Get-CimInstance -Class Win32_Share -Filter "Name = 'TestShare1'"
```

6. Removing the share

```
Get-CimInstance -Class Win32_Share -Filter "Name = 'TestShare1'" |
  Invoke-CimMethod -MethodName Delete
```

How it works...

In *step 1*, you use `Get-CimClass` to retrieve and display the methods provided by the `Win32_Share` WMI class. The output is as follows:

```
PS C:\Foo> # 1. Reviewing methods of Win32_Share class on SRV1
PS C:\Foo> Get-CimClass -ClassName Win32_Share |
             Select-Object -ExpandProperty CimClassMethods

Name          ReturnType Parameters                                                                Qualifiers
----          ---------- ----------                                                                ----------
Create            UInt32 {Access, Description, MaximumAllowed, Name, Password, Path, Type} {Constructor, Implemented, MappingStrings, Static}
SetShareInfo      UInt32 {Access, Description, MaximumAllowed}                                   {Implemented, MappingStrings}
GetAccessMask     UInt32 {}                                                                      {Implemented, MappingStrings}
Delete            UInt32 {}                                                                      {Destructor, Implemented, MappingStrings}
```

Figure 13.35: Reviewing the methods contained in the Win32_Share WMI class

In *step 2*, you use the `Get-CimClass` cmdlet to get the properties of each instance of the `Win32_Share` class, producing the following:

```
PS C:\Foo> # 2. Reviewing properties of Win32_Share class
PS C:\Foo> Get-CimClass -ClassName Win32_Share |
             Select-Object -ExpandProperty CimClassProperties |
               Format-Table -Property Name, CimType

Name             CimType
----             -------
Caption           String
Description       String
InstallDate     DateTime
Name              String
Status            String
AccessMask        UInt32
AllowMaximum     Boolean
MaximumAllowed    UInt32
Path              String
Type              UInt32
```

Figure 13.36: Reviewing the properties of an instance of the Win32_Share class

With *step 3*, you use the `Invoke-CimMethod` cmdlet to invoke the `Create()` method of the `Win32_Share` class and create a new SMB share on `SRV1`, with output like this:

```
PS C:\Foo> # 3. Creating a new SMB share using the Create() static method
PS C:\Foo> $NewShareDetails = @{
             Name        = 'TestShare1'
             Path        = 'C:\Foo'
             Description = 'Test Share'
             Type        = [uint32] 0 # disk
           }
PS C:\Foo> $CimShareHT = @{
             ClassName  = 'Win32_Share'
             MethodName = 'Create'
             Arguments  = $NewShareDetails
           }
PS C:\Foo> Invoke-CimMethod @CimShareHT

ReturnValue PSComputerName
----------- --------------
          0
```

Figure 13.37: Creating a new SMB share using WMI

In *step 4*, you use the `Get-SMBShare` cmdlet to get the SMB share information for the share you created in the previous step, producing output like this:

```
PS C:\Foo> # 4. Viewing the new SMB share
PS C:\Foo> Get-SMBShare -Name 'TestSharel'

Name       ScopeName Path   Description
----       --------- ----   -----------
TestSharel *         C:\Foo Test Share
```

Figure 13.38: Viewing the newly created SMB share using Get-SMBShare

In *step 5*, you use `Get-CimInstance` to view the details of the share via WMI. This step produces the following output:

```
PS C:\Foo> # 5. Viewing the new SMB share using Get-CimInstance
PS C:\Foo> Get-CimInstance -Class Win32_Share -Filter "Name = 'TestSharel'"

Name       Path   Description
----       ----   -----------
TestSharel C:\Foo Test Share
```

Figure 13.39: Viewing the newly created SMB share using Get-CimInstance

In the final step in this recipe, *step 6*, you use `Invoke-CimMethod` to delete a specific share (the one you created in *step 3*). The output is as follows:

```
PS C:\Foo> # 6. Removing the share
PS C:\Foo> Get-CimInstance -Class Win32_Share -Filter "Name = 'TestSharel'" |
             Invoke-CimMethod -MethodName Delete

ReturnValue PSComputerName
----------- ---------------
          0
```

Figure 13.40: Removing the SMB share

There's more...

In *step 3*, you create a new SMB share using the `Invoke-CimMethod` cmdlet. This cmdlet takes a hash table containing the properties and property values for the new SMB share. The cmdlet returns an object containing a `ReturnCode` property. A return code of `0` indicates success – in this case, it means that WMI has created a new share.

You need to consult the documentation for other values of the return code. For the `Win32_Share` class, you can find more online documentation at `https://learn.microsoft.com/windows/win32/cimwin32prov/create-method-in-class-win32-share`. This page shows the return codes that the `Create()` method could generate and what those return codes indicate.

For example, a return code of 8 would suggest that you attempted to create a share whose name already exists. If you plan to use WMI in production scripting, consider testing for non-zero return codes and handling common errors gracefully.

In *step 6*, you use a WMI method, `Delete()`, to delete the previously created SMB share. You delete this share by first using the `Get-CimInstance` with a WMI filter to retrieve the share(s) to be deleted. You then pipe these share objects to the `Invoke-CimMethod` cmdlet and invoke the `Delete()` method on the instance passed in the pipeline. The approach taken in *step 6* is a common way to remove WMI class instances of this class and, for that matter, any WMI class.

Using WMI Events

A key feature of WMI is its event handling. Thousands of events can occur within a Windows system that might possibly be of interest. For example, you might want to know if someone adds a new member to a high-privilege AD group such as Enterprise Admins. You can tell WMI to notify you when such an event occurs, then take whatever action is appropriate. For example, you might print out an updated list of group members when group membership changes occur. You could also check a list of users who should be members of the group and take some action if the user added is not authorized.

Events are handled both by WMI itself and by WMI providers. WMI can signal an event should a change be detected in a CIM class – that is, any new, updated, or deleted class instance. You can detect changes too in entire classes or namespaces. WMI calls these events **intrinsic** events. One common intrinsic event would occur when you (or Windows) start a new process and, by doing so, WMI adds a new instance to the `Win32_Process` class (contained in the `ROOT/CIMV2` namespace).

WMI providers can also implement events. These are known as **extrinsic** WMI events. The AD WMI provider, for example, implements an event that fires any time the membership of an AD group changes. The Windows registry provider also provides an extrinsic event that detects changes to the registry, such as a new registry key or an updated registry value.

To use WMI event management, you first create an event subscription. The event subscription tells WMI which event you want it to track. Additionally, you can define an event handler that tells WMI what you want to do if the event occurs. For example, if a new process starts, you may wish to display the event's details. If an AD group membership changes, you might want to check to see if any group members are not authorized and report the fact, or possibly even delete the invalid group member.

For more details on how you can receive WMI events, see `https://learn.microsoft.com//windows/win32/wmisdk/receiving-a-wmi-event`.

For information about the types of events to receive, see `https://learn.microsoft.com/windows/win32/wmisdk/determining-the-type-of-event-to-receive`.

There are two types of WMI eventing you can utilize. In this recipe, you create and handle temporary WMI events that work within a PowerShell session. If you close a session, WMI stops tracking all the events you registered for in that session. In the *Implementing Permanent WMI Eventing* recipe, you will look at creating event subscriptions independent of the current PowerShell session.

When you register for a temporary event, you can provide WMI with a script block that you want WMI to execute when the event occurs. WMI runs this script block in the background, inside a PowerShell job.

When you register for a WMI event, PowerShell creates this job in which it runs the action script. As with all PowerShell jobs, you use `Receive-Job` to view any output generated by the script. If your script block contains `Write-Host` statements, PowerShell sends any output directly to the console (not the background job). You can also register for a WMI event without specifying an action block. In that case, WMI queues the events, and you can use `Get-WinEvent` to retrieve the event details.

When WMI detects an event, it generates an event record containing the event's details. These event records can be useful in helping you with more information on the event, but the records are not a complete snapshot of the event. You can register for a WMI event should the membership of an AD group change, and receive details such as the new member. However, the event record does not contain details of the user who modified the group's membership or of the host's IP address used to effect the unauthorized change.

Getting ready

This recipe uses `DC1`, a domain controller in the Reskit domain. You have used this host in many of the previous chapters. You have installed PowerShell 7 and VS Code on this host. Also, you should have previously created the user `Malcolm` in the AD.

How to do it...

1. Registering an intrinsic event

```
$Query1 = "SELECT * FROM __InstanceCreationEvent WITHIN 2
          WHERE TargetInstance ISA 'Win32_Process'"
$EventHT = @{
  Query            = $Query1
  SourceIdentifier = 'NewProcessEvent'
}
Register-CimIndicationEvent @EventHT
```

2. Running `Notepad` to trigger the event

```
notepad.exe
```

3. Getting the new process event

```
$NotepadEvent = Get-Event -SourceIdentifier 'NewProcessEvent' |
                  Select-Object -Last 1
```

4. Displaying event details

```
$NotepadEvent.SourceEventArgs.NewEvent.TargetInstance
```

5. Unregistering the event

```
Unregister-Event -SourceIdentifier 'NewProcessEvent'
```

6. Registering an event query based on the registry provider

```
New-Item -Path 'HKLM:\SOFTWARE\Packt' | Out-Null
$Query2 = "SELECT * FROM RegistryValueChangeEvent
            WHERE Hive='HKEY_LOCAL_MACHINE'
              AND KeyPath='SOFTWARE\\Packt' AND ValueName='MOLTUAE'"
$Action2 = {
  Write-Host -Object "Registry Value Change Event Occurred"
  $Global:RegEvent = $Event
}
$RegisterHT = @{
  Query  = $Query2
  Action = $Action2
  Source = 'RegChange'
```

```
}
Register-CimIndicationEvent @RegisterHT
```

7. Creating a new registry key and setting a value entry

```
$Query3HT = [ordered] @{
  Type  = 'DWord'
  Name  = 'MOLTUAE'
  Path  = 'HKLM:\Software\Packt'
  Value = 42
}
Set-ItemProperty @Query3HT
Get-ItemProperty -Path HKLM:\SOFTWARE\Packt
```

8. Unregistering the event

```
Unregister-Event -SourceIdentifier 'RegChange'
```

9. Examining event details

```
$RegEvent.SourceEventArgs.NewEvent
```

10. Creating a WQL event query

```
$Group = 'Enterprise Admins'
$Query1 = @"
  SELECT * From __InstanceModificationEvent Within 5
   WHERE TargetInstance ISA 'ds_group' AND
         TargetInstance.ds_name = '$Group'
"@
```

11. Creating a temporary WMI event registration

```
$EventHT= @{
  Namespace = 'ROOT\directory\LDAP'
  SourceID  = 'DSGroupChange'
  Query     = $Query1
  Action    = {
    $Global:ADEvent = $Event
    Write-Host 'We have a group change'
  }
}
Register-CimIndicationEvent @EventHT
```

12. Adding a user to the `Enterprise Admins` group

```
Add-ADGroupMember -Identity 'Enterprise Admins' -Members Malcolm
```

13. Viewing the newly added user within the group

```
$ADEvent.SourceEventArgs.NewEvent.TargetInstance |
  Format-Table -Property DS_sAMAccountName,DS_Member
```

14. Unregistering the event

```
Unregister-Event -SourceIdentifier 'DSGroupChange'
```

How it works...

In *step 1*, you register for an intrinsic event that occurs whenever Windows starts a process. The registration does not include an action block. In *step 2*, you run `Notepad.exe` to trigger the event. In *step 3*, you use `Get-WinEvent` to retrieve details of the event. These three steps produce no console output.

In *step 4*, you view details of the process startup event, with output like this:

```
PS C:\Foo> # 4. Displaying event details
PS C:\Foo> $NotepadEvent.SourceEventArgs.NewEvent.TargetInstance

ProcessId Name        HandleCount WorkingSetSize VirtualSize
--------- ----        ----------- -------------- -----------
10184     notepad.exe 214         18427904       22034896
```

Figure 13.41: Displaying event details

In *step 5*, you remove the registration for the process start event. This step generates no output. In *step 6*, you register a new event subscription using a WMI query that targets the WMI provider, with output like this:

```
PS C:\Foo> # 6. Registering an event query based on the registry provider
PS C:\Foo> New-Item -Path 'HKLM:\SOFTWARE\Packt' | Out-Null
PS C:\Foo> $Query2 = "SELECT * FROM RegistryValueChangeEvent
                 WHERE Hive='HKEY_LOCAL_MACHINE'
                   AND KeyPath='SOFTWARE\\Packt' AND ValueName='MOLTUAE'"
PS C:\Foo> $Action2 = {
             Write-Host -Object "Registry Value Change Event Occurred"
             $Global:RegEvent = $Event
           }

PS C:\Foo> $RegisterHT = @{
             Query  = $Query2
             Action = $Action2
             Source = 'RegChange'
           }
PS C:\Foo> Register-CimIndicationEvent @RegisterHT

Id     Name            PSJobTypeName   State         HasMoreData     Location            Command
--     ----            -------------   -----         -----------     --------            -------
1      RegChange                       NotStarted    False                               …
```

Figure 13.42: Registering for a registry provider-based event

With the event registration complete, in *step 7*, you create a new registry key and set a registry key value to test the event subscription. The output of this step looks like this:

```
PS C:\Foo> # 7. Creating a new registry key and setting a value entry
PS C:\Foo> $Query2HT = [ordered] @{
             Type  = 'DWord'
             Name  = 'MOLTUAE'
             Path  = 'HKLM:\Software\Packt'
             Value = 42
           }
PS C:\Foo> Set-ItemProperty @Query2HT

Registry Value Change Event Occurred

PS C:\Foo> Get-ItemProperty -Path HKLM:\SOFTWARE\Packt

MOLTUAE       : 42
PSPath        : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Packt
PSParentPath  : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE
PSChildName   : Packt
PSDrive       : HKLM
PSProvider    : Microsoft.PowerShell.Core\Registry
```

Figure 13.43: Invoking the WMI registry event

In *step 8*, you remove the registry event to avoid more event handling and event output. This step generates no console output.

In *step 9*, you examine the output WMI generated based on the registry changes you made in *step 7*. The event details look like this:

```
PS C:\Foo> # 9. Examining event details
PS C:\Foo> $RegEvent.SourceEventArgs.NewEvent

SECURITY_DESCRIPTOR :
TIME_CREATED        : 133136814589107181
Hive                : HKEY_LOCAL_MACHINE
keyPath             : SOFTWARE\Packt
ValueName           : MOLTUAE
PSComputerName      :
```

Figure 13.44: Examining a registry change WMI event

In *step 10*, you create a WQL query that captures changes to the `Enterprise Admins` AD group, generating no output. In *step 11*, you use the query to create a temporary WMI event that fires when the group membership changes. The output looks like this:

```
PS C:\Foo> # 11. Creating a temporary WMI event registration
PS C:\Foo> $EventHT= @{
             Namespace = 'ROOT\directory\LDAP'
             SourceID  = 'DSGroupChange'
             Query     = $Query1
             Action    = {
               $Global:ADEvent = $Event
               Write-Host 'We have a group change'
             }
           }
PS C:\Foo> Register-CimIndicationEvent @EventHT

Id  Name            PSJobTypeName   State        HasMoreData  Location   Command
--  ----            -------------   -----        -----------  --------   -------
2    DSGroupChange                   NotStarted   False                    …
```

Figure 13.45: Examining a registry change WMI event

To test this new directory change event, in *step 12*, you add a user to the `Enterprise Admins` AD group, generating output like this:

```
PS C:\Foo> # 12. Adding a user to the Enterprise Admins group
PS C:\Foo> Add-ADGroupMember -Identity 'Enterprise Admins' -Members Malcolm
PS C:\Foo> We have a group change
```

Figure 13.46: Triggering an AD group membership change event

In *step 13*, you examine the details of the event you generated in the previous step, with output like this:

```
PS C:\Foo> # 13. Viewing the newly added user within the group
PS C:\Foo> $ADEvent.SourceEventArgs.NewEvent.SourceInstance |
             Format-Table -Property DS_sAMAccountName,DS_Member

DS_sAMAccountName DS_Member
----------------- ---------
Enterprise Admins {CN=Malcolm,OU=IT,DC=Reskit,DC=Org,
                   CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org,
                   CN=Administrator,CN=Users,DC=Reskit,DC=Org}
```

Figure 13.47: Examining the details of the AD group membership change event

In the final step in this recipe, *step 14*, you unregister for the AD group membership change. This step generates no output.

There's more...

In *step 7*, you test the registry event handling, which includes an event action script block that you want PowerShell to execute when the event occurs. Since the script block you specified in *step 6* contains a `Write-Host` statement, you see the output on the PowerShell console.

In *step 9*, you examine the details WMI generated when the WMI registry change event occurred. As with other events, the event details omit potentially critical information. For example, the event does not tell you which user made this change or provide the new value of the registry value. You can examine the Windows Security event log to discover the user logged on to that system (and, therefore, the user who made the change). And you can use `Get-ItemProperty` to determine the new value for this registry key property.

The registry value name, `MOLTUAE`, is an acronym for "meaning of life, the universe, and everything" – a famous line from a well-known book. The value, of course, is `42`.

In *step 12*, you add the user `Malcolm` to the group, You created the `Malcolm` user in *Chapter 5*, in the recipe *Adding users to active directory using a CSV file*.

In *step 14*, you explicitly remove the registration for the AD change event. Alternatively, you could have closed the current PowerShell session, removing the event subscriptions.

If you use WMI eventing, you may find other inconsistencies between different event types. In *step 7*, you examine the output from a registry change event, while in *step 13*, you review the result from an AD change event. In the latter, you can see in the event details the membership of the Enterprise Admins group, but in the registry change event, you do not see the new registry value.

Implementing Permanent WMI Eventing

In the *Managing WMI events* recipe, you used PowerShell's WMI event handling capability and used temporary event handling – the event handlers are active only as long as the PowerShell session is active and a user logs in to the host. You created an event subscription in that recipe and handled the events as your system generated them. This temporary event handling is a great troubleshooting tool that works well if you are logged in and running PowerShell.

WMI also provides permanent event handling. WMI permanent event registrations enable WMI to detect and act on some events, even if no one is logged on. You configure WMI to subscribe and handle events as they occur without using an active and open session. With permanent event handling, you configure WMI to subscribe to specific events, for instance, adding a new member to a high-privilege AD group such as `Enterprise Admins`, along with an action that WMI should perform when that event occurs. For example, you could create a log entry, a complete report, or possibly send an email message to report on the event if/when it occurs.

WMI in Windows defines several different types of permanent event consumers you can use to set a permanent event:

- **Active Script Consumer**: You use this to run a specific VBS script.
- **Log File Consumer**: This handler writes details of events to event log files.
- **NT Event Log Consumer**: This consumer writes event details into the Windows event log.
- **SMTP Event Consumer**: You can use this consumer to send a simple SMTP email message when an event occurs.
- **Command Line Consumer**: With this consumer, you can run a program, such as PowerShell 7, and pass a script filename. When the event occurs, the script has access to the event details and can do pretty much anything you can do in PowerShell.

Microsoft developed the Active Script consumer in the days of Visual Basic and VBS scripts. Unfortunately, this consumer does not support PowerShell scripts. On the other hand, the command-line WMI permanent event handler enables you to run any programs you wish when the event handler detects an event occurrence. In this recipe, you ask WMI to run `pwsh.exe` and execute a specific script file when the event fires.

Managing permanent event handling is similar to the temporary WMI events you explored in the *Managing WMI events* recipe. But there are some significant differences. You tell WMI which event to trap and what to do when that event occurs.

You add a WMI class occurrence to three WMI classes, as you see in the recipe, to implement a permanent event handler as follows:

1. Define an event filter: The event filter specifies the specific event that WMI should handle. You do this by adding a new instance to the particular event class you want WMI to detect. This event filter is the same as in the previous recipe, *Managing WMI events*.
2. Define an event consumer: In this step, you define the action you want WMI to take when the event occurs.
3. Bind the event filter and event consumer: This step adds a new occurrence to an event-binding class. This occurrence directs WMI to act (invoke the event consumer) whenever WMI detects that a specific WMI event (specified in the event filter) has occurred.

The AD WMI provider implements a wide range of AD-related events to which you can subscribe. WMI namespaces typically contain specific event classes that can detect when anything changes within the namespace. The namespace `ROOT/Directory/LDAP` has a system class named `__InstanceModificationEvent`.

A small word of caution is appropriate. You need to be very careful when working with WMI permanent event handling. A best practice is understanding how you remove the objects related to the permanent event handler. Until you explicitly remove these records, WMI continues to monitor your host for the events, which can unnecessarily consume host resources.

This recipe also demonstrates a useful approach – creating PowerShell functions to display the event subscription and then remove the subscription fully. This can be useful to ensure you do not end up with unnecessary event details in WMI.

Finally, be careful when changing an event filter's refresh time (specified in the WMI event filter). Decreasing the event refresh time can consume additional CPU and memory. A refresh rate of once per second or every 5 seconds is possibly overly excessive. For testing, shorter refresh times can be helpful, but checking every 10 seconds or longer is more than adequate for production.

Getting ready

This recipe uses `DC1`, a domain-joined host. You have installed PowerShell 7 and VS Code on this host.

Also, ensure that the user `Malcolm` is not an `Enterprise Admins` AD group member.

How to do it...

1. Creating a list of valid users for the Enterprise Admins group

```
$OKUsersFile = 'C:\Foo\OKUsers.Txt'
$OKUsers  =  @'
Administrator
JerryG
'@
$OKUsers |
  Out-File -FilePath $OKUsersFile
```

2. Defining helper functions to get/remove permanent events

```
Function Get-WMIPE {
  '*** Event Filters Defined ***'
  Get-CimInstance -Namespace root\subscription -ClassName __
EventFilter |
    Where-Object Name -eq "EventFilter1" |
     Format-Table Name, Query
  '***Consumer Defined ***'
  $NS = 'ROOT\subscription'
  $CN = 'CommandLineEventConsumer'
  Get-CimInstance -Namespace $ns -Classname  $CN |
    Where-Object {$_.name -eq "EventConsumer1"}  |
     Format-Table Name, Commandlinetemplate
  '***Bindings Defined ***'
  Get-CimInstance -Namespace root\subscription -ClassName __
FilterToConsumerBinding |
    Where-Object -FilterScript {$_.Filter.Name -eq "EventFilter1"} |
      Format-Table Filter, Consumer
}
Function Remove-WMIPE {
  Get-CimInstance -Namespace root\subscription __EventFilter |
    Where-Object Name -eq "EventFilter1" |
      Remove-CimInstance
  Get-CimInstance -Namespace root\subscription
CommandLineEventConsumer |
    Where-Object Name -eq 'EventConsumer1' |
```

```
      Remove-CimInstance
  Get-CimInstance -Namespace root\subscription __
FilterToConsumerBinding  |
    Where-Object -FilterScript {$_.Filter.Name -eq 'EventFilter1'}
|
      Remove-CimInstance
}
```

3. Creating an event filter query

```
$Group = 'Enterprise Admins'
$Query = @"
  SELECT * From __InstanceModificationEvent Within 10
   WHERE TargetInstance ISA 'ds_group' AND
         TargetInstance.ds_name = '$Group'
"@
```

4. Creating the event filter

```
$Param = @{
  QueryLanguage =  'WQL'
  Query          =  $Query
  Name           =  "EventFilter1"
  EventNameSpace =  "root/directory/LDAP"
}
$IHT = @{
  ClassName = '__EventFilter'
  Namespace = 'root/subscription'
  Property  = $Param
}
$InstanceFilter = New-CimInstance @IHT
```

5. Creating the `Monitor.ps1` script run when the WMI event occurs

```
$Monitor = @'
$LogFile   = 'C:\Foo\Grouplog.Txt'
$Group     = 'Enterprise Admins'
"On:  [$(Get-Date)]  Group [$Group] was changed" |
  Out-File -Force $LogFile -Append -Encoding Ascii
$ADGM = Get-ADGroupMember -Identity $Group
```

```
# Display who's in the group
"Group Membership"
$ADGM | Format-Table Name, DistinguishedName |
  Out-File -Force $LogFile -Append  -Encoding Ascii
$OKUsers = Get-Content -Path C:\Foo\OKUsers.txt
# Look at who is not authorized
foreach ($User in $ADGM) {
  if ($User.SamAccountName -notin $OKUsers) {
    "Unauthorized user [$($User.SamAccountName)] added to $Group"  |
      Out-File -Force $LogFile -Append  -Encoding Ascii
  }
}
"**********'n'n" |
Out-File -Force $LogFile -Append -Encoding Ascii
'@
$Monitor | Out-File -Path C:\Foo\Monitor.ps1
```

6. Creating a WMI event consumer which consumer runs PowerShell 7 to execute `C:\Foo\Monitor.ps1`

```
$CLT = 'Pwsh.exe -File C:\Foo\Monitor.ps1'
$Param =[ordered] @{
  Name                = 'EventConsumer1'
  CommandLineTemplate = $CLT
}
$ECHT = @{
  Namespace = 'root/subscription'
  ClassName = "CommandLineEventConsumer"
  Property  = $param
}
$InstanceConsumer = New-CimInstance @ECHT
```

7. Binding the filter and consumer

```
$Param = @{
  Filter   = [ref]$InstanceFilter
  Consumer = [ref]$InstanceConsumer
}
```

```
$IBHT = @{
  Namespace = 'root/subscription'
  ClassName = '__FilterToConsumerBinding'
  Property  = $Param
}
$InstanceBinding = New-CimInstance   @IBHT
```

8. Viewing the event registration details

```
Get-WMIPE
```

9. Adding a user to the `Enterprise Admins` group

```
Add-ADGroupMember -Identity 'Enterprise admins' -Members Malcolm
```

10. Viewing `Grouplog.txt` file

```
Get-Content -Path C:\Foo\Grouplog.txt
```

11. Tidying up

```
Remove-WMIPE   # invoke this function you defined above
$RGMHT = @{
 Identity = 'Enterprise admins'
 Member   = 'Malcolm'
 Confirm  = $false
}
Remove-ADGroupMember @RGMHT
Get-WMIPE       # ensure you have removed the event handling
```

How it works...

In *step 1*, you create a text file containing the `SAMAccountName` of users that you have specified should be a member of the `Enterprise Admins` group. In *step 2*, you create two helper functions to view and delete the WMI class instances that handle the event. In *step 3*, you create an event filter query, which, in *step 4,* you add to WMI. These steps produce no output.

When you change the group membership, the permanent WMI event occurs. At that point, you want WMI to run a specific PowerShell script. In *step 5*, you create a file, `C:\Foo\Monitor.ps1`. This step creates no console output.

In *step 6*, you create a new event consumer, telling WMI to run the monitor script to detect the event. Then in *step 7*, you bind the event consumer and the event filter to complete setting up a permanent event handler. These two steps also produce no output.

In *step 8*, you use the `Get-WMIPE` function you defined in *step 2* earlier. The function outputs the details of the event filter, with the output of this step as follows:

```
PS C:\Foo> # 8. Viewing the event registration details
PS C:\Foo> Get-WMIPE

*** Event Filters Defined ***

Name         Query
----         -----
EventFilter1   SELECT * From __InstanceModificationEvent Within 10
                WHERE TargetInstance ISA 'ds_group' AND
                      TargetInstance.ds_name = 'Enterprise Admins'

***Consumer Defined ***

Name           Commandlinetemplate
----           -------------------
EventConsumer1 Pwsh.exe -File C:\Foo\Monitor.ps1

***Bindings Defined ***

Filter                                Consumer
------                                --------
__EventFilter (Name = "EventFilter1") CommandLineEventConsumer (Name = "EventConsumer1")
```

Figure 13.48: Viewing event registration details

In *step 9*, you test the permanent event handling by adding a new user (`Malcolm`) to the `Enterprise Admins` group. This step does not generate console output because you did not add `Write-Host` statements to `Monitor.ps1`.

In *step 10*, you view the `Grouplog.txt` file, with output like this:

```
PS C:\Foo> # 10. Viewing Grouplog.txt file
PS C:\Foo> Get-Content -Path C:\Foo\Grouplog.txt

On:  [11/25/2022 15:44:54]  Group [Enterprise Admins] was changed

Name          DistinguishedName
----          -----------------
Malcolm       CN=Malcolm,OU=IT,DC=Reskit,DC=Org
Jerry Garcia  CN=Jerry Garcia,OU=IT,DC=Reskit,DC=Org
Administrator CN=Administrator,CN=Users,DC=Reskit,DC=Org

Unauthorized user [Malcolm] added to Enterprise Admins
**********************************
```

Figure 13.49: Viewing event registration details

In the final step in this recipe, *step 11*, you tidy up by invoking the `Remove-WMIPE` function you defined in *step 2*. This removes the event details from WMI. At the end of this step, you run the `Get-WMIPE` function to ensure you have deleted all the event subscription class instances. The output of this step looks like this:

```
PS C:\Foo> # 11. Tidying up
PS C:\Foo> Remove-WMIPE   # invoke this function you defined above
PS C:\Foo> $RGMHT = @{
             Identity = 'Enterprise admins'
             Member   = 'Malcolm'
             Confirm  = $false
           }
PS C:\Foo> Remove-ADGroupMember @RGMHT
PS C:\Foo> Get-WMIPE       # ensure you have removed the event handling

*** Event Filters Defined ***
***Consumer Defined ***
***Bindings Defined ***
```

Figure 13.50: Tidying up

There's more...

In *step 5*, you create a script that you want WMI to run any time the membership of the `Enterprise Admins` group changes. This script writes details to a text file (`Grouplog.txt`) containing the time the event occurred, the new membership, and whether this group now includes any unauthorized users. You could modify `Monitor.ps1` to send an email to an administrative mailbox or remove the unauthorized user. You could also look in the Windows Security event log to find the most recent user to log on to the server.

In *step 10*, you view the output generated by the `Monitor.ps1` script. Note that it can take a few seconds for the permanent event handler to run the script to completion and generate output to the log file.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/SecNet`