PowerShell and WMI Covers 150 practical techniques 读书笔记(1)(21-34-24)

Google for powershell tutorial

By Richard Siddaway

Part 1 Tools of the trade

1 Solving administrative challenges

1.1 Administrative challenges
Hardware costs is decreasing,
while Infrastructure complexity and Administration costs are increasing.

1.1.1 Too many machines

1.1.2 Too many changes

1.1.3 Complexity and Understanding

1.2 Automation: the way forward

1.3 PowerShell overview

1.3.1 PowerShell scope

1.3.2 PowerShell and .NET
PowerShell is based on .NET and can access most of the .NET Framework.
eg:
```
Get-Service wi* | Format-Table -AutoSize
```

1.3.3 Breaking the curve

1.4 WMI overview

1.4.1 What is WMI?
Windows Management Instrumentation.
automatically installed with Windows.
eg: to look at the WMI classes available for working with disks
```
Get-WmiObject -List *disk* | sort name | select name
```
Note: the CIM_class is the parent, corresponding to the definition supplied by the DMTF; the Win_32 classes are child classes that Microsoft has implemented.

1.4.2 Is WMI really too hard?

1.5 Automation with WMI and PowerShell
eg: VBScript to retrieve process information
```
set objWMIService = GetObject("winmgmts:" & "{impersonationlevel=impersonate}!\\" & ".\root\cimv2")
set colProcesses = objWMIService.ExecQuery ("SELECT * FROM Win32_Process")
for each objProcess in colProcesses
    WScript.Echo " "
    WScript.Echo "Process Name : " + objProcess.Name
    WScript.Echo "Handle   : " + objProcess.Handle
    WScript.Echo "Total Handles : " + Cstr(objProcess.HandleCount)
    WScript.Echo "ThreadCount : " + Cstr(objProcess.ThreadCount)
    WScript.Echo "Path"  : " + objProcess.ExecutablePath
```

next
Note: The script starts by creating an object, objWMIService, to enable interrogation of the WMI service. A list of active processes is retrieved by running a WQL querry. The collection of processes is iterated through, and you write a caption and the value of a particular property to the screen.

eg: PowerShell translation
```
$procs = Get-WmiObject Query "SELECT * FROM Win32_Process"
foreach ($proc in $procs) {
    Write-Host "Name   :" $proc.ProcessName
    Write-Host "Handle   :" $proc.Handle
    Write-Host "Total Handles :" $proc.Handles
    Write-Host "ThreadCount  :" $proc.ThreadCount
    Write-Host "Path    :" $proc.ExecutablePath
}
```
Note: You first run the WMI query to select the information you need and put the results into a variable. The variable is a collection of objects reresenting the different processes. You can then loop through the collection of processes (using the foreach command), and for each process in that collection use the Write-Host cmdlet to output a caption and the value of the properties you're interested in.

eg: PowerShell command in a single line
```
Get-WmiObject Win32_Process | Format-Table ProcessName, Handle, Handles, ThreadCount, ExecutablePath -AutoSize
```
Note: This final version uses the Get-WmiObject cmdlet directly. Get-WmiObject returns an object for each process, and you use the PowerShell pipeline to pass them into a Format-Table cmdlet. This combines the data selection and display functionality and roduces neatly formatted tabular outut. (to display the output in a list format, substitute Format-List for Format-Table.)

1.6 Putting PowerShell and WMI to work
eg1: Shutting down all the Windows machines in your data center.

eg2: auditing a large number of machines to discover their capabilities.

1.6.1 Example 1: Shutting down a data center
eg: Shut down a data center
```
Import-Csv computers.csv |
foreach {
    (Get-WmiObject -Class Win32_Operating System -ComputerName $_.Computer ).Win32Shutdown(5)
}
```
Note: This script uses a CSV file called computers.csv, which contains a list of computer names.
eg :
```
Computer
W08R2CS01
W08R2CS02
W08R2SQL08
W08R2SQL08A
WSS08
DC02
```
Note: The $_ symbol refers to the current object on the pipeline, and the Computer part comes from the CSV header.

1.6.2 Example 2: Auditing hundreds of machines
The audit should return the following information:
Server make and model
CPU data (numbers, cores, logical processors, and speed)
Memory
Windows version and service pack level
eg: to gather basic information from many machines.
```
Import-Csv computers.csv |
foreach {
     $system = " " | select Name, Make, Model, CPUs, Cores, LogProc, Speed, Memory, Windows, SP
     $server = Get-WmiObject -Class Win32_ComputerSystem  -ComputerName $_.Computer
```

```
    $system.Name = $server.Name
    $system.Make = $server.Manufacturer
    $system.Model = $server.Model
    $system.Memory = $server.TotalPhysicalMemory
    $system.CPUs = $server.NumberOfProcessors
    $cpu = Get-WmiObject -Class Win32_Processor  -ComputerName $_.Computer | select -First 1
    $system.Speed = $cpu.MaxClockSpeed
    $os = Get-WmiObject -Class Win32_OperatingSystem  -ComputerName $_.Computer
    $system.Windows = $os.Caption
    $system.SP = $os.ServicePackMajorVersion
    if (($os.Version -split "\.")[0] -ge 6) {
        $system.Cores = $cpu.NumberOfCores
        $system.LogProc = $cpu.NumberOfLogicalProcessors
    }
    else {
        $system.CPUs = ""
        $system.Cores = $server.NumberOfProcessors
    }
    $system
} |
Format-Table -AutoSize -Wrap
```

Note: You have a CSV file that contains a list of computernames. This file is read using Import-Cs and the results are piped into foreach (an alias of ForEach-Object). The easiest way to present the final data is in a table, so you need to create an object to hold the results. One method of creating such an object is to pipe an empty string, " ", into a select statement with the names of the properties you want the object to have. Note that this only works for properties that are strings. There are other ways of creating objects.

This script could be enhanced in a number of ways:

Ping the server to ensure it's reachable

Add further information, such as disks, installed applications, hotfixes, and page file configuration

Output the data to a file that could become the basis of your server documentation

Add the data directly into a CMDB for configuration management

1.7 Summary

2 Using PowerShell

PowerShell cmdlets

PowerShell and .NET

PowerShell scripting language

Creating PowerShell code for reuse

2.1 PowerShell in a nutshell

You get:

A shell

A set of command-line tools (cmdlets)

A scripting language

An automation engine that allows for remote access, asynchronous processing, and integration between products

You can do:

Run PowerShell commands

Run the standard Windows utilities, such as ipconfig or ping

Work with the filesystem using standard commands

Run Windows batch files (with some provisos around environmental variables)

Run VBScripts

2.2 Cmdlets

eg: to get an standard set of verbs in PowerShell v2

Get-Verb (a built-in function)

or

Get-Command -CommandType cmdlet | group verb | sort count -Descending | select name -First 20
Note: This command groups the results by the verb in the cmdlet name and sorts the verbs on the number of occurrences. The top 20 verbs are then displayed.

2.2.1 Utility cmdlets
eg: Utility cmdlets, aliases, and purposes
Compare-Object (compare, diff) : compares two sets of objects.
ForEach-Object (foreach, %) : performs an operation against each member of a set of input objects.
Group-Object (group) : groups objects that contain the same value for specified properties.
Measure-Object (measure) : calculates the numeric properties of objects, and the characters, words, and lines in string objects, such as files of text.
New-Object : Creats an instance of a Microsoft .NET Framework or COM object.
Select-Object (select) : selects specified properties of an object or set of objects. It can also select unique objects from an array of objects, or it can select a specified number of objects from the beginning or end of an array of objects.
Sort-Object (sort) : sorts objects by property values.
Tee-Object (tee) : saves command output in a file or variable and displays it in the console. This functions exactly like a T junction on a road. The stem of the T is the pipeline. When it reaches t he top, it splits into two and object is duplicated and sent to the variable or file in one drection and along the pipeline in the other.
Where-Object (where,?) : creates a filter that controls which objects will be passed along a command pipeline.

2.2.2 Where-Object
eg: to display the services running on a system
Get-WmiObject -Class Win32_Service
eg: to trim it down using Select-Object
Get-WmiObject -Class Win32_Service | select name, startmode, state
eg: to sort on the service state to put all of the running and all of the stopped services together
Get-WmiObject -Class Win32_Service | select name, statmode, state | where {$_.state -eq "stopped"}
or
Get-WmiObject -Class Win32_Service | where {$_.state -eq "stopped"} | select name, startmode, state
Note: The code inside the braces, {}, is known as a script block.
eg:
$computername = "."
Get-WmiObject -Class Win32_Service -ComputerName $computername |
where {$_.state -eq "stopped" -and $_.startmode -eq "auto"} |
select name, startmode, state
Note: The first line defines a variable to hold the computer name.

2.2.3 Foreach-Object
eg:
$computername = "."
Get-WmiObject -Class Win32_Service -ComputerName $computername |
where {$_.state -eq "stopped" -and $_.startmode -eq "auto"} |
foreach {$_.StartService()}

2.2.4 Aliases
eg: to get a list of currently defined aliases
Get-Alias
or
Get-Alias | where {$_.definition -like "*object"} |
Format-Table Name, Definition -AutoSize
eg: to get other cmdlets that are available for working with aliases.
Get-Command *alias | select name
Note: The import and export commands are for reading and writing the alias information to a file so you can reuse it in other PowerShell sessions.
eg: to create an alias
New-Alias -Name filter -Value Where-Object
or
Set-Alias -Name sieve -Value Where-Object

eg: use your new alias to check if it works

```
Get-Alias | filter {_.definition -like "*object"} | Format-table Name, Definition -AutoSize
```

Note: The defined aliases in a PowerShell session are exposed as the alias:drive

eg: to view the aliases

```
"filter","sieve" | foreach {dir alias:\$_}
```

eg: to delete the aliases

```
"filter","sieve" | foreach {Remove-Item alias:\$_}
```

eg: to find the list of installed PowerShell drives

```
Get-PSDrive
```

eg: for more infomation

```
Get-Help about_Providers
Get-Help Get-PSdrive
```

2.3 Pipeline

eg:

```
Get-Command -CommandType cmdlet | group verb | sort count -Descending | select name -First 20
```

2.4 .NET for administrators

2.4.1 Objects

an object is a package that contains both data and the information on how to use that data.

2.4.2 PowerShell objects

PowerShell puts a wrapper around .NET objects.

2.4.3 Creating .NET Objects

eg:

```
$rand = New-Object -TypeName System.Random -ArgumentList 42
$ran.Next()
```

2.4.4 Creating your own objects

1) Use the select method

eg:

```
$myobject = " " | select name, number, description
$myobject.Name = "Object1"
$myobject-Number = 100
$myobject.Description = "Simplest object creation"
```

Note: The drawback to this method is that the resultant object is a string, as are all of its properties. Technically, it's a Selected.System.String.

2) Use New-Object and the PSObject class

eg:

```
$myobject = New-Object System.Management.Automation.PSObject |
Add-Member -MemberType NoteProperty -Name "Name" -Value "object2" -PassThru |
Add-Member -MemberType NoteProperty -Name "Number" -Value 100 -PassThru |
Add -Member -MemberType NoteProperty -Name "Description" -Value "More complicated" -PassThru
```

Note: This approach uses System.Management.Automation.PSObject to create an object.

or

```
$myobject = New-Object PSObject -Property @{
    Name = "object2a"
    Number = 100
    Description = "More complicated"
}
```

3) Create a new .NET class using c#

```
$source = @"
public class pawobject
{
    public string Description {get;set;}
    public string Name {get;set;}
```

```
    public int Number {get;set;}
}
"@
Add-Type $source -Language CSharpVersion3
$myobject = New-Object -TypeName pawobject -Property @{
    Name = "myobject3";
    Number = 200;
    Description = "More complicated again"
    }
```

Note: The herestring starts with @" and ends with "@.

2.5 PowerShell scripting language

2.5.1 Loops

FOREACH

eg:

```
$lower = "a","b","c","d"
foreach ($letter in $lower) {Write-Host $letter.ToUpper() }
```

or

```
"a","b","c","d" | foreach {Write-Host $_.ToUpper() }
```

FOR LOOP

eg:

```
for ($i=65; $i -le 68; $i++) {Write-Host $([char]$i) }
```

or

```
65..68 | foreach {Write-Host $([char]$_)}
```

Note: The.. or range operation is used to define a range of numbers.

OTHER LOOPS

WHILE LOOP

DO LOOP

eg: for more details

```
Get-Help about_While
Get-Help about_Do
```

2.5.2 Branching

IF STATEMENT

eg:

```
$date = Get-Date
if ($date.DayOfWeek -eq "Friday") {"The weekend starts tonight"}
elseif ($date.DayOfWeek -eq "Saturday" -or $date.DayOfWeek -eq "Sunday") {"It's the weekend"}
else {"Still working!"}
```

SWITCH STATEMENT

eg:

```
switch ( (Get-Date).DayOfWeek ) {
    "Sunday" {"It's the weekend but work tomorrow"; break}
    "Monday" {"Back to work": break}
    "Tuesday" {"Long time until Friday"; break}
    "Wednesday" {"Half way through the week"; break}
    "Thursday" {"Friday tomorrow"; break}
    "Friday" {"It's the weekend tomorrow"; break}
    "Saturday" {"It's the weekend"; break}
    default {"Something's gone wrong"}
}
```

2.5.3 Input and output

1) Writing to the screen and accepting data typed in response to a prompt

2) Writing to or reading from a file on disk

3) Writing to or reading from a specialized data store, such as Active Directory, the registry, or SQL Server.

eg: common cmdlets that provide input and output functionality

1)
Write-Host
Read-Host
Out-Host
Out-GridView
Write-Output
2)
Out-File
Export-Csv
Import-Csv
ConvertFrom-Csv
ConvertTo-Csv
Add-Content
Clear-Content
Get-Content
Set-Content
3)
Write-Debug
Write-Error
Write-EventLog
Write-Progress
Write-Verbose
Write-Warning

2.6 Finding help
Get-Help
Get-Command
Get-Member
2.6.1 Get-Help
eg: to view the function definition
(Get-Item -Path function:\help).Definition
eg:
Get-Help Get-Command
eg: to get the full information on using the help system.
Get-Help Get-Help -Full
eg: to get help online
Get-Help Get-Command -Online
eg: to get conceptual information about PowerShell.
Get-Help about*

2.6.2 Get-Command
eg:
Get-Command ipconfig | f1 *
eg:
Get-Command *wmi* -CommandType cmdlet | select name
eg:
Get-Command -Module BitsTransfer

2.6.3 Get-Member
eg:
Get-Process pwoershell | Get-Member

2.6.4 PowerShell community

2.7 Code reuse
eg: to get the PowerShell execution policy
Get-ExecutionPolicy
eg: to enable scripts to run from the local drive but expects scripts on remote drives to be signed with a recognized code-

signing certificate.
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned

2.7.1 Editors
Notepad
PowerShell ISE
PowerGUI Script Editor
PowerShell Plus

2.7.2 Scripts
eg: to discover what disks are installed in your computers

```
param (
    [string]$computername = "localhost"
)
    Get-WmiObject -Class Win32_DiskDrive -ComputerName $computername |
    Format-List DeviceId, Status, Index, InterfaceType, Partitions, BytesPerSector, SectorsPerTrack, TracksPerCylinder,
TotalHeads, TotalCylinders, TotalTracks, TotalSectors,
    @{Name="Disk Size (GB)"; Expression={"{0:F3}" -f $($_.Size/1GB)}}
```

.\Get-DiskInfo.ps1
or
.\Get-DiskInfo.ps1 -computername "rslaptop01"

2.7.3 Functions
eg: function to investigate physical disks

```
function get-disk {
param (
    [string] $computername = "$env:COMPUTERNAME"
}
Get-WmiObject -Class Win32_DiskDrive -ComputerName $computername |
Format-List DeviceID, Status, Index, InterfaceType, Partitions, BytesPerSector, SectorsPerTrack, TracksPerCylinder, TotalHeads,
TotalCylinders, TotalTracks, TotalSectors, @{Name="Disk Size (GB)"; Expression={"{0:F3}" -f $($_.Size/1GB)}}
}
Import-Csv computers2.csv | foreach {get-disk $_.computer}
```

eg: Advanced function to investigate physical disks

```
function get-disk {
[CmdletBinding()]
param (
    [Parameter (ValueFromPipelineByPropertyName=$true)]
    [string]
    [ValidNotNullOrEmpty()]
    $computername
)
process {
    Write-Debug $computername
    Get-WmiObject -Class Win32_DiskDrive -ComputerName $computername | Format-List DeviceID, Status, Index, InterfaceType,
Partitions, BytesPerSector, SectorsPerTrack, TracksPerCylinder, TotalHeads, TotalClyinders, TotalTracks, TotalSectors,
    @{Name="Disk Size (GB) "; Expression={"0:F3}" -f $($_.Size/1GB)}}
}
}
```

2.7.4 Modules
eg: to discover the modules installed in your system
Get-Module -ListAvailable
eg: to get the PSModulePath environment variable value, which stores the locations of Modules.
$env:psmodulepath -split ";"
eg: to import the module
Import-Module BitsTransfer

eg: to discover the functions or cmdlets it has loaded
Get-Command -Module BitsTransfer | select name
eg: to remove the module
Remove-Module BitsTransfer

2.8 PowerShell remoting
2.8.1 Remoting by cmdlet
eg: to discover the full list of cmdlets that can work with a -ComputerName parameter to enable access to remote machines
Get-Help * -Parameter computername | Format-Wide -Column 3
eg: to enable WMI remote access by configuring the remote firewall from PowerShell or a command prompt
Netsh firewall set service RemoteAdmin
Netsh advfirewall set currentprofile settings remotemanagement enable
eg:
Get-Service winrm
eg: to check on remote machines
Get-Service -Name winrm -ComputerName -w08r2sq108
Get-Service -Name winrm -ComputerName -w08r2sq108a
Get-Service -Name winrm -ComputerName -w08r2sq108, w08r2sq108a
or
$computers = @("localhost","w08r2sq108","w08r2sq108a")
Get-WmiObject -Class Win32_Service -Filter "Name='winrm'" -ComputerName $computername | ft SystemName, State, StarMode
Note: This gives you a quick way to test whether the required services are running on remote machines.

2.8.2 PowerShell remote sessions
requirements for PowerShell remoting to work:
The WinRM service must be installed on the remote system.
PowerShell v2 must be installed on both systems.
The Enable-PSRemoting cmdlet must have been run on remote machines with elevated privileges to configure WinRM, the firewall, and other necessary elements.
eg: to create a connection to one or more remote systems
$s = New-PSSession -ComputerName W08R2SQL08, W08R2SQL08A
eg: to get available sessions
Get-PSSession
eg:
Invoke-Command -Session $s -ScriptBlock {Get-Service winrm}
eg:
$sa = Get-PSSession -Id 2
Invoke-Command -Session $sa -ScriptBlock {Get-Service sql*}
eg: to remove the remote session
Get-PSSession | Remove-PSSession

2.9 PowerShell jobs
eg: Cmdlets for working with PowerShell jobs
Invoke-Command
Get-WmiObject
Invoke-WmiMethod
Remove-WmiObject
Set-WmiInstance
Test-Connection
Restart-Computer
Stop-Computer
Get-Job
Receive-Job
Remove-Job
Start-Job
Stop-Job
Wait-Job
eg: to create a job on the local machine

Start-Job -Name PaW1 -ScriptBlock {
Get-WmiObject -Class Win32_Service -Filter "Name='winrm'"}
eg: to create a job to run against remote machines
Invoke-Command -ComputerName w08r2sqp08, w082sql08a -ScriptBlock {
Get-WmiObject -Class Win32_Service -Filter "Name='winrm'"} -AsJob
eg: to access data held in the jobs
Receive-Job -Name Paw1 -Keep (get the information from the job you created with Start-Job)
or
Receive-Job -Name Job3 -Keep (retrieve data from the job created with Invoke-Command)
or
Get-Job -Id 3 | Receive-Job -Location w08r2sql08a -Keep (access data for one computer when the job has been run against a number of machines.)
eg: a complete and final cleanup
Get-Job | Remove-Job

2.10 Summary

3. WMI in depth

3.1 The structure of WMI
eg: get the number of Managed Object Format (MOF) files.
(Get-ChildItem -Path C:\Windows\System32\wbem -Filter *.mof).count
Note: The MOF files store the definitions of the WMI classes.

3.1.1 Providers
eg: to find the providers
Get-WmiObject -Class __Win32Provider | select name (This lists all of the providers in the namespace that PowerShell uses as a default, root\cimv2.)
eg: WMI provider types
Class
Event
Event consumer
Instance
Method
Property

3.1.2 Namespaces
WMI namespaces are used to logically subdivide the available WMI classes.
the namespace at the top of the hierarchy is called root and your default namespace is root\cimv2.

DISCOVERING NAMESPACES
eg: take a look at the system classes for a namespace root\cimv2
Get-WmiObject -Namespace 'root\cimv2' -List "__*"
eg: investigate the class __NAMESPACE and generate a list of the namespaces in the root\cimv2 namespace
Get-WmiObject -Class __NAMESPACE | select name
eg: Find WMI namespace installed on a system
```
function get-namespace {
param ([string]$name)
    Get-WmiObject -Namespace $name -Class "__NAMESPACE" |
    foreach {
        "$name\" + $_.Name
        get-namespace $("$name\" + $_.Name)
    }
}
"root"
get-namespace "root"
```
Note: The function takes a namespace name as its only parameter. It then uses that name to retrieve the namespaces contained

within that namespace using Get-WmiObject. The results are piped into a ForEach-Object cmdlet that displays the name of the current namespace and the discovered namespaces using a backslash (\) as a driver. This process of getting a function to call itself is called recursion.

eg: Find WMI providers in each namespace

```
function get-namespace {
param ([string]$name)
    Get-WmiObject -Namespace $name -Class "__NAMESPACE" |
    foreach {
        $ns = "$name\" + $_.Name
        "`nNameSpace: $ns"
        "providers:"
        Get-WmiObject -NameSpace $ns -Class __Win32Provider |
        select name
        get-namespace $("$name\" + $_.Name)
    }
}
"root"
get-namespace "root"
```

REGISTRATIONS

eg: to find a full list of the classes involved in dealing with registrations by using the -List parameter

```
Get-WmiObject -Namespace 'root\cimv2' -List "__*Registration*"
```

or

```
Get-WmiObject -Namespace 'root\cimv2' -Class __ProviderRegistration
```

eg: to see what the registration classes can tell you, look at one of each type of registration class

```
Get-WmiObject -Namespace 'root\cimv2' -List "__*Registration*" |
foreach {
    Get-WmiObject -Namespace 'root\cimv2' -Class $($_.Name) |
    select -First 1
}
```

eg: to look at which providers perform which type of registration

```
Get-WmiObject -Namespace 'root\cimv2' -List "__*Registration*" |
foreach {
    Get-WmiObject -Namespace 'root\cimv2' -Class $($_.Name) |
    Format-Table __CLASS, provider -AutoSize
}
```

eg: to look at the __EventProviderRegistration class

```
Get-WmiObject -Namespace 'root\cimv2' -Class __EventProviderRegistration |
foreach {
    "`n"
    $_ | Format-Table EventQueryList, provider -Wrap
}
```

3.1.3 Classes

A WMI class represents a specific item in your system. It could be a piece of hardware, software, event or a data store.

eg: Three class categories

Core classes: represent managed objects that apply to all areas of management.

Common classes: represent managed objects that applyto specific management areas (these classes are prefixed with CIM_).

Extended classes: represents managed objects that are technology-specific-- that is, classes that are created to manage the Windows platform and Microsoft applications.

eg: WMI class types

System class (can be static or abstract)

Abstract

Static

Dynamic

Association

SYSTEM CLASSES

eg: to find the full list for the root\cimv2 namespace by using the -List parameter in Get-WmiObject

```
Get-WmiObject -Namespace 'root\cimv2' -List "__*"
```

eg: too see that the same system classes are present in each namespace by comparing two namespaces at random

```
$cimv2 = Get-WmiObject -Namespace 'root\cimv2' -List "__*" | select name
$scent = Get-WmiObject -Namespace 'root\SecurityCenter' -List "__*" | select name
Compare-Object -ReferenceObject $cimv2 -DifferenceObject $scent -IncludeEqual
```

Note: The proceding code generate a list of system classes for each namespace. The Compare-Object cmdlet is used to compare the two objects that are produced. If you don't use the -IncludeEqual parameter, you'll get nothing back, which indicates that the two objects are identical.

CLASSES BY PROVIDER

eg: Find WMI classes installed by a provider

```
$namespace = "root\cimv2"
Get-WmiObject -Namespace $namespace -Class __Win32Provider |
foreach {
    $provider = $_.Name
    "Provider: $provider"
    $refs = Get-WmiObject -Namespace $namespace `
    -Query "REFERENCES OF {__Win32Provider.Name='$provider'}"
    foreach ($ref in $refs) {
        $type = $ref.__CLASS
        " Registration: $type"
        switch ($type) {
            "__PropertyProviderRegistration" {
                " does not have classes"
                break
            }
            "__ClassProviderRegistration" {
                " only provides class definitions"
                break
            }
            "__EventConsumerProviderRegistration" {
                " uses these classes"
                "  $($ref.ConsumerClassNames)"
                break
            }
            "__EventProviderRegistration" {
                "  queries these classes:"
                foreach ($query in $ref.EventQueryList) {
                    $a = $query -split " "
                    "  $($a[($a.length-1(])"
                }
                break
            }
            defualt {
                "  supplies these classes:"
                Get-WmiObject -Namespace $namespace -List -Amended |
                foreach {
                    if (_.Qualifiers["provider"].Value -eq "$provider") {
                        $(_.Name)"
                    }
                }
                break
            }
        } #switch
    } #refs
    } #provider loop
```

```
    CLASSES IN A NAMESPACE
    eg:
    Get-WmiObject -List
    eg: use the Get-Help to look at the parameter definition
    Get-Help Get-WmiObject -Parameter list
    Note: the Get-CIMClass cmdlet does a similar job.
    eg: to add a filter to restrict the number of classes returned
    Get-WmiObject -List *network*
    or
    Get-WmiObject -List | where {$_.Name -like "*network*"}
    Note: the upper one doesn't work in PowerShell v1.
    eg: to change the namespace to be interrogated by using the -Namespace parameter
    Get-WmiObject -Namespace "root\Security" -List

    QUALIFIERS
    eg:
    $q = Get-WmiObject -List Win32_Process -Amended
    $q.Qulifiers | Format-Table Name, Value -AutoSize -Wrap
    eg: to display the description of each class
    Get-WmiObject -List Win32*networkadapter* |
    foreach {
        "`n$($_.Name)"
        ((Get-WmiObject -List $($_.Name) -Amended).Qualifiers |
        where {$_.Name -eq "Description"}).Value
    }
```

3.2 Methods and properties

3.2.1 Methods

```
eg:
Get-WmiObject -List *process*
eg:
Get-WmiObject -Class Win32_Process | select -First 1 | Get-Member
eg:
Get-Help Get-Member -Parameter View
eg:
Get-WmiObject -Class Win32_Process | select -First 1 | Get-Member -View all
```

3.2.2 Class properties

```
eg:
Get-WmiObject -Class Win32_OperatingSystem
eg:
Get-WmiObject -Class Win32_OperatingSystem | Format-List *
eg:
Get-ChildItem -Path $pshome -filter *.format.ps1xml
```

3.2.3 System properties

```
eg:
Get-WmiObject -Class Win32_OperatingSystem | select -ExcludeProperty __* -Property *
```

3.2.4 key properties

```
eg: Discover the key property of a WMI class
function get-key {
[CmdletBinding()]
param (
    [string]
    [ValidateNortNullOrEmpty()]
    $class
```

```
)
    $t = [WMIClass] $class
    $t.properties | select @{Name="PName";Expression={$_.name}} -ExpandProperty Qualifiers |
    where {$_.Name -eq "key"} |
    foreach {"The key for the $class class is $($_.Pname)"}
}
```

3.3 Documenting WMI

eg: Document WMI on a machine

```
function main {
    "root"
    get-namespace "root"
}
function get-namespace {
param ([string]$name)
    Get-WmiObject -Namespace $name -Class "__NAMESPACE" |
    foreach {
        $ns = "$name\" + $_.Name
        "`nNameSpace:$ns"
        get-providerclass $ns
        get-namespace $ns
    }
}
function get-providerclass {
param ([string] $namespace)
    Get-WmiObject -Namespace $namespace -Class __Win32Provider |
    foreach {
        $provider = $_.Name
        "Provider: $provider"
        $refs = Get-WmiObject -Namespace $namespace -Query "REFERENCES OF
            {__Win32Provider.Name='$provider'}"
        foreach ($ref in $refs) {
            $type = $ref.__CLASS
            " Registration: $type"
            switch ($type) {
                "__PropertyProviderRegistration" {
                    " does not have classes"
                    break
                }
                "__ClassProviderRegistration" {
                    " only provides class definitions"
                    break
                }
                "__EventConsumerProviderRegistration" {
                    " uses these classes"
                    " $($ref.ConsumerClassNames)"
                    break
                }
                "__EventProviderRegistration" {
                    " queries these classes:"
                    foreach ($query in $ref.EventQueryList) {
                    $a = $query -split " "
                    " $($a[($a.length-1)])"
                    }
                    break
                }
                default {
                    " supplies these classes:"
```

```
                    Get-WmiObject -Namespace $namespace -List -Amended |
                    foreach {
                        if ($_.Qualifiers["provider"].Value -eq "$provider"){
                        " $($_.Name)"
                        }
                    } # class list
                    break
                }
            } #switch
        } # refs
    } # provider loop
}
main
eg:
./Get-WmiList | Out-File wmidoc.txt
```

3.4 WMI cmdlets and accelerators

3.4.1 Cmelets

```
Get-WmiObject
Set-WmiInstance
eg:
Get-WmiObject -Class Win32_NTEventLogFile -Filter "LogFileName='Scripts'" | Get-Member
eg:
$log = Get-WmiObject -Class Win32_NTEventLogFile -Filter "LogFileName='Scripts'"
Set-WmiInstance -InputObject $log -Arguments @{MaxFileSize=31457280}

Invoke-WmiMethod
eg:
$log = Get-WmiObject -Class Win32_NTEventLogFile -Filter "LogFileName='Scripts'"
$log | Get=Member -MemberType method
eg:
Invoke-WmiMethod -InputObject $log -Name BackupEventLog -ArgumentList "c:\test\paw3.evt
eg:
Get-WmiObject -Class Win32_NTEventLogFile -Filter "LogFileName='Scripts'" |
Invoke -WmiMethod -Name BackupEvenlog -ArgumentList "c:\test\paw4.evt"

Remove-WmiObject
eg:
Get-WmiObject -Class Win32_Process | where {$_.Name -like "Notepad*"}
eg:
Get-WmiObject -Class Win32_Process | where {_.Name -like "Notepad*"} |
Remove-WmiObject
```

3.4.2 Type accelerators

```
eg:
Get-WmiObject -Class Win32_Process -Filter "Name='powershell.ext'"
Get-WmiObject -Query "SELECT * FROM Win32_Process WHERE Name='powershell.exe'"
eg:
$query = [wmisearcher] "SELECT * FROM Win32_Process WHERE Name='powershell.exe'"
$query.Get()
egL
Get-WmiObject -Class Win32_Process -Filter "Name='calc.exe'"
```

3.5 Using WQL

3.5.1 Keywords

```
eg:
```

```
Get-WmiObject -Query "SELECT * FROM Win32_Process WHERE Name='PowerShell.exe'"
eg:
Get-WmiObject -Query "SELEC Name, Threadcount, UserModetime FROM Win32_Process WHERE Name='PowerShell.exe'"
eg:
Get-WmiObject -Class Win32_Process -Filter "Name='PowerShell.exe'"
eg:
Get-WmiObject -Class Win32_Process -Filter "Name='PowerShell.exe'" |
Format-List Name, Threadcount, UserModetime
```

3.5.2 Operations