**WMI: Windows Management Instrumentation**

[More Sharing ServicesShare](http://www.addthis.com/bookmark.php?v=250&pub=xa-4a7a64957fb03b5c) | [Share on twitterShare on facebook](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/25/chapter-18-wmi-windows-management-instrumentation.aspx)[Share on myspace](http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=xa-4a7a64957fb03b5c&source=tbx-300&lng=en-AU&s=myspace&url=http%3A%2F%2Fpowershell.com%2Fcs%2Fblogs%2Febookv2%2Farchive%2F2012%2F03%2F25%2Fchapter-18-wmi-windows-management-instrumentation.aspx&title=Chapter%2018.%20WMI%3A%20Windows%20Management%20Instrumentation%20-%20Master-PowerShell%20%7C%20With%20Dr.%20Tobias%20Weltner%20-%20PowerShell.com%20%E2%80%93%20PowerShell%20Scripts%2C%20Tips%2C%20Forums%2C%20and%20Resources&ate=AT-xa-4a7a64957fb03b5c/-/-/55114223cf597cb4/1&frommenu=1&uid=551142235be4522c&ct=1&pre=http%3A%2F%2Fwww.google.com.au%2Furl%3Fsa%3Dt%26rct%3Dj%26q%3D%26esrc%3Ds%26source%3Dweb%26cd%3D5%26ved%3D0CDcQFjAE%26url%3Dhttp%253A%252F%252Fpowershell.com%252Fcs%252Fblogs%252Febookv2%252Farchive%252F2012%252F03%252F25%252Fchapter-18-wmi-windows-management-instrumentation.aspx%26ei%3Dm0ARVbiyJYbm8AWO84DQCw%26usg%3DAFQjCNG9OURd-Hi52gMEtvp4kbS8ymrtUQ%26sig2%3Dht71CzjD0MpH9OK712g0Uw&tt=0&captcha_provider=recaptcha)[Share on blogger](http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=xa-4a7a64957fb03b5c&source=tbx-300&lng=en-AU&s=blogger&url=http%3A%2F%2Fpowershell.com%2Fcs%2Fblogs%2Febookv2%2Farchive%2F2012%2F03%2F25%2Fchapter-18-wmi-windows-management-instrumentation.aspx&title=Chapter%2018.%20WMI%3A%20Windows%20Management%20Instrumentation%20-%20Master-PowerShell%20%7C%20With%20Dr.%20Tobias%20Weltner%20-%20PowerShell.com%20%E2%80%93%20PowerShell%20Scripts%2C%20Tips%2C%20Forums%2C%20and%20Resources&ate=AT-xa-4a7a64957fb03b5c/-/-/55114223cf597cb4/2&frommenu=1&uid=5511422360e84bbf&ct=1&pre=http%3A%2F%2Fwww.google.com.au%2Furl%3Fsa%3Dt%26rct%3Dj%26q%3D%26esrc%3Ds%26source%3Dweb%26cd%3D5%26ved%3D0CDcQFjAE%26url%3Dhttp%253A%252F%252Fpowershell.com%252Fcs%252Fblogs%252Febookv2%252Farchive%252F2012%252F03%252F25%252Fchapter-18-wmi-windows-management-instrumentation.aspx%26ei%3Dm0ARVbiyJYbm8AWO84DQCw%26usg%3DAFQjCNG9OURd-Hi52gMEtvp4kbS8ymrtUQ%26sig2%3Dht71CzjD0MpH9OK712g0Uw&tt=0&captcha_provider=recaptcha)[Share on live](http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=xa-4a7a64957fb03b5c&source=tbx-300&lng=en-AU&s=live&url=http%3A%2F%2Fpowershell.com%2Fcs%2Fblogs%2Febookv2%2Farchive%2F2012%2F03%2F25%2Fchapter-18-wmi-windows-management-instrumentation.aspx&title=Chapter%2018.%20WMI%3A%20Windows%20Management%20Instrumentation%20-%20Master-PowerShell%20%7C%20With%20Dr.%20Tobias%20Weltner%20-%20PowerShell.com%20%E2%80%93%20PowerShell%20Scripts%2C%20Tips%2C%20Forums%2C%20and%20Resources&ate=AT-xa-4a7a64957fb03b5c/-/-/55114223cf597cb4/3&frommenu=1&uid=55114223294ae94f&ct=1&pre=http%3A%2F%2Fwww.google.com.au%2Furl%3Fsa%3Dt%26rct%3Dj%26q%3D%26esrc%3Ds%26source%3Dweb%26cd%3D5%26ved%3D0CDcQFjAE%26url%3Dhttp%253A%252F%252Fpowershell.com%252Fcs%252Fblogs%252Febookv2%252Farchive%252F2012%252F03%252F25%252Fchapter-18-wmi-windows-management-instrumentation.aspx%26ei%3Dm0ARVbiyJYbm8AWO84DQCw%26usg%3DAFQjCNG9OURd-Hi52gMEtvp4kbS8ymrtUQ%26sig2%3Dht71CzjD0MpH9OK712g0Uw&tt=0&captcha_provider=recaptcha)[Share on google](http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=xa-4a7a64957fb03b5c&source=tbx-300&lng=en-AU&s=google&url=http%3A%2F%2Fpowershell.com%2Fcs%2Fblogs%2Febookv2%2Farchive%2F2012%2F03%2F25%2Fchapter-18-wmi-windows-management-instrumentation.aspx&title=Chapter%2018.%20WMI%3A%20Windows%20Management%20Instrumentation%20-%20Master-PowerShell%20%7C%20With%20Dr.%20Tobias%20Weltner%20-%20PowerShell.com%20%E2%80%93%20PowerShell%20Scripts%2C%20Tips%2C%20Forums%2C%20and%20Resources&ate=AT-xa-4a7a64957fb03b5c/-/-/55114223cf597cb4/4&frommenu=1&uid=55114223b25763e9&ct=1&pre=http%3A%2F%2Fwww.google.com.au%2Furl%3Fsa%3Dt%26rct%3Dj%26q%3D%26esrc%3Ds%26source%3Dweb%26cd%3D5%26ved%3D0CDcQFjAE%26url%3Dhttp%253A%252F%252Fpowershell.com%252Fcs%252Fblogs%252Febookv2%252Farchive%252F2012%252F03%252F25%252Fchapter-18-wmi-windows-management-instrumentation.aspx%26ei%3Dm0ARVbiyJYbm8AWO84DQCw%26usg%3DAFQjCNG9OURd-Hi52gMEtvp4kbS8ymrtUQ%26sig2%3Dht71CzjD0MpH9OK712g0Uw&tt=0&captcha_provider=recaptcha)[Share on email](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/25/chapter-18-wmi-windows-management-instrumentation.aspx)

.

**Table of Contents**

* [Chapter 1. The PowerShell Console](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/08/chapter-1-the-powershell-console.aspx)
* [Chapter 2. Interactive PowerShell](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/09/interactive-powershell.aspx)
* [Chapter 3. Variables](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/10/chapter-3-variables.aspx)
* [Chapter 4. Arrays and Hashtables](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/11/chapter-4-arrays-and-hashtables.aspx)
* [Chapter 5. The PowerShell Pipeline](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/12/chapter-5-the-powershell-pipeline.aspx)
* [Chapter 6. Working with Objects](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/13/chapter-6-working-with-objects.aspx)
* [Chapter 7. Conditions](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/14/chapter-7-conditions.aspx)
* [Chapter 8. Loops](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/15/chapter-8-loops.aspx)
* [Chapter 9. Functions](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/16/chapter-9-functions.aspx)
* [Chapter 10. Scripts](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/17/chapter-10-scripts.aspx)
* [Chapter 11. Error Handling](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/18/chapter-11-error-handling.aspx)
* [Chapter 12. Managing Scope](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/19/chapter-12-command-discovery-and-scriptblocks.aspx)
* [Chapter 13. Text and Regular Expressions](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/20/chapter-13-text-and-regular-expressions.aspx)
* [Chapter 14. XML](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/21/chapter-14-xml.aspx)
* [Chapter 15. Working with the File System](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/22/chapter-15-working-with-the-file-system.aspx)
* [Chapter 16. Managing Windows Registry](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/23/chapter-16-managing-windows-registry.aspx)
* [Chapter 17. Processes, Services, and Event Logs](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/24/chapter-17-processes-services-and-event-logs.aspx)
* [Chapter 18. WMI: Windows Management Instrumentation](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/25/chapter-18-wmi-windows-management-instrumentation.aspx)
* [Chapter 19. User Management](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/25/chapter-19-user-management.aspx)
* [Chapter 20. Loading .NET Libraries and Compiling Code](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/27/chapter-20-loading-net-libraries-and-compiling-code.aspx)

Windows Management Instrumentation (WMI) is a technique available on all Windows systems starting with Windows 2000. WMI can provide you with a wealth of information about the Windows configuration and setup. It works both locally and remotely, and PowerShell makes accessing WMI a snap.

* [WMI Quick Start](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#wmi-quick-start)
* [Retrieving Information](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#retrieving-information)
  + [Exploring WMI Classes](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#exploring-wmi-classes)
  + [Swallowing The Red Pill](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#swallowing-the-red-pill)
  + [Filtering WMI Results](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#filtering-wmi-results)
  + [Direct WMI Object Classes](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#direct-wmi-object-access)
* [Changing System Configuration](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#changing-system-configuration)
  + [Modifying Properties](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#modifying-properties)
  + [Invoking WMI Methods](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#invoking-wmi-methods)
  + [Static Methods](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#static-methods)
  + [Using WMI Auto-Documentation](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#using-wmi-auto-documentation)
* [WMI Events](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#wmi-events)
* [Using WMI Remotely](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#using-wmi-remotely)
  + [Accessing WMI Objects on Another Computer](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#accessing-wmi-objects-on-another-computer)
* [WMI Background Information](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#wmi-background-information)
  + [Converting the WMI Date Format](http://powershell.com/cs/blogs/ebookv2/archive/2012/03/07/chapter-18-wmi-windows-management-instrumentation.aspx#converting-the-wmi-date-format)

**WMI Quick Start**

To work with WMI you need to know just a little bit of theory. Let's check out what the terms "class" and "object" stand for.

A "class" pretty much is like the "kind of an animal". There are dogs, cats, horses, and each kind is a class. So there is always only one class of a kind.

An "object" works like an "animal", so there are zillions of real dogs, cats, and horses. So, there may be one, ten, thousands, or no objects (or "instances") of a class. Let's take the class "mammoth". There are no instances of this class these days.

WMI works the same. If you'd like to know something about a computer, you ask WMI about a class, and WMI returns the objects. When you ask for the class "Win32\_BIOS", you get back exactly one instance (or object) because your computer has just one BIOS. When you ask for "Win32\_Share", you get back a number of instances, one for each share. And when you ask for "Win32\_TapeDrive", you get back nothing because most likely, your computer has no built-in tape drive. Tape drives thus work like mammoths in the real world. While there is a class ("kind"), there is no more instance.

**Retrieving Information**

How do you ask WMI for objects? It's easy! Just use the cmdlet Get-WmiObject. It accepts a class name and returns objects, just like the cmdlet name and its parameter suggest:

PS> Get-WmiObject -Class Win32\_BIOS

SMBIOSBIOSVersion : RKYWSF21

Manufacturer : Phoenix Technologies LTD

Name : Phoenix TrustedCore(tm) NB Release SP1 1.0

SerialNumber : 701KIXB007922

Version : PTLTD - 6040000

**Exploring WMI Classes**

As you can see, working with WMI does not require much knowledge. It does require though that you know the name of a WMI class that represents what you are after. Fortunately, Get-WmiObject can also work like a dictionary and look up WMI class names for you. This will get you all WMI class names that have the keyword "print" in them:

PS> Get-WmiObject -List Win32\_\*Print\*

NameSpace: ROOT\cimv2

Name Methods Properties

---- ------- ----------

Win32\_PrinterConfiguration {} {BitsPerPel, Captio...

Win32\_PrinterSetting {} {Element, Setting}

Win32\_PrintJob {Pause, Resume} {Caption, Color, Da...

Win32\_Printer {SetPowerState, R... {Attributes, Availa...

Win32\_PrinterDriver {StartService, St... {Caption, ConfigFil...

Win32\_TCPIPPrinterPort {} {ByteCount, Caption...

Win32\_PrinterShare {} {Antecedent, Depend...

Win32\_PrinterDriverDll {} {Antecedent, Depend...

Win32\_PrinterController {} {AccessState, Antec...

**Swallowing The Red Pill**

By default, PowerShell limits the information WMI returns to you so you don't get carried away. It's pretty much like in the movie "The Matrix": you need to decide whether you want to swallow the blue pill and live in a simple world, or whether you dare to swallow the red pill and see the real world. By default, you live in the blue-pill-world with only limited information.

PS> Get-WmiObject -Class Win32\_BIOS

SMBIOSBIOSVersion : 02LV.MP00.20081121.hkk

Manufacturer : Phoenix Technologies Ltd.

Name : Phoenix SecureCore(tm) NB Version 02LV.MP00.20081121.hkk

SerialNumber : ZAMA93HS600210

Version : SECCSD - 6040000

To see the red-pill-world, pipe the results to Select-Object and ask it to show all available properties:

PS> Get-WmiObject -Class Win32\_BIOS | Select-Object -Property \*

Status : OK

Name : Phoenix SecureCore(tm) NB Version

02LV.MP00.20081121.hkk

Caption : Phoenix SecureCore(tm) NB Version

02LV.MP00.20081121.hkk

SMBIOSPresent : True

\_\_GENUS : 2

\_\_CLASS : Win32\_BIOS

\_\_SUPERCLASS : CIM\_BIOSElement

\_\_DYNASTY : CIM\_ManagedSystemElement

\_\_RELPATH : Win32\_BIOS.Name="Phoenix SecureCore(tm) NB Version

02LV.MP00.20081121.hkk",SoftwareElementID="Phoenix

SecureCore(tm) NB Version 02LV.MP00.20081121.hkk",Softw

areElementState=3,TargetOperatingSystem=0,Version="SECC

SD - 6040000"

\_\_PROPERTY\_COUNT : 27

\_\_DERIVATION : {CIM\_BIOSElement, CIM\_SoftwareElement,

CIM\_LogicalElement, CIM\_ManagedSystemElement}

\_\_SERVER : DEMO5

\_\_NAMESPACE : root\cimv2

\_\_PATH : \\DEMO5\root\cimv2:Win32\_BIOS.Name="Phoenix

SecureCore(tm) NB Version

02LV.MP00.20081121.hkk",SoftwareElementID="Phoenix

SecureCore(tm) NB Version 02LV.MP00.20081121.hkk",Softw

areElementState=3,TargetOperatingSystem=0,Version="SECC

SD - 6040000"

BiosCharacteristics : {4, 7, 8, 9...}

BIOSVersion : {SECCSD - 6040000, Phoenix SecureCore(tm) NB Version

02LV.MP00.20081121.hkk, Ver 1.00PARTTBL}

BuildNumber :

CodeSet :

CurrentLanguage :

Description : Phoenix SecureCore(tm) NB Version

02LV.MP00.20081121.hkk

IdentificationCode :

InstallableLanguages :

InstallDate :

LanguageEdition :

ListOfLanguages :

Manufacturer : Phoenix Technologies Ltd.

OtherTargetOS :

PrimaryBIOS : True

ReleaseDate : 20081121000000.000000+000

SerialNumber : ZAMA93HS600210

SMBIOSBIOSVersion : 02LV.MP00.20081121.hkk

SMBIOSMajorVersion : 2

SMBIOSMinorVersion : 5

SoftwareElementID : Phoenix SecureCore(tm) NB Version

02LV.MP00.20081121.hkk

SoftwareElementState : 3

TargetOperatingSystem : 0

Version : SECCSD - 6040000

Scope : System.Management.ManagementScope

Path : \\DEMO5\root\cimv2:Win32\_BIOS.Name="Phoenix

SecureCore(tm) NB Version

02LV.MP00.20081121.hkk",SoftwareElementID="Phoenix

SecureCore(tm) NB Version 02LV.MP00.20081121.hkk",Softw

areElementState=3,TargetOperatingSystem=0,Version="SECC

SD - 6040000"

Options : System.Management.ObjectGetOptions

ClassPath : \\DEMO5\root\cimv2:Win32\_BIOS

Properties : {BiosCharacteristics, BIOSVersion, BuildNumber,

Caption...}

SystemProperties : {\_\_GENUS, \_\_CLASS, \_\_SUPERCLASS, \_\_DYNASTY...}

Qualifiers : {dynamic, Locale, provider, UUID}

Site :

Container :

Once you see the real world, you can pick the properties you find interesting and then put together a custom selection. Note that PowerShell adds a couple of properties to the object which all start with "\_\_". These properties are available on all WMI objects. \_\_Server is especially useful because it always reports the name of the computer system the WMI object came from. Once you start retrieving WMI information remotely, you should always add \_\_Server to the list of selected properties.

PS> Get-WmiObject Win32\_BIOS | Select-Object \_\_Server, Manufacturer, SerialNumber, Version

\_\_SERVER Manufacturer SerialNumber Version

-------- ------------ ------------ -------

DEMO5 Phoenix Technolo... ZAMA93HS600210 SECCSD - 6040000

**Filtering WMI Results**

Often, there are more instances of a class than you need. For example, when you query for Win32\_NetworkAdapter, you get all kinds of network adapters, including virtual adapters and miniports.

PowerShell can filter WMI results client-side using Where-Object. So, to get only objects that have a MACAddress, you could use this line:

PS> Get-WmiObject Win32\_NetworkAdapter | Where-Object { $\_.MACAddress -ne $null } |  
>> Select-Object Name, MACAddress, AdapterType  
>>

Name MACAddress AdapterType

---- ---------- -----------

Intel(R) 82567LM-Gigabi... 00:13:77:B9:F2:64 Ethernet 802.3

RAS Async Adapter 20:41:53:59:4E:FF Wide Area Network (WAN)

Intel(R) WiFi Link 5100... 00:22:FA:D9:E1:50 Ethernet 802.3

Client-side filtering is easy because it really just uses Where-Object to pick out those objects that fulfill a given condition. However, it is slightly inefficient as well. All WMI objects need to travel to your computer first before PowerShell can pick out the ones you want.

If you only expect a small number of objects and/or if you are retrieving objects from a local machine, there is no need to create more efficient code. If however you are using WMI remotely via network and/or have to deal with hundreds or even thousands of objects, you should instead use server-side filters.

These filters are transmitted to WMI along with your query, and WMI only returns the wanted objects in the first place. Since these filters are managed by WMI and not PowerShell, they use WMI syntax and not PowerShell syntax. Have a look:

PS> Get-WmiObject Win32\_NetworkAdapter -Filter 'MACAddress != NULL' |  
>> Select-Object Name, MACAddress, AdapterType  
>>

Name MACAddress AdapterType

---- ---------- -----------

Intel(R) 82567LM-Gigabi... 00:13:77:B9:F2:64 Ethernet 802.3

RAS Async Adapter 20:41:53:59:4E:FF Wide Area Network (WAN)

Intel(R) WiFi Link 5100... 00:22:FA:D9:E1:50 Ethernet 802.3

Simple filters like the one above are almost self-explanatory. WMI uses different operators ("!=" instead of "-ne" for inequality) and keywords ("NULL" instead of $null), but the general logic is the same.

Sometimes, however, WMI filters can be tricky. For example, to find all network cards that have an IP address assigned to them, in PowerShell (using client-side filtering) you would use:

PS> Get-WmiObject Win32\_NetworkAdapterConfiguration | Where-Object { $\_.IPAddress -ne $null } |  
>> Select-Object Caption, IPAddress, MACAddress  
>>

Caption IPAddress MACAddress

------- --------- ----------

[00000011] Intel(R) WiF... {192.168.2.109, fe80::a... 00:22:FA:D9:E1:50

If you translated this to a server-side WMI filter, it fails:

PS> Get-WmiObject Win32\_NetworkAdapterConfiguration -Filter 'IPAddress != NULL' |  
>> Select-Object Caption, IPAddress, MACAddress  
>>

Get-WmiObject : Invalid query "select \* from Win32\_NetworkAdapterConfiguration where IPAddress != NULL"

The reason for this is the nature of the *IPAddress* property. When you look at the results from your client-side filtering, you'll notice that the column *IPAddress* has values in braces and displays more than one *IP address*. The property IPAddress is an array. WMI filters cannot check array contents.

So in this scenario, you would have to either stick to client-side filtering or search for another object property that is not an array and could still separate network cards with IP address from those without. There happens to be a property called *IPEnabled* that does just that:

PS> Get-WmiObject Win32\_NetworkAdapterConfiguration -Filter 'IPEnabled = true' |  
>> Select-Object Caption, IPAddress, MACAddress  
>>

Caption IPAddress MACAddress

------- --------- ----------

[00000011] Intel(R) WiF... {192.168.2.109, fe80::a... 00:22:FA:D9:E1:50

A special WMI filter operator is "LIKE". It works almost like PowerShell’s comparison operator -like. Use "%" instead of "\*" for wildcards, though. So, to find all services with the keyword "net" in their name, try this:

PS> Get-WmiObject Win32\_Service -Filter 'Name LIKE "%net%"' | Select-Object Name, DisplayName, State

Name DisplayName State

---- ----------- -----

aspnet\_state ASP.NET-Zustandsdienst Stopped

Net Driver HPZ12 Net Driver HPZ12 Stopped

Netlogon Netlogon Running

Netman Network Connections Running

NetMsmqActivator Net.Msmq Listener Adapter Stopped

NetPipeActivator Net.Pipe Listener Adapter Stopped

netprofm Network List Service Running

NetTcpActivator Net.Tcp Listener Adapter Stopped

NetTcpPortSharing Net.Tcp Port Sharing Se... Stopped

WMPNetworkSvc Windows Media Player Ne... Running

PowerShell supports the *[WmiSearcher]* type accelerator, which you can use to achieve basically the same thing you just did with the –query parameter:

$searcher = [WmiSearcher]"select caption,commandline from Win32\_Process where name like 'p%'"

$searcher.Get()| Format-Table [a-z]\* -Wrap

**Direct WMI Object Access**

Every WMI instance has its own unique path. This path is important if you want to access a particular instance directly. The path of a WMI object is located in the *\_\_PATH* property. First use a "traditional" query to list this property and find out what it looks like:

Get-WmiObject Win32\_Service | ForEach-Object { $\_.\_\_PATH }

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="AeLookupSvc"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="AgereModemAudio"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="ALG"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="Appinfo"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="AppMgmt"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="Ati External Event Utility"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="AudioEndpointBuilder"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="Audiosrv"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="Automatic LiveUpdate - Scheduler"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="BFE"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="BITS"

\\JSMITH-PC\root\cimv2:Win32\_Service.Name="Browser"

(...)

The path consists basically of the class name as well as one or more key properties. For services, the key property is *Name* and is the English-language name of the service. If you want to work directly with a particular service through WMI, specify its path and do a type conversion. Use either the *[wmi]* type accelerator or the underlying *[System.Management.ManagementObject]* .NET type:

[wmi]"Win32\_Service.Name='Fax'"

ExitCode : 1077

Name : Fax

ProcessId : 0

StartMode : Manual

State : Stopped

Status : OK

In fact, you don’t necessarily need to specify the name of the key property as long as you at least specify its value. This way, you’ll find all the properties of a specific WMI instance right away.

$disk = [wmi]'Win32\_LogicalDisk="C:"'

$disk.FreeSpace

10181373952

[int]($disk.FreeSpace / 1MB)

9710

$disk | Format-List [a-z]\*

Status :

Availability :

DeviceID : C:

StatusInfo :

Access : 0

BlockSize :

Caption : C:

Compressed : False

ConfigManagerErrorCode :

ConfigManagerUserConfig :

CreationClassName : Win32\_LogicalDisk

Description : Local hard drive

DriveType : 3

ErrorCleared :

ErrorDescription :

ErrorMethodology :

FileSystem : NTFS

FreeSpace : 10181373952

InstallDate :

LastErrorCode :

MaximumComponentLength : 255

MediaType : 12

Name : C:

NumberOfBlocks :

PNPDeviceID :

PowerManagementCapabilities :

PowerManagementSupported :

ProviderName :

Purpose :

QuotasDisabled :

QuotasIncomplete :

QuotasRebuilding :

Size : 100944637952

SupportsDiskQuotas : False

SupportsFileBasedCompression : True

SystemCreationClassName : Win32\_ComputerSystem

SystemName : JSMITH-PC

VolumeDirty :

VolumeName :

VolumeSerialNumber : AC039C05

**Changing System Configuration**

WMIs primary purpose is to read information about the current system configuration but it can also be used to make changes to a system. Most WMI object properties are read-only, but some are writeable, too. In addition, a number of WMI objects contain methods that you can call to make changes.

Note that WMI objects returned by PowerShell Remoting always are read-only. They cannot be used to change the remote system. If you want to change a remote system using WMI objects, you must connect to the remote system using the *-ComputerName* parameter provided by *Get-WmiObject*.

**Modifying Properties**

Most of the properties that you find in WMI objects are read-only. There are few, though, that can be modified. For example, if you want to change the description of a drive, add new text to the *VolumeName* property of the drive:

$drive = [wmi]"Win32\_LogicalDisk='C:'"

$drive.VolumeName = "My Harddrive"

$drive.Put()

Path : \\.\root\cimv2:Win32\_LogicalDisk.DeviceID="C:"

RelativePath : Win32\_LogicalDisk.DeviceID="C:"

Server : .

NamespacePath : root\cimv2

ClassName : Win32\_LogicalDisk

IsClass : False

IsInstance : True

IsSingleton : False

Three conditions must be met before you can modify a property:

* The property must be writeable. Most properties are read-only.
* You require the proper permissions for modifications. The drive description applies to all users of a computer so only administrators may modify them.
* You must use *Put()* to save the modification. Without *Put()*, the modification will not be written back to the system.

**Invoking WMI Methods**

WMI objects derived from the *Win32\_Process* class have a *Terminate()* method you can use to terminate a process. Of course it is much easier to terminate a process with *Stop-Process*, so why would you use WMI? Because WMI supports remote connections. Stop-Process can only stop processes on your local machine.

This line would kill all instances of the Windows Editor "notepad.exe" on your local machine:

Get-WmiObject Win32\_Process -Filter "name='notepad.exe'" |  
 ForEach-Object { $\_.Terminate().ReturnValue }

Add the parameter *-ComputerName* to *Get-WmiObject*, and you'd be able to kill notepads on one or more remote machines - provided you have Administrator privileges on the remote machine.

For every instance that *Terminate()* closes, it returns an object with a number of properties. Only the property *ReturnValue* is useful, though, because it tells you whether the call succeeded. That's why it is generally a good idea to add ".ReturnValue" to all calls of a WMI method. A return value of 0 generally indicates success, any other code failure. To find out what the error codes mean you would have to surf to an Internet search engine and enter the WMI class name (like "Win32\_Process"). One of the first links will guide you to the Microsoft MSDN documentation page for that class. It lists all codes and clear text translations for all properties and method calls.

If you already know the process ID of a process, you can work on the process directly just as you did in the last section because the process ID is the key property of processes. For example, you could terminate the process with the ID 1234 like this:

([wmi]"Win32\_Process='1234'").Terminate()

If you’d rather check your hard disk drive C:\ for errors, the proper invocation is:

([wmi]"Win32\_LogicalDisk='C:'").Chkdsk(...

However, since this method requires additional arguments, the question here is what you should specify. Invoke the method without parentheses in order to get initial brief instructions:

([wmi]"Win32\_LogicalDisk='C:'").Chkdsk

MemberType : Method

OverloadDefinitions : {System.Management.ManagementBaseObject Chkdsk(System.Boolean FixErrors, System.Boolean

VigorousIndexCheck, System.Boolean SkipFolderCycle, System.Boolean ForceDismount, Syst

em.Boolean RecoverBadSectors, System.Boolean OkToRunAtBootUp)}

TypeNameOfValue : System.Management.Automation.PSMethod

Value : System.Management.ManagementBaseObject Chkdsk(System.Boolean FixErrors, System.Boolean

VigorousIndexCheck, System.Boolean SkipFolderCycle, System.Boolean ForceDismount, Syste

m.Boolean RecoverBadSectors, System.Boolean OkToRunAtBootUp)

Name : Chkdsk

IsInstance : True

*Get-Member* will tell you which methods a WMI object supports:

PS> Get-WmiObject Win32\_Process | Get-Member -MemberType Method

TypeName: System.Management.ManagementObject#root\cimv2\Win32\_Process

Name MemberType Definition

---- ---------- ----------

AttachDebugger Method System.Management.ManagementBaseObject AttachDebugger()

GetOwner Method System.Management.ManagementBaseObject GetOwner()

GetOwnerSid Method System.Management.ManagementBaseObject GetOwnerSid()

SetPriority Method System.Management.ManagementBaseObject SetPriority(System.Int32 Priority)

Terminate Method System.Management.ManagementBaseObject Terminate(System.UInt32 Reason)

**Static Methods**

There are WMI methods not just in WMI objects that you retrieved with Get-WmiObject. Some WMI classes also support methods. These methods are called "static".

If you want to renew the IP addresses of all network cards, use the *Win32\_NetworkAdapterConfiguration* class and its static method *RenewDHCPLeaseAll()*:

([wmiclass]"Win32\_NetworkAdapterConfiguration").RenewDHCPLeaseAll().ReturnValue

You get the WMI class by using type conversion. You can either use the *[wmiclass]* type accelerator or the underlying *[System.Management.ManagementClass]*.

The methods of a WMI class are also documented in detail inside WMI. For example, you get the description of the *Win32Shutdown()* method of the *Win32\_OperatingSystem* class like this:

$class = [wmiclass]'Win32\_OperatingSystem'

$class.Options.UseAmendedQualifiers = $true

(($class.methods["Win32Shutdown"]).Qualifiers["Description"]).Value

The Win32Shutdown method provides the full set of shutdown options supported by Win32  
 operating systems. The method returns an integer value that can be interpretted as follows:

0 – Successful completion.

Other – for integer values other than those listed above, refer to Win32 error code documentation.

If you’d like to learn more about a WMI class or a method, navigate to an Internet search page like Google and specify as keyword the WMI class name, as well as the method. It’s best to limit your search to the Microsoft MSDN pages: *Win32\_NetworkAdapterConfiguration RenewDHCPLeaseAll site:msdn2.microsoft.com*.

**Using WMI Auto-Documentation**

Nearly every WMI class has a built-in description that explains its purpose. You can view this description only if you first set a hidden option called *UseAmendedQualifiers* to *$true*. Once that’s done, the WMI class will readily supply information about its function:

$class = [wmiclass]'Win32\_LogicalDisk'

$class.psbase.Options.UseAmendedQualifiers = $true

($class.psbase.qualifiers["description"]).Value

The Win32\_LogicalDisk class represents a data source that resolves to an actual local storage  
 device on a Win32 system. The class returns both local as well as mapped logical disks.  
 However, the recommended approach is to use this class for obtaining information on local  
 disks and to use the Win32\_MappedLogicalDisk for information on mapped logical disk.

In a similarly way, all the properties of the class are documented. The next example retrieves the documentation for the property *VolumeDirty* and explains what its purpose is:

$class = [wmiclass]'Win32\_LogicalDisk'

$class.psbase.Options.UseAmendedQualifiers = $true

($class.psbase.properties["VolumeDirty"]).Type

Boolean

(($class.psbase.properties["VolumeDirty"]).Qualifiers["Description"]).Value

The VolumeDirty property indicates whether the disk requires chkdsk to be run at next boot up time.   
 The property is applicable to only those instances of logical disk that represent a physical disk in  
 the machine. It is not applicable to mapped logical drives.

**WMI Events**

WMI returns not only information but can also wait for certain events. If the events occur, an action will be started. In the process, WMI can alert you when one of the following things involving a WMI instance happens:

* **\_\_InstanceCreationEvent**: A new instance was added such as a new process was started or a new file created.
* **\_\_InstanceModificationEvent**: The properties of an instance changed. For example, the FreeSpace property of a drive was modified.
* **\_\_InstanceDeletionEvent**: An instance was deleted, such as a program was shut down or a file deleted.
* **\_\_InstanceOperationEvent**: This is triggered in all three cases.

You can use these to set up an alarm signal. For example, if you want to be informed as soon as Notepad is started, type:

Select \* from \_\_InstanceCreationEvent WITHIN 1 WHERE targetinstance ISA 'Win32\_Process' AND  
 targetinstance.name = 'notepad.exe'

*WITHIN* specifies the time interval of the inspection and “WITHIN 1” means that you want to be informed no later than one second after the event occurs. The shorter you set the interval, the more effort involved, which means that WMI will require commensurately more computing power to perform your task. As long as the interval is kept at not less than one second, the computation effort will be scarcely perceptible. Here is an example:

$alarm = New-Object Management.EventQuery

$alarm.QueryString = "Select \* from \_\_InstanceCreationEvent WITHIN 1 WHERE targetinstance ISA 'Win32\_Process' AND `  
targetinstance.name = 'notepad.exe'"

$watch = New-Object Management.ManagementEventWatcher $alarm

“Start Notepad after issuing a wait command:”

$result = $watch.WaitForNextEvent()

“Get target instance of Notepad:”

$result.targetinstance

“Access the live instance:”

$path = $result.targetinstance.\_\_path

$live = [wmi]$path

# Close Notepad using the live instance

$live.terminate()

**Using WMI Remotely**

WMI comes with built-in remoting so you can retrieve WMI objects not just from your local machine but also across the network. WMI uses "traditional" remoting techniques like DCOM which are also used by the Microsoft Management Consoles.

To be able to use WMI remoting, your network must support DCOM calls (thus, the firewall needs to be set up accordingly). Also, you need to have Administrator privileges on the target machine.

**Accessing WMI Objects on Another Computer**

Use the *-ComputerName* parameter of *Get-WmiObject* to access another computer system using WMI. Then specify the name of the computer after it:

Get-WmiObject -ComputerName pc023 Win32\_Process

You can also specify a comma-separated list of a number of computers and return information from all of them. The parameter *-ComputerName* accepts an array of computer names. Anything that returns an array of computer names or IP addresses can be valid input. This line, for example, would read computer names from a file:

Get-WmiObject Win32\_Process -ComputerName (Get-Content c:\serverlist.txt)

If you want to log on to the target system using another user account, use the *–Credential* parameter to specify additional log on data as in this example:

$credential = Get-Credential

Get-WmiObject -ComputerName pc023 -Credential $credential Win32\_Process

In addition to the built-in remoting capabilities, you can use Get-WmiObject via PowerShell Remoting (if you have set up PowerShell Remoting correctly). Here, you send the WMI command off to the remote system:

Invoke-Command { Get-WmiObject Win32\_BIOS } -ComputerName server12, server16

Note that all objects returned by PowerShell Remoting are read-only and do not contain methods anymore. If you want to change WMI properties or call WMI methods, you need to do this inside the script block you send to the remote system - so it needs to be done before PowerShell Remoting sends back objects to your own system.

**WMI Background Information**

WMI has a hierarchical structure much like a file system does. Up to now, all the classes that you have used have come from the WMI “directory” *root\cimv2*. Third-party vendors can create additional WMI directories, known as *Namespaces*, and put in them their own classes, which you can use to control software, like Microsoft Office or hardware like switches and other equipment.

Because the topmost directory in WMI is always named *root*, from its location you can inspect existing namespaces. Get a display first of the namespaces on this level:

Get-WmiObject -Namespace root \_\_Namespace | Format-Wide Name

subscription DEFAULT

MicrosoftDfs CIMV2

Cli nap

SECURITY RSOP

Infineon WMI

directory Policy

ServiceModel SecurityCenter

MSAPPS12 Microsoft

aspnet

As you see, the *cimv2* directory is only one of them. What other directories are shown here depends on the software and hardware that you use. For example, if you use Microsoft Office, you may find a directory called *MSAPPS12*. Take a look at the classes in it:

Get-WmiObject -Namespace root\msapps12 -List | Where-Object { $\_.Name.StartsWith("Win32\_") }

Win32\_PowerPoint12Tables Win32\_Publisher12PageNumber

Win32\_Publisher12Hyperlink Win32\_PowerPointSummary

Win32\_Word12Fonts Win32\_PowerPointActivePresentation

Win32\_OutlookDefaultFileLocation Win32\_Word12Document

Win32\_ExcelAddIns Win32\_PowerPoint12Table

Win32\_ADOCoreComponents Win32\_Publisher12SelectedTable

Win32\_Word12CharacterStyle Win32\_Word12Styles

Win32\_OutlookSummary Win32\_Word12DefaultFileLocation

Win32\_WordComAddins Win32\_PowerPoint12AlternateStartupLoc

Win32\_OutlookComAddins Win32\_ExcelCharts

Win32\_Word12Settings Win32\_FrontPageActiveWeb

Win32\_OdbcDriver Win32\_AccessProject

Win32\_Word12StartupFileLocation Win32\_ExcelActiveWorkbook

Win32\_FrontPagePageProperty Win32\_Publisher12MailMerge

Win32\_Language Win32\_FrontPageAddIns

Win32\_Word12PageSetup Win32\_Word12HeaderAndFooter

Win32\_ServerExtension Win32\_Publisher12ActiveDocumentNoTable

Win32\_Word12Addin Win32\_WordComAddin

Win32\_PowerPoint12PageNumber Win32\_JetCoreComponents

Win32\_Publisher12Fonts Win32\_Word12Table

Win32\_OutlookAlternateStartupFile Win32\_Word12Tables

Win32\_Access12ComAddins Win32\_Excel12AlternateStartupFileLoc

Win32\_Word12FileConverters Win32\_Access12StartupFolder

Win32\_Word12ParagraphStyle Win32\_Access12ComAddin

Win32\_Excel12StartupFolder Win32\_PowerPointPresentation

Win32\_FrontPageWebProperty Win32\_Publisher12Table

Win32\_Publisher12StartupFolder Win32\_WebConnectionErrorText

Win32\_ExcelSheet Win32\_Publisher12Tables

Win32\_FrontPageTheme Win32\_PowerPoint12ComAddins

Win32\_Word12Template Win32\_ExcelComAddins

Win32\_Access12AlternateStartupFileLoc Win32\_Word12ActiveDocument

Win32\_PublisherSummary Win32\_Publisher12DefaultFileLocation

Win32\_Word12Field Win32\_Publisher12Hyperlinks

Win32\_PowerPoint12ComAddin Win32\_PowerPoint12Hyperlink

Win32\_PowerPoint12DefaultFileLoc Win32\_Publisher12Sections

Win32\_OutlookStartupFolder Win32\_Access12JetComponents

Win32\_Word12ActiveDocumentNotable Win32\_Publisher12CharacterStyle

Win32\_Word12Hyperlinks Win32\_Word12MailMerge

Win32\_Word12FileConverter Win32\_PowerPoint12Hyperlinks

Win32\_FrontPageActivePage Win32\_Word12Summary

Win32\_OleDbProvider Win32\_Publisher12PageSetup

Win32\_Word12SelectedTable Win32\_PowerPoint12StartupFolder

Win32\_OdbcCoreComponent Win32\_PowerPoint12PageSetup

Win32\_FrontPageSummary Win32\_AccessSummary

Win32\_Word12Hyperlink Win32\_OfficeWatsonLog

Win32\_Publisher12Font Win32\_WebConnectionErrorMessage

Win32\_AccessDatabase Win32\_Publisher12Styles

Win32\_Publisher12ActiveDocument Win32\_Word12AlternateStartupFileLocation

Win32\_PowerPoint12Fonts Win32\_Word12Sections

Win32\_ExcelComAddin Win32\_Excel12DefaultFileLoc

Win32\_Word12Fields Win32\_ExcelActiveWorkbookNotable

Win32\_Publisher12COMAddIn Win32\_ExcelWorkbook

Win32\_OutlookComAddin Win32\_PowerPoint12Font

Win32\_FrontPageAddIn Win32\_ExcelChart

Win32\_WebConnectionError Win32\_Word12Font

Win32\_RDOCoreComponents Win32\_Word12PageNumber

Win32\_Publisher12ParagraphStyle Win32\_Publisher12COMAddIns

Win32\_Transport Win32\_Access12DefaultFileLoc

Win32\_FrontPageThemes Win32\_ExcelSummary

Win32\_ExcelAddIn Win32\_Publisher12AlternateStartupFileLocation

Win32\_PowerPoint12SelectedTable

**Converting the WMI Date Format**

WMI uses special date formats. For example, look at *Win32\_OperatingSystem* objects:

Get-WmiObject Win32\_OperatingSystem | Format-List \*time\*

CurrentTimeZone : 120

LastBootUpTime : 20111016085609.375199+120

LocalDateTime : 20111016153922.498000+120

The date and time are represented a sequence of numbers: first the year, then the month, and finally the day. Following this is the time in hours, minutes, and milliseconds, and then the time zone. This is the so-called *DMTF* standard, which is hard to read. However, you can use *ToDateTime()* of the *ManagementDateTimeConverter* .NET class to decipher this cryptic format:

$boottime = (Get-WmiObject win32\_OperatingSystem).LastBootUpTime

$boottime

20111016085609.375199+120

$realtime = [System.Management.ManagementDateTimeConverter]::ToDateTime($boottime)

$realtime

Tuesday, October 16, 2011 8:56:09 AM

Now you can also use standard date and time cmdlets such as *New-TimeSpan* to calculate the current system uptime:

New-TimeSpan $realtime (Get-Date)

Days : 0

Hours : 6

Minutes : 47

Seconds : 9

Milliseconds : 762

Ticks : 244297628189

TotalDays : 0.282751884478009

TotalHours : 6.78604522747222

TotalMinutes : 407.162713648333

TotalSeconds : 24429.7628189

TotalMilliseconds : 24429762.8189