WMI Query Language via PowerShell

Ravikanth Chaganti

Explore basics of WMI Query Language, different types of WMI queries, and learn how PowerShell can be used to retrieve WMI management information using WQL.

Table of Contents

This book is dedicated to Andrew Tearle, the most passionate PowerSheller and a good friend.
Rest in peace Andy.

Acknowledgements
I would like to thank Shay Levy (MVP), Aleksandar Nikolic (MVP), Philip LaVoie, and Robert Robelo for providing their feedback. Their feedback really helped shape the ebook and include extra content that was not planned initially. Also, thanks to everyone who read my blog posts on WMI query language and provided feedback. Your encouragement and support helped me write quite a bit about WQL and now this ebook.

Introduction

Windows Management Instrumentation (WMI) is Microsoft's implementation of Web Based Enterprise Management (WBEM), which is an industry initiative to develop a standard technology for accessing management information in an enterprise environment. WMI uses the Common Information Model (CIM) industry standard to represent systems, applications, networks, devices, and other managed components. CIM is developed and maintained by the Distributed Management Task Force (DMTF). We can write WMI scripts to automate several tasks on local or remote computer(s).

Windows PowerShell has a few cmdlets to retrieve the management data exposed by WMI. We can list these cmdlets by using:

```
#Use Get-Command and mention WMI* as the Noun
Get-Command -Noun WMI*
```

There are five cmdlets that are used to work with WMI. However, within the scope of this book, we shall use *Get-WMIObject* and *Register-WMIEvent* only. Get-WMIObject, in its basic usage, gets the instance(s) of a WMI class. So, for example, if we need to list out all drives of type 3 (disk drives) in a system,

```
Get-WMIObject -Class Win32_LogicalDisk | Where-Object {
    $_.DriveType -eq 3
}
```

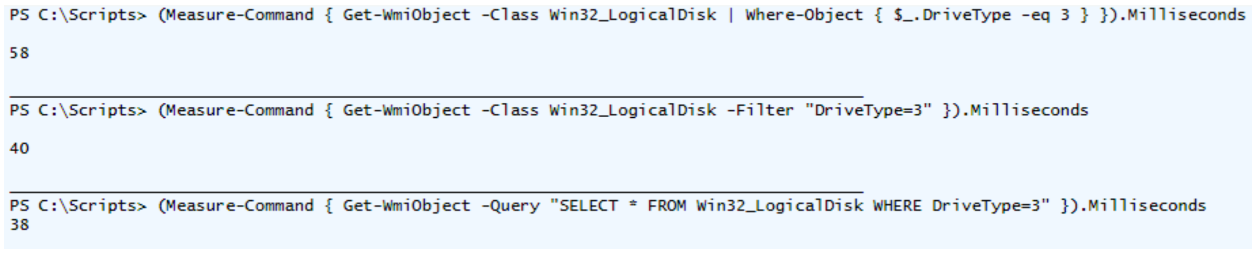
In the above method, we retrieve all instances of *Win32_LogicalDisk* and then pass it to *Where-Object* to filter out what we need. This can take a while depending on how many instances are there. You can use an alternative approach by specifying -*Query* parameter instead of -*Class.*

```
#This example uses -Query parameter and specifies the query using WQL
Get-WMIObject -Query "SELECT * FROM Win32_LogicalDisk WHERE DriveType=3"
```

Introducing WMI Query Language

The above example uses WMI Query Language to get the same information as the earlier example but a bit faster. We can verify this using *Measure-Command* cmdlet.

Let us see this in action:



```
PS C:\Scripts> (Measure-Command { Get-WmiObject -Class Win32_LogicalDisk | Where-Object { $_.DriveType -eq 3 } }).Milliseconds

58

_____________________________________________________________________________________
PS C:\Scripts> (Measure-Command { Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" }).Milliseconds

40

_____________________________________________________________________________________
PS C:\Scripts> (Measure-Command { Get-WmiObject -Query "SELECT * FROM Win32_LogicalDisk WHERE DriveType=3" }).Milliseconds
38
```

In the above example, we used three variations of *Get-WMIObject* to do the same job of retrieving all instances of *Win32_LogicalDisk* where the DriveType is 3 (a disk drive). From the output, we can see that using -*Query* and -*Filter* are the fastest ways to retrieve the WMI information.

> **Note**
> -*Filter*, as we shall see in next chapters, is a variation of -*Query*. In fact, the value of -*Filter* represents the value of a WHERE clause when using -*Query* parameter. When using -*Filter*, *Get-WMIObejct* cmdlet internally builds the WMI query as required.

The above example is very basic and may not really explain the usefulness of WQL — the speed of execution is just one benefit. There are quite a few advanced querying techniques that can be used to retrieve WMI information in an efficient manner. And, sometimes, such as working with WMI events, WQL becomes a necessity. We shall see each of these benefits as we proceed further.

So, what is WQL?
The WMI Query Language is a subset of the American National Standards Institute Structured Query Language (ANSI SQL)—with minor semantic changes. Similar to SQL, WQL has a set of keywords & operators and supports three types of queries.

WMI Query Types

WMI supports three types of queries:

1. Data Queries
2. Event Queries
3. Schema Queries

Data Queries

This type is the simplest form of querying for WMI data and the most commonly used query type when working with WMI. Data queries are used to retrieve class instances and data associations. The earlier example, where we queried for all instances of *Win32_LogicalDisk* where the driveType is 4, is a data query. The WQL keywords such as SELECT, ASSOCIATORS OF, REFERENCES OF, and ISA are used in data queries.

Event Queries

The event queries are used to create WMI event subscriptions. For example, using these queries we can create an event subscription to notify whenever a USB drive gets attached to the system. The WQL keywords such as GROUP, HAVING, and WITHIN are used when creating event queries. The event queries are critical when we want use PowerShell cmdlets such as *Register-WMIEvent* for creating **temporary** event consumers. Using this cmdlet, we can create WMI event consumers and invoke an action when the event gets triggered. We shall see more on this in the subsequent sections.

Schema Queries

Schema queries are used to retrieve class definitions (rather than class instances) and schema associations. In layman's terms, these queries are used to get information about WMI and its structure. Schema queries return a result set of class definition objects rather than actual instances of classes. The WQL keywords such as SELECT, ASSOCIATORS OF, REFERENCES OF, and ISA are used in schema queries and of course, in a slightly different way than how data queries use these keywords.

WMI does not support cross-namespace queries or associations. Using WQL, we cannot query for all instances of a specified class residing in all of the namespaces on the target computer. Also, WQL queries are read-only. There are no keywords such as INSERT or UPDATE. We cannot modify the WMI objects using WQL.

WQL Keywords

Similar to SQL, WQL queries use keywords to retrieve data from the management objects. WQL has 19 keywords to perform these queries against WMI repositories. Even though there are 19 WQL keywords, only a few of them can be used in all three possible query types we discussed earlier. The following table lists all the WQL keywords and lists the query type in which they can be used.

Keyword	Query Type			Description
	Data	Schema	Event	
AND	X		X	Combines two Boolean expressions, and returns TRUE when both expressions are TRUE.
ASSOCIATORS OF	X	X		Retrieves all instances that are associated with a source instance. Use this statement with schema queries and data queries.
__CLASS	X	X		References the class of the object in a query.
FROM	X	X	X	Specifies the class that contains the properties listed in a SELECT statement. Windows Management Instrumentation (WMI) supports data queries from only one class at a time.
GROUP			X	Causes WMI to generate one notification to represent a group of events.
HAVING			X	Filters the events that are received during the grouping interval that is specified in the WITHIN clause.
IS	X		X	Comparison operator used with NOT and NULL. The syntax for this statement is the following: IS [NOT] NULL (where NOT is optional)
ISA	X	X	X	Operator that applies a query to the subclasses of a specified class

KEYSONLY	X			Used in REFERENCES OF and ASSOCIATORS OF queries to ensure that the resulting instances are only populated with the keys of the instances. This reduces the overhead of the call.
LIKE	X			Operator that determines whether or not a given character string matches a specified pattern.
NOT	X			Comparison operator that use in a WQL SELECT query
NULL	X			Indicates an object does not have an explicitly assigned value. NULL is not equivalent to zero (0) or blank.
OR	X			Combines two conditions. When more than one logical operator is used in a statement, the OR operators are evaluated after the AND operators.
REFERENCES OF	X	X		Retrieves all association instances that refer to a specific source instance. The REFERENCES OF statement is similar to the ASSOCIATORS OF statement. However, it does not retrieve endpoint instances; it retrieves the association instances.
SELECT	X	X	X	Specifies the properties that are used in a query.
TRUE	X		X	Boolean operator that evaluates to -1 (minus one).
WHERE	X	X	X	Narrows the scope of a data, event, or schema query.
WITHIN			X	Specifies polling or grouping interval.
FALSE	X	X	X	Boolean operator that evaluates to 0 (zero).

We shall look at each of these keywords when we start discussing different types of WMI queries in-depth.

WQL Operators

Similar to arithmetic operators, WQL uses the same set. The following table lists all the operators supported in WQL.

Operator	Description
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!= or <>	Not equal to

A few WQL keywords such as IS, ISA, NOT, and LIKE[1] can also be considered as operators. In these keywords, IS and IS NOT operators are valid in the WHERE clause only if the constant is NULL.

Once again, there is a detailed discussion on how to use these operators in the upcoming sections.

[1] LIKE operator is not available in Windows 2000

Tools for the job

In the first chapter, we looked at what is WQL, different types of WMI queries, and WQL keywords & operators. Before we dig deep into this subject, we shall look at different tools available to execute WMI queries.

There are several tools that can be used to execute WMI queries. This includes

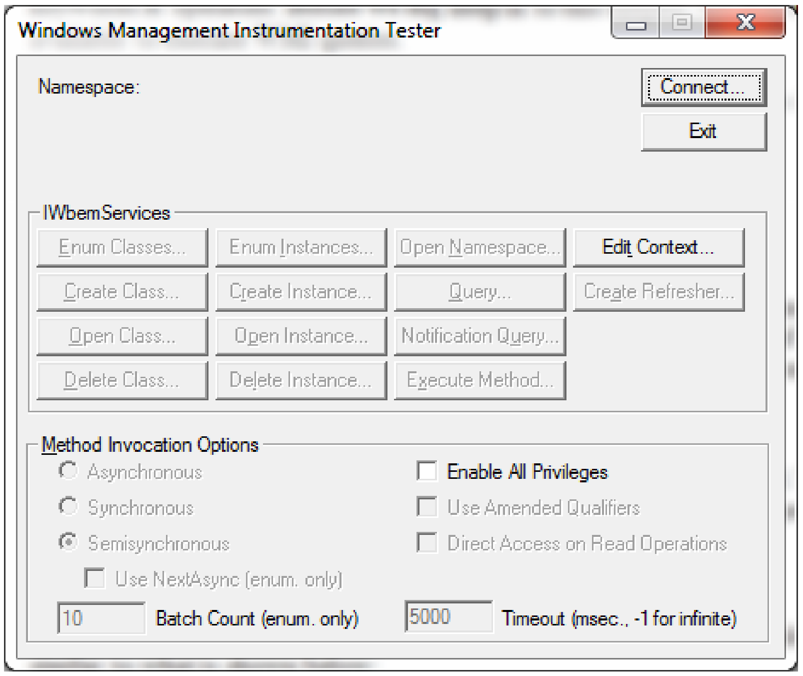
- WBEMTest.exe
- WMI tools
- [WMISEARCHER] type accelerator in PowerShell
- PowerShell WMI cmdlets

In the subsequent sections, we shall look at each of the above mentioned tools in detail and see how WMI queries can be executed using these tools. However, rest of the chapters in this book will focus only on WMI PowerShell cmdlets and refer to other tools only when required.

WBEMTEST

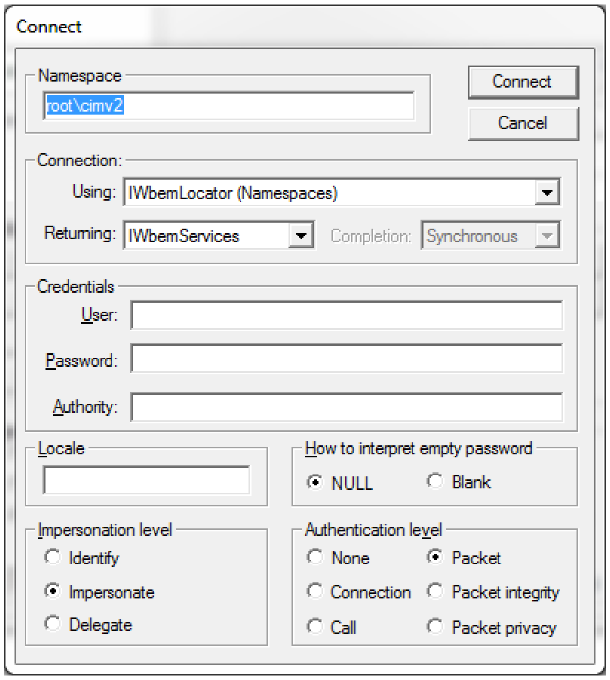
WBEMTest.exe is present on every computer that has WMI installed. This tool provides a great way to explore WMI classes and instances. This was initially developed to test WMI COM API and hence can do everything you can achieve using WMI.

To open WBEMTest.exe,
Click Start-> Run, type "WBEMTest.exe" and Press Enter. This should open a GUI window similar to what is shown below:



Asynchronous
Synchronous
[✓ Semisynchronous]
☐ Use NextAsync (enum. only)
☐ Enable All Privileges
☐ Use Amended Qualifiers
☐ Direct Access on Read Operations
10 Batch Count (enum. only)
5000 Timeout (msec., -1 for infinite)

Now, we can connect to a WMI namespace using the “Connect” button at the top.



Namespace: root\cimv2

Connection:
Using: IWbemLocator (Namespaces)
Returning: IWbemServices Completion: Synchronous

Credentials
User:
Password:
Authority:

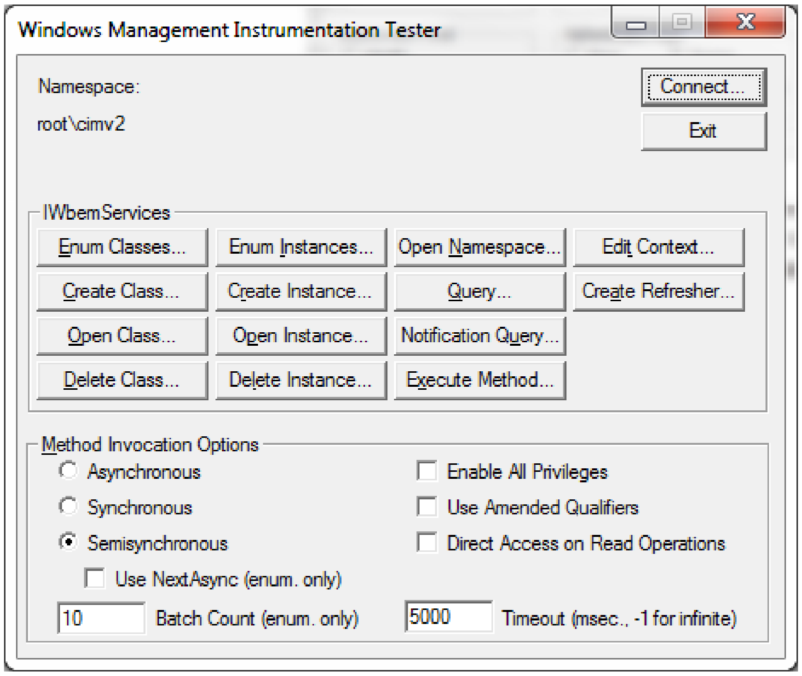
Locale

How to interpret empty password: [✓ NULL] Blank

Impersonation level: Identify [✓ Impersonate] Delegate

Authentication level: None [✓ Packet] Connection Packet integrity Call Packet privacy

Assuming you have the necessary permissions to access root\cimv2 namespace, just click “Connect”. Once we are connected to the root\cimv2 namespace, we can perform several actions such as enumerating classes, creating instances, executing methods, and also executing WMI queries. The last part is what we are interested in.

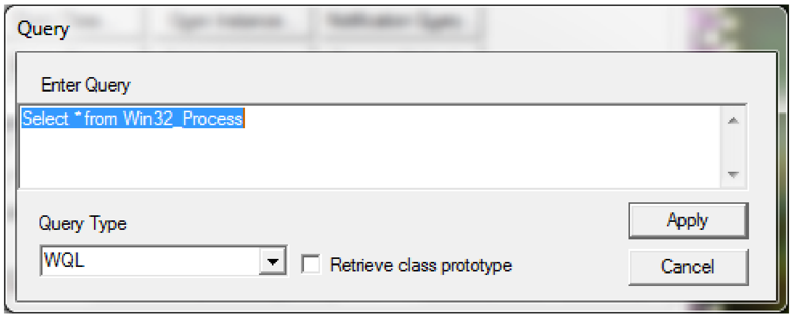


Namespace:
root\cimv2

Method Invocation Options
Asynchronous
Synchronous
[✓ Semisynchronous]
Use NextAsync (enum. only)
Enable All Privileges
Use Amended Qualifiers
Direct Access on Read Operations
10 Batch Count (enum. only)
5000 Timeout (msec., -1 for infinite)

In the above window, we can use the “Query” and “Notification Query” options to execute WMI queries. The “Query” option can be used for WMI data and schema queries. And, “Notification Query” can be used for running WMI event queries.

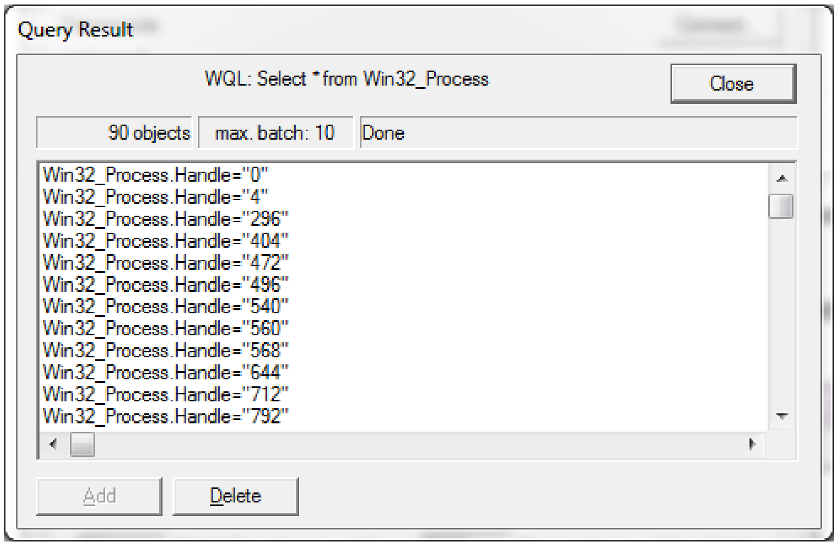
As an example, we shall now look at how we can execute simple data queries and save the notification queries part for a later part of this book.



```
Select * from Win32_Process
```

Query Type: WQL

Click on the “Query” button in the WBEMTest window; enter a WQL statement as shown above and click “Apply”.



```
WQL: Select * from Win32_Process
90 objects   max. batch: 10   Done
Win32_Process.Handle="0"
Win32_Process.Handle="4"
Win32_Process.Handle="296"
Win32_Process.Handle="404"
Win32_Process.Handle="472"
Win32_Process.Handle="496"
Win32_Process.Handle="540"
Win32_Process.Handle="560"
Win32_Process.Handle="568"
Win32_Process.Handle="644"
Win32_Process.Handle="712"
Win32_Process.Handle="792"
```

The query result window shows all instances of *Win32_Process* class. This is it. WBEMTest is an expert tool and is a great tool to explore WMI interfaces.

WMI Administrative Tools

WMI administrative tools can be used to view and edit WMI classes, properties, qualifiers and instances in a WMI repository. You can also use these tools to invoke WMI methods, manage WMI objects, register and receive WMI events. The free WMI tools download package includes

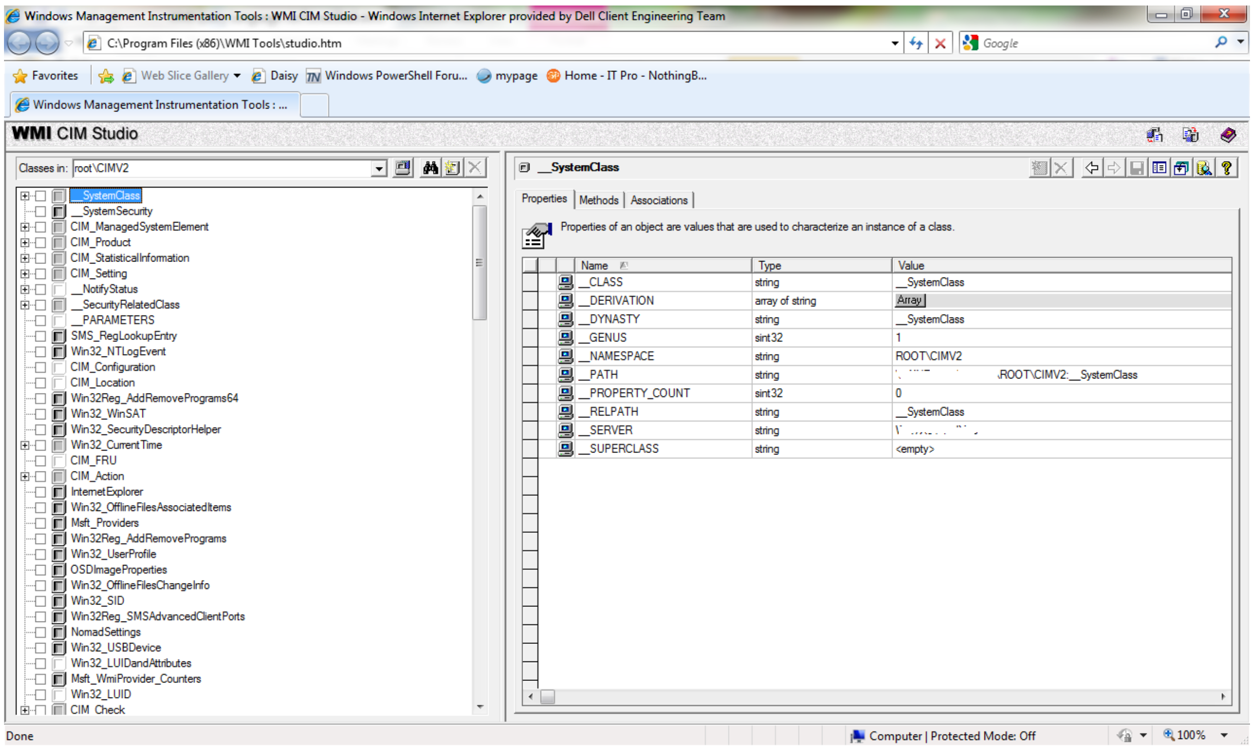
1. WMI CIM Studio
2. WMI Object browser
3. WMI Event registration

4. WMI Event viewer

These tools can be downloaded from http://www.microsoft.com/downloads/en/details.aspx?FamilyID=6430f853-1120-48db-8cc5-f2abdc3ed314

> **Note**
> WMI CIM Studio can be used to execute data/schema queries. WMI Tools do not support executing WMI event queries.

Once the WMI tools are installed, open WMI CIM studio by selecting Start -> All Programs -> WMI Tools -> WMI CIM Studio. This prompts for the WMI namespace selection and credentials.


Windows Management Instrumentation Tools : WMI CIM Studio - Windows Internet Explorer provided by Dell Client Engineering Team

C:\Program Files (x86)\WMI Tools\studio.htm

WMI CIM Studio

Classes in: root\CIMV2

- __SystemClass
- __SystemSecurity
- CIM_ManagedSystemElement
- CIM_Product
- CIM_StatisticalInformation
- CIM_Setting
- __NotifyStatus
- __SecurityRelatedClass
- __PARAMETERS
- SMS_RegLookupEntry
- Win32_NTLogEvent
- CIM_Configuration
- CIM_Location
- Win32Reg_AddRemovePrograms64
- Win32_WinSAT
- Win32_SecurityDescriptorHelper
- Win32_CurrentTime
- CIM_FRU
- CIM_Action
- InternetExplorer
- Win32_OfflineFilesAssociatedItems
- Msft_Providers
- Win32Reg_AddRemovePrograms
- Win32_UserProfile
- OSDImageProperties
- Win32_OfflineFilesChangeInfo
- Win32_SID
- Win32Reg_SMSAdvancedClientPorts
- NomadSettings
- Win32_USBDevice
- Win32_LUIDandAttributes
- Msft_WmiProvider_Counters
- Win32_LUID
- CIM_Check

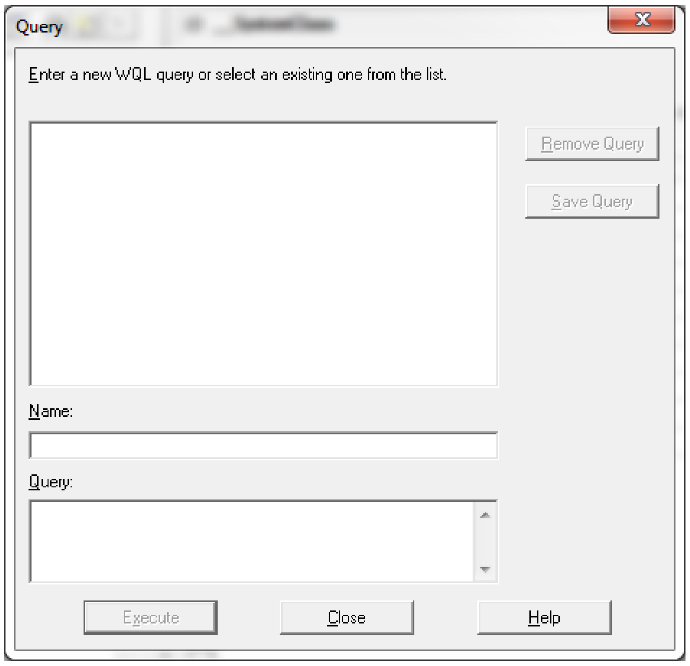
__SystemClass

Properties of an object are values that are used to characterize an instance of a class.

Name	Type	Value
__CLASS	string	__SystemClass
__DERIVATION	array of string	Array
__DYNASTY	string	__SystemClass
__GENUS	sint32	1
__NAMESPACE	string	ROOT\CIMV2
__PATH	string	[redacted]ROOT\CIMV2:__SystemClass
__PROPERTY_COUNT	sint32	0
__RELPATH	string	__SystemClass
__SERVER	string	[redacted]
__SUPERCLASS	string	<empty>

Done | Computer \| Protected Mode: Off | 100%

In the resulting browser window, click on the icon at the top-right corner. This brings up another window as shown here:


Query
Enter a new WQL query or select an existing one from the list.
Remove Query
Save Query
Name:
Query:
Execute
Close
Help

Type '*Select * from Win32_Process*' (without quotes) in the query textbox and click "Execute". This shows all the instances of *Win32_Process* class. Similarly, we can execute schema queries using CIM Studio. The WMI tools are used for many other purposes than just executing WMI queries. These tools are also a great way to explore and learn about WMI.

[WMISEARCHER] type accelerator

Windows PowerShell provides shortcuts to allow direct access to .NET namespaces. These shortcuts are called type accelerators. There are three type accelerators to access WMI management namespaces.

WMI type accelerator	Description
[WMI]	Retrieve single instance of a WMI class.
[WMICLASS]	Access static properties and methods of a WMI class.
[WMISEARCHER]	Search for WMI objects

Within the scope of this book, we are more interested in how [WMISEARCHER] type accelerator can be used to run WMI queries.

[WMISEARCHER] is a shortcut for System.Management.ManagementObjectSearcher[2] .NET namespace. The way we use this type accelerator is:

1. Provide a WQL statement.
2. Invoke the Get() method to execute the WMI query.

[2] ManagementObjectSearcher Class - http://msdn.microsoft.com/en-us/library/z0zx1aya.aspx

Simple! And, this is how we use it at the PowerShell console.

> **Note**
> To see a list of all members of [WMISEARCHER] accelerators you can pipe [WMISEARCHER]"" to Get-Member cmdlet.

```
$svcs = [WMISEARCHER]"SELECT * FROM Win32_Service WHERE State='Running'"
$svcs.Get()
```

In the above code snippet, we are using the [WMISEARCHER] shortcut to get a list of all services in "running" state. By default, the scope of the query is set to root\cimv2 namespace. In case you want to change the scope to a different namespace, you can do so by setting the "scope" property of this object. For example,

```
$objSearch = [WMISEARCHER]"SELECT * FROM MSPower_DeviceEnable"
$objSearch.Scope = "root\WMI"
$objSearch.Get()
```

In this above example, we change the scope to root\WMI namespace to retrieve all the instances of *MSPower_DeviceEnable* class. Using this type accelerator, we can set some extended properties[3] such as blocksize, timeout, etc. These options are not available in the built-in WMI cmdlets method.

We can use [WMISEARCHER] for some quick access to WMI information but it gets quite verbose soon. This brings us to the last section of this chapter – using PowerShell WMI cmdlets to execute WMI queries.

PowerShell WMI cmdlets

As mentioned in the first chapter, PowerShell v2.0 includes five WMI cmdlets. Within these five cmdlets, *Get-WMIObject* and *Register-WMIEvent* can be used to execute WMI queries. *Get-WMIObject* can be used for data/schema queries while *Register-WMIEvent* can be used for triggering event queries.

We have seen several examples using Get-WMIObject already. So, here is a quick example of an event query using *Register-WMIEvent*.

```
Register-WMIEvent -Query "SELECT * FROM __InstanceCreationEvent WITHIN 10
WHERE TargetInstance ISA 'Win32_Process'" -Action {Write-Host "New process
created"}
```

The above query will display a message, "New Process Created", every time a new process gets created on the system. Don't worry much even if you don't understand anything here. WMI event queries require a complete discussion and we shall see that in the later chapters.

[3] [WMISEARCHER] Query options: http://msdn.microsoft.com/en-us/library/system.management.enumerationoptions.aspx

There are several other tools to execute WMI queries or access WMI information in general. This list includes Sapien WMI Browser[4], MoW's WMI Explorer PowerShell script, etc. Go, explore!

[4] Sapien's WMI browser - http://www.sapien.com/

WMI Data Queries

WMI data queries are the simplest form of querying for WMI data. WMI data queries are used to retrieve class instances and associations. There are several keywords and operators that are used in WMI data queries. This includes keywords such as SELECT, FROM, WHERE, ASSOCIATORS OF, REFERENCES OF, NOT, LIKE, IS, NULL, and all other operators we saw in chapter one.

In this chapter, we shall start looking at the most commonly used form of data queries and then move on to data queries for association. As a part of the examples, we shall look at how some of the operators are used, gotchas when using operators, and finally how to use multiple operators in a query.

SELECT, FROM, and WHERE

The general syntax when using SELECT keyword for data queries is:

SELECT [* | Property Names] FROM ClassName

So, if you want to get all instances of Win32_Service class and all properties,

```
$query = "SELECT * FROM Win32_Service"
Get-WMIObject -Query $query
```

This will list all instances of Win32_Service and properties of each instance.

> **Note**
> Remember that you can perform data queries only from one class at a time. For example, the following query will produce an invalid query error:
> ```
> Get-WMIObject -Query "SELECT * FROM win32_Service, Win32_Process"
> ```

What if we want to limit the instances to one particular service? Let us say AudioSrv. We can use WHERE clause to filter that. Here is how we do it:

```
$query = "SELECT * FROM Win32_Service WHERE Name='AudioSrv'"
Get-WMIObject -Query $query
```

This will list only one instance and the properties of that instance. WHERE is used to narrow the scope of retrieved data. This keyword can be used in all of the three query types. In general, WHERE clause when used with SELECT statement can take one of the following forms:

SELECT * FROM class WHERE property operator constant
SELECT * FROM class WHERE constant operator property

In the above two forms, property denotes a valid property of a WMI instance, operator is any valid WQL operator and constant must be of the correct type for the property. Let us look at an example for the second form of using WHERE.

```
$query = "SELECT Name, State FROM Win32_Service WHERE 'AudioSrv' LIKE Name"
Get-WMIObject -Query $query
```

In the above example, we replaced '*' with Name & State to limit the number of properties in the output. This query, when executed, gets us an instance of Win32_Service with name 'audiosrv' and lists only the name and state properties of the service. By doing so and in case of remote query execution, we are reducing the bandwidth required to execute the query and the amount of data we get in return.

> **Note**
> The above query outputs the system properties such as __PATH, etc in the output. This is because the default formatting for the WMI class is lost when we specify a selected set of properties.

Using Operators

In this section, we shall look at different operators that can be used with WMI data queries and how we can use them.

LIKE

In the preceding example, we filtered out the instances by using WHERE clause and specifying that we need only one instance of Win32_Service. What if we don't know the exact service name but we know that it has the word 'audio' in it.

```
$query = "SELECT * FROM Win32_Service WHERE Name LIKE '%Audio%'"
Get-WMIObject -Query $query
```

This will list all the services that have the word 'audio' in the name of the service.

> **Note**
> You can use *-Filter* parameter instead of a WHERE clause or even *-Query*. When using -*Filter*, *Get-WMIObject* cmdlet builds the required WQL statement internally. For example:
> ```
> Get-WMIObject -Class Win32_Service -Filter {Name='AudioSrv'}
> ```
> This is just a preference and in the scope of this book, I will use *-Query* only.

Observe carefully how we used the keyword LIKE and wrapped the word audio between %%. We have known, probably since the DOS days that '*' is the wildcard character for specifying something like "get anything that has the word". However, in WQL, '%' is the wild card character when using LIKE keyword. There are also other meta characters such as [ ], ^, and _ that we can use with LIKE operator.

Here are some examples of how we use these additional meta characters.

```
$query = "SELECT * FROM Win32_Service WHERE Name LIKE '[af]%'"
Get-WMIObject -Query $query
```

The above query gets us all services with a name that starts with 'a' or 'f'. The way we use '[ ]' is very similar to its usage in regular expressions. This is used to specify a range of characters.

In case you need all services that start with any letter from 'a' to 'f', we still use '[ ]' meta character but specify the range. Here is how we do it:

```
$query = "SELECT * FROM Win32_Service WHERE Name LIKE '[a=f]%'"
Get-WMIObject -Query $query
```

> **Note**
> You can either use '[a=f]%' or '[a-f]%' in the above query. Although, MSDN documentation specifies only '='; the '-' character has the same meaning.

Let us look at another meta character, '^'.

```
$query = "SELECT * FROM Win32_Service WHERE Name LIKE '[^afgh]%'"
Get-WMIObject -Query $query
```

The above query gets us only the services with a name that does not start with 'a' or 'f' or 'g' or 'h'. In the above query, by using '^', we specify that we want to list all the services with names **not** starting with the characters in the range.

The last meta character is '_' and it matches exactly one character in the specified position similar to '?' in DOS. Here is how we use it.

```
$query = "SELECT * FROM Win32_Service WHERE Name LIKE '%a_diosrv%'"
Get-WMIObject -Query $query
```

What we are trying in the above query is obvious. The '_' gets replaced by any character and the matching service will be listed in the output.

AND, OR, and NOT

We can test for multiple conditions inside the WHERE clause. Let us see how we can do that. For example, if we want to list all services in the 'running' state but the StartMode set to 'manual'

```
$query = "SELECT * FROM Win32_Service WHERE State='Running' AND
StartMode='Manual'"
Get-WMIObject -Query $query
```

Simple! By using AND, we specify that we need both conditions to evaluate to TRUE. What if we want to list all services that are in 'running' state and StartMode set to 'manual' and the service name starts with the characters 'a' or 'f'. We have seen the first part of this query already. From the description of the problem, we know that we need to use another AND operator. Here is how the query will look like:

```
$query = "SELECT * FROM Win32_Service
                WHERE (State='Running' AND StartMode='Manual')
                AND (Name LIKE '[af]%')"
Get-WMIObject -Query $query
```

See how we did that. Enclosing the conditions in parentheses is not very important, at least in this query. But, this is a good practice to express a query like that. In the later sections, we will see where using parentheses is necessary.

Now, let us look at OR operator. By using OR, we specify that we want to get a list of instances whenever one of more conditions evaluate to TRUE. Let us see an example:

```
$query = "SELECT * FROM Win32_Service WHERE State='Stopped' OR
State='Paused'"
Get-WMIObject -Query $query
```

In this preceding example, we are retrieving all instances of *Win32_Service* where the service state is either 'stopped' or 'paused'. Now, let us make it a bit more complex by adding one more condition. Let us get a list of all services that are either in 'running' or 'paused' state AND with name that starts with either 'a' or 'f'. If the problem description is not clear, read it again. Now, run the below query and see what you get in the output.

```
$query = "SELECT * FROM Win32_Service WHERE State='Running' OR State='Paused'
AND Name LIKE '[af]%'"
Get-WMIObject -Query $query
```

So, what was there in output? All instances of running services irrespective of the service name condition we wanted to check. Why did that happen? Because, when using OR, it is just enough to evaluate any one condition to TRUE. So, whenever service state was 'running', this query returned the instance of the class. It did not really bother to look at other conditions. How do we make this work to get only what we want?

Here is how we do it:

```
$query = "SELECT * FROM Win32_Service WHERE (State='Running' OR
State='Paused') AND Name LIKE '[af]%'"
Get-WMIObject -Query $query
```

See how we combined the first two conditions. This is where parentheses are useful.

Let us now see how we can use NOT operator. Here is a very simple example and probably does not require any explanation.

```
$query = "SELECT * FROM Win32_Service WHERE NOT (State='Running')"
Get-WMIObject -Query $query
```

The same thing can be expressed using the other operators such as '<>' and '!=' also.

```
$query = "SELECT * FROM Win32_Service WHERE State<>'Running'"
Get-WMIObject -Query $query
```

```
$query = "SELECT * FROM Win32_Service WHERE State!='Running'"
Get-WMIObject -Query $query
```

All three preceding examples have the same meaning. They are just more than one way to get the same result.

IS, IS NOT, and NULL

You can also use IS and IS NOT operators within WHERE clause. However, the query will be valid only if the constant is NULL. For example, the following query

```
$query = "SELECT * FROM Win32_LogicalDisk WHERE FileSystem IS NULL"
Get-WMIObject -Query $query
```

…is valid and returns the disk drive information with no file system information. On my system, this query returns the DVD drive information.

> **Note**
> Some properties of the WMI classes may not be displayed when using default output formatting. For example, when we just query using "Select * From Win32_LogicalDisk", we won't see FileSystem property. To see all the properties, you need to pipe the output of *Get-WMIObject* cmdlet to Format-List *.

The following example,

```
$query = "SELECT * FROM Win32_LogicalDisk WHERE DriveType IS 5"
Get-WMIObject -Query $query
```

…will result in an invalid query error since the constant we specified is not NULL.

Using the IS NOT operator combination is similar to the first example using the IS operator.

```
$query = "SELECT * FROM Win32_LogicalDisk WHERE FileSystem IS NOT NULL"
Get-WMIObject -Query $query
```

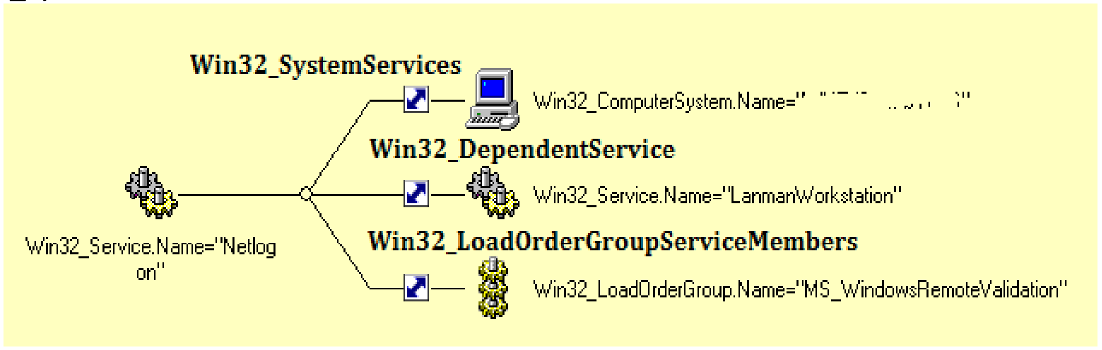
ASSOCIATORS OF

As we saw in the previous section, SELECT queries can be used to retrieve instances of a WMI class. But SELECT queries are not the only way to query for instances. We can also use the 'ASSOCIATORS OF' keyword to do the same. However, there is a difference. SELECT queries always return a collection of instances of a WMI class whereas "ASSOCIATORS OF" returns a collection of WMI objects that belong to different WMI classes or associated WMI classes. Before we dig too much into this, let us first understand, **what are these associated WMI classes?**

Take an example of a Windows service. WMI has several classes that represent windows service information. Let us look at *Win32_Service* and explore how this class is associated with other classes in WMI. We shall use CIM Studio for this purpose. To see this,

- Open CIM Studio from WMI Tools.
- Connect to root\cimv2 namespace.
- Double-click on *Win32_Service* class; retrieve all instances by clicking on icon.
- In the results window, right-click on "Netlogon" instance and select "Go to Object".
- Now, select the "Associations" Tab.

The below screen capture shows the associations. If we mouse-over the icon, we can see a class name that relates *Win32_Service* to *Win32_ComputerSystem*. This class, *Win32_SystemServices*, is called an association class.


Win32_SystemServices
Win32_ComputerSystem.Name="
Win32_DependentService
Win32_Service.Name="LanmanWorkstation"
Win32_Service.Name="Netlogon"
Win32_LoadOrderGroupServiceMembers
Win32_LoadOrderGroup.Name="MS_WindowsRemoteValidation"

The Netlogon service depends on LanmanWorkstation service. This relation is indicated by another association class *Win32_DependentService*. And, finally, the third association class – *Win32_LoadOrderGroupServiceMembers* – relates a load order group and a base service.

In the above example, Win32_Service.Name="NetLogon" is the source instance while instances of Win32_ComputerSystem, Win32_Service.Name="LanmanWorkstation", and Win32_LoadOrderGroup.Name="MS_WindowsRemoteValidation" are the target instances or endpoints.

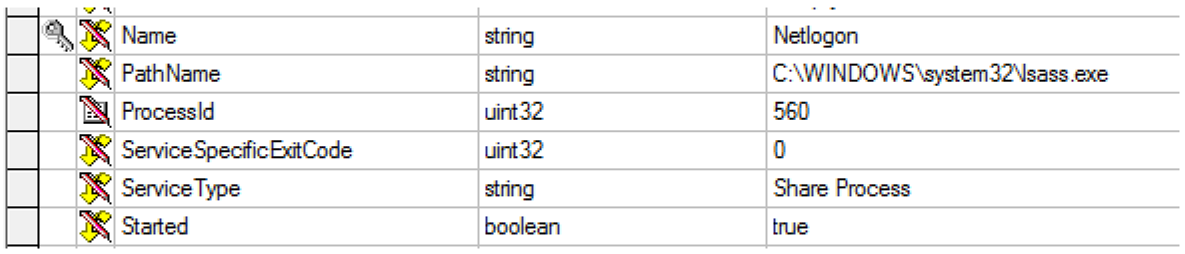
Now, using the "ASSOCIATORS OF" keyword, we can retrieve all instances that are associated with a particular source instance. Here is the general syntax of this keyword:

ASSOCIATORS OF {ObjectPath}

Note that the brackets are part of the syntax. Any valid ObjectPath can be used for ObjectPath. Let us look at an example to understand this.

```
$query = "ASSOCIATORS OF {Win32_Service.Name='NetLogon'}"
Get-WMIObject -Query $query
```

The above snippet shows the basic usage of 'ASSOCIATORS OF'. Make a note of the syntax inside curly brackets. This query — when executed — gets all the instances of all associated classes. So, this can take a while and the output can be overwhelming. Remember, this query without Name='NetLogon' will not return anything. This is required and the property – 'Name' in this case – should be the "key". You can identify the property that is designated as key by looking at properties list in WMI CIM Studio. This property uniquely identifies the WMI class instance.



🔑	Name	string	Netlogon
	PathName	string	C:\WINDOWS\system32\lsass.exe
	ProcessId	uint32	560
	ServiceSpecificExitCode	uint32	0
	ServiceType	string	Share Process
	Started	boolean	true

Look at the 🔑 icon next to the property 'Name'. This identifies the key property.
Similar to the data queries using SELECT keyword, the association queries can also use WHERE clause. However, the usage of WHERE clause is a bit different from how you do that with SELECT queries. There are predefined keywords that you can use with WHERE clause. They are:

ASSOCIATORS OF {ObjectPath} WHERE
AssocClass = AssocClassName
ClassDefsOnly
RequiredAssocQualifier = QualifierName
RequiredQualifier = QualifierName
ResultClass = ClassName
ResultRole = PropertyName
Role = PropertyName

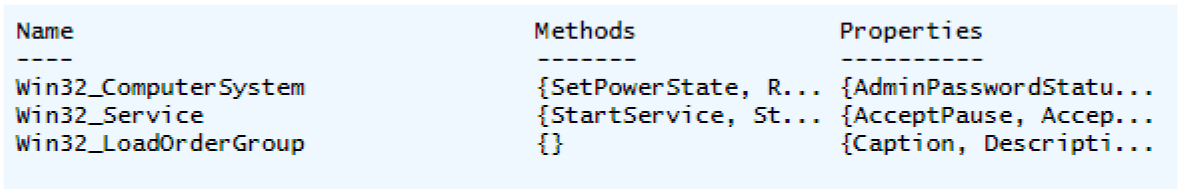
Note
You cannot use logical operators such as AND, OR, and NOT within the WHERE clause while using ASSOCIATORS OF keyword. You can use more than one keyword by just separating them by a space.

Let us see the examples around how to use these keywords within WHERE clause.

ClassDefsOnly

Let us first see a way to list only the associated class names as shown in the screenshot above. You can use the keyword ClassDefsOnly for this purpose.

```
$query = "ASSOCIATORS OF {Win32_Service.Name='NetLogon'} WHERE ClassDefsOnly"
Get-WMIObject -Query $query
```



```
Name                                    Methods              Properties
----                                    -------              ----------
Win32_ComputerSystem                    {SetPowerState, R... {AdminPasswordStatu...
Win32_Service                           {StartService, St... {AcceptPause, Accep...
Win32_LoadOrderGroup                    {}                   {Caption, Descripti...
```

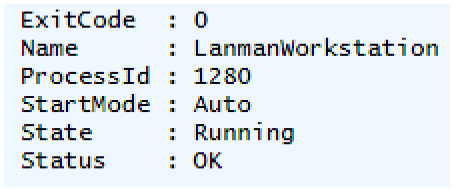
As we see, the list of associated class names we see here is the same as what we saw in the CIM Studio output.

AssocClass

The AssocClass keyword indicates that the returned endpoints must be associated with the source through the specified class or one of its derived classes. If you want to retrieve the instance of associated class thorugh a single association class:

```
$query = "ASSOCIATORS OF {Win32_Service.Name='NetLogon'} WHERE
AssocClass=Win32_DependentService"
Get-WMIObject -Query $query
```

This query gets us the endpoint instance associated through *Win32_DependentService* association class. This is shown in the below screen capture.



```
ExitCode  : 0
Name      : LanmanWorkstation
ProcessId : 1280
StartMode : Auto
State     : Running
Status    : OK
```

ResultClass

This keyword indicates that you want to retrieve the endpoints associated only with the specified ResultClass. For example,

```
$query = "ASSOCIATORS OF {Win32_Service.Name='NetLogon'} WHERE
ResultClass=Win32_LoadOrderGroup"
Get-WMIObject -Query $query
```

ResultRole

The ResultRole keyword indicates that the returned endpoints must play a particular role in their association with the source object.

For a moment think that Get-Service cmdlet never existed. Now, if you want to get a list of services that depend on a specific service without using “ASSOCIATORS OF”, we would do that by parsing Win32_DependentService instances. However, using ‘ASSOCIATORS OF’, we can do the same by:

```
$query = "ASSOCIATORS OF {Win32_Service.Name='LanmanWorkstation'} WHERE
ResultRole=Dependent"
Get-WMIObject -Query $query
```

The above query will list all the services that depend on LanmanWorkstation service. And, this is similar to:

```
Get-Service -Name "LanmanWorkstation" -DependentServices
```

Similarly, to get a list of services that a specific service depends on:

```
$query = "ASSOCIATORS OF {Win32_Service.Name='NetLogon'} WHERE
ResultRole=Antecedent"
Get-WMIObject -Query $query
```

This example will list LanmanWorkstation as NetLogon service depends on this service to start. And, this is similar to running:

```
Get-Service -Name "Netlogon" -RequiredServices
```

So, what are these dependent and antecedent roles we used in the above queries? Let us slow down a bit and understand the WMI relationships.

We can classify WMI relationships into three basic types:

- Dependency relationship
- Element-Setting relationship
- Component relationship

On a related note, MSDN documentation for several WMI classes lists properties such as Dependent, antecedent, Element, setting, Group Component, and part component. We shall explore the meaning of these in the coming sections.

Dependency Relationship

This relationship defines an association where one object is ***dependent*** on another object – ***antecedent***. The association class we just used – *Win32_DependentService* – is a classic example. This class defines a dependency relationship between two Windows services. From the above examples, the Win32_Service.Name="NetLogon" is dependent on Win32_Service.Name="LanmanWorkstation" – which is the antecedent. Now, go back to the examples and read them again.

Another example is the *Win32_DiskDriveToDiskPartition* association class. This class relates *Win32_DiskDrive* and *Win32_DiskPartition* classes where *Win32_DiskDrive* plays the antecedent and *Win32_DiskPartition* plays the dependent role. In simple English, a partition is dependent on a disk drive and by specifying one of these roles – dependent or antecedent; we can get either class instances – *Win32_DiskDrive* or *Win32_DiskPartition*, respectively.

So, if we want to query for all disk partitions on a specific hard drive:

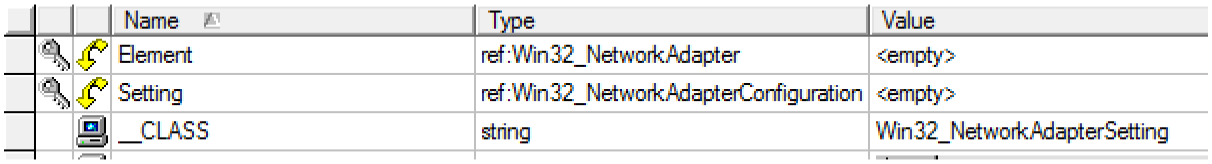
```
$query = "ASSOCIATORS OF {Win32_DiskDrive.DeviceID='\\.\PHYSICALDRIVE0'}
WHERE ResultRole=Dependent"
Get-WMIObject -Query $query
```

Here is what I see on my system:

```
NumberOfBlocks    : 976769024
BootPartition     : True
Name              : Disk #0, Partition #0
PrimaryPartition  : True
Size              : 500105740288
Index             : 0
```

Element-Setting Relationship

This relationship defines the association between an element – for example a network adapter – and the settings for that element. These association classes have properties named Element and Setting. For example, take a look at *Win32_NetworkAdapterSetting* class. The following screen capture from CIM studio shows the two properties of this association class.



Name	Type	Value
Element	ref:Win32_NetworkAdapter	<empty>
Setting	ref:Win32_NetworkAdapterConfiguration	<empty>
__CLASS	string	Win32_NetworkAdapterSetting

Win32_NetworkAdapterSetting association class defines a relationship between *Win32_NetworkAdapter* and *Win32_NetworkAdapterConfiguration* classes. In this relationship, *Win32_NetworkAdapter* is the Element and *Win32_NetworkAdapterConfiguration is the Setting*. **How do we use this?** Let us look at a few examples. We can retrieve the network adapter information by using the following query:

```
$query = "ASSOCIATORS OF {Win32_NetworkAdapterConfiguration.Index=12} WHERE
ResultRole=Element"
Get-WMIObject -Query $query
```

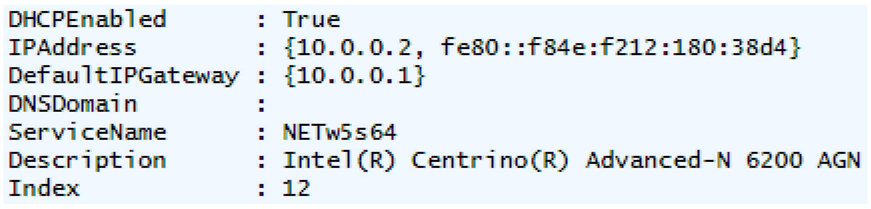
```
ServiceName       : NETw5s64
MACAddress        : 00:27:10:91:D5:BC
AdapterType       : Ethernet 802.3
DeviceID          : 12
Name              : Intel(R) Centrino(R) Advanced-N 6200 AGN
NetworkAddresses  :
Speed             : 54000000
```

Similarly, by querying Win32_NetworkAdapter class and for ‘setting’ role, we get the network adapter configuration information.

Note

In the above query, I used network adapter index 12. This may or may not exist on your system. Use Get-WMIObject –Class Win32_NetworkAdapter to find the index value for your network adapters.

```
$query = "ASSOCIATORS OF {Win32_NetworkAdapter.DeviceID=12} WHERE
ResultRole=Setting"
Get-WMIObject -Query $query
```



```
DHCPEnabled      : True
IPAddress        : {10.0.0.2, fe80::f84e:f212:180:38d4}
DefaultIPGateway : {10.0.0.1}
DNSDomain        :
ServiceName      : NETw5s64
Description      : Intel(R) Centrino(R) Advanced-N 6200 AGN
Index            : 12
```

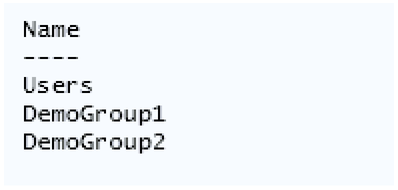
Component relationship

This WMI relationship contains two or more classes as GroupComponent and PartComponent. Let us understand this with the help of an example. The Win32_GroupUser WMI class associates *Win32_Group* (GroupComponent) and *Win32_Account* (PartComponent) classes. *Win32_GroupUser* defines a relationship between a group and an account that is a member of the group.

Let us see a few examples now.

We can get a list of group memberships for any given user account using the following query:

```
$query = "ASSOCIATORS OF {Win32_Account.Name='DemoUser1',Domain='DomainName'}
WHERE ResultRole=GroupComponent ResultClass=Win32_Account"
Get-WMIObject -Query $query | Select Name
```



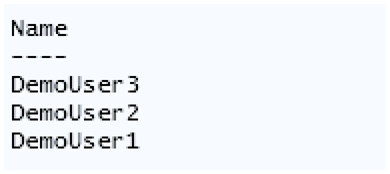
```
Name
----
Users
DemoGroup1
DemoGroup2
```

In the above query, we list all the groups in which DemoUser1 is a member. In the ObjectPath, we used both Name and Domain properties. This is required as both of these properties are the key properties in *Win32_Account* WMI class. We will get an invalid ObjectPath error if one of the properties is not specified. Also, make a note that the value of Domain property can be either a domain name or the local computer name.

We were also limiting the ResultClass to *Win32_Account*. This helps us in getting only the group details. Otherwise, we may end up seeing computer system details also as a part of the query output.

Similarly, if we want to get a list of all users in a group:

```
$query = "ASSOCIATORS OF {Win32_Account.Name='DemoGroup2',Domain='DomainName'}
WHERE ResultRole=PartComponent ResultClass=Win32_Account"
Get-WMIObject -Query $query | Select Name
```



```
Name
----
DemoUser3
DemoUser2
DemoUser1
```

As shown in the screen capture above, by executing the query, we see a list of all users who are part of DemoGroup2.

That is about WMI relationships. Let us move on to other interesting aspects of using **ASSOCIATORS OF** keyword.

Role

The Role keyword indicates that the returned endpoints participate in an association with the source object where the source object plays a particular role.

The difference between this keyword and ResultRole keyword is: when using *ResultRole* keyword, we indicate that **the endpoint must play a particular role in association** with the source instance. When using *Role* keyword -- we specify that the endpoints participate in an association **where the source object plays a particular role**.

Now, let us see some examples to understand this. We can just tweak the query that we used to list all members of a given group to:

```
$query = "ASSOCIATORS OF {Win32_Account.Name='DemoGroup2',Domain='DomainName'}
WHERE Role=GroupComponent ResultClass=Win32_Account"
Get-WMIObject -Query $query | Select Name
```

The result of both these queries is same. They just list all members of DemoGroup2. However, observe the difference between two queries. In the second variation, using *Role* keyword, we queried for all instances of Win32_Account.Name='DemoGroup2' where *Win32_Account* is a GroupComponent.

RequiredQualifier and RequiredAssocQualifier

There are other keywords such as RequiredQualifier and RequireAssocQualifier.

> **RequiredQualifier** keyword indicates that the returned endpoints associated with the source object must include the specified qualifier. The **RequiredAssocQualifier** keyword indicates that the returned endpoints must be associated with the source object through an association class that includes the specified qualifier.

WMI qualifiers are values that provide additional information about classes, associations, indications, methods, method parameters, triggers, instances, properties, or references. These qualifiers[5] can be used to narrow the result set from the WMI query. There is no in-depth discussion of these two keywords here.

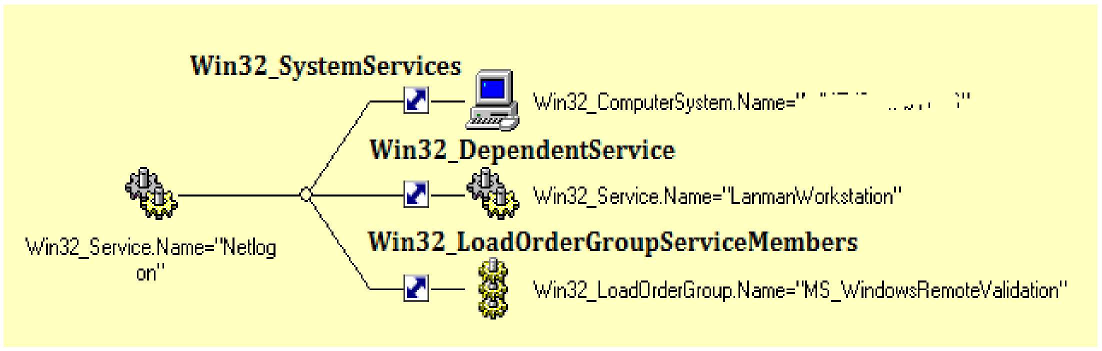
This brings us to the end of discussion on “ASSOCIATORS OF” keyword. Let us now move on to the final keyword used in WMI data queries.

REFERENCES OF

This keyword retrieves all association instances that refer to a particular source instance. The REFERENCES OF statement is similar to the ASSOCIATORS OF statement in its syntax.

REFERENCES OF {ObjectPath}

However, rather than retrieving endpoint instances, it retrieves the intervening association instances. Let us re-visit an earlier example to understand this.


Win32_SystemServices
Win32_ComputerSystem.Name="
Win32_DependentService
Win32_Service.Name="LanmanWorkstation"
Win32_Service.Name="Netlogon"
Win32_LoadOrderGroupServiceMembers
Win32_LoadOrderGroup.Name="MS_WindowsRemoteValidation"

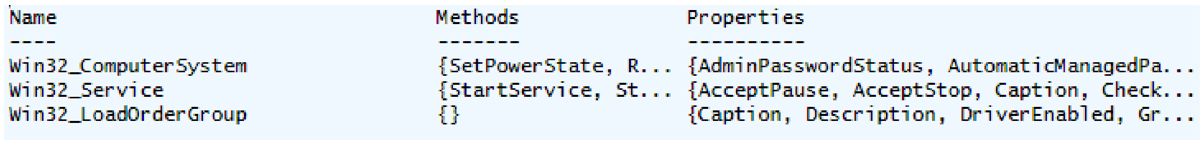
In the above screen capture from CIM Studio, *Win32_SystemServices*, *Win32_DependentService*, and *Win32_LoadOrderGroupServiceMembers* are the association classes. When using “REFERENCES OF” keyword alone, we retrieve the instances of these association classes whereas using “ASSOCIATORS OF” keyword, we retrieve the instances of endpoints – *Win32_ComputerSystem*, *Win32_Service*, and *Win32_LoadOrderGroup*. Let us see both keywords side by side in an example:

```
$query = "REFERENCES OF {Win32_Service.Name='Netlogon'} WHERE ClassDefsOnly"
Get-WMIObject -Query $query
```

```
Name                                Methods              Properties
----                                -------              ----------
Win32_SystemServices                {}                   {GroupComponent, Pa...
Win32_DependentService              {}                   {Antecedent, Depend...
Win32_LoadOrderGroupServiceMembers  {}                   {GroupComponent, Pa...
```

```
$query = "ASSOCIATORS OF {Win32_Service.Name='Netlogon'} WHERE ClassDefsOnly"
Get-WMIObject -Query $query
```

[5] WMI Standard Qualifiers: http://msdn.microsoft.com/en-us/library/aa393651(v=vs.85).aspx



```
Name                         Methods              Properties
----                         -------              ----------
Win32_ComputerSystem         {SetPowerState, R... {AdminPasswordStatus, AutomaticManagedPa...
Win32_Service                {StartService, St... {AcceptPause, AcceptStop, Caption, Check...
Win32_LoadOrderGroup         {}                   {Caption, Description, DriverEnabled, Gr...
```

You see the difference. By using the **ClassDefsOnly** keyword, we limit the verbosity of the output and this helps us understand the difference.

Similar to ASSOCIATORS OF keyword, you can use the WHERE clause with “REFERENCES OF” keyword also. There are predefined keywords that you can use with WHERE clause. The general syntax for using WHERE clause is:

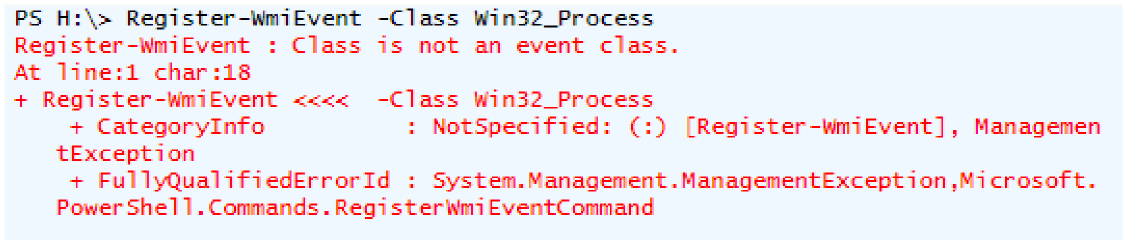
REFERENCES OF {ObjectPath} WHERE
ClassDefsOnly
RequiredQualifier = QualifierName
ResultClass = ClassName
Role = PropertyName

The usage of these keywords is similar to “ASSOCIATORS OF”.
This concludes this chapter on WMI data queries. We looked at several keywords and operators in this chapter. In the next chapter, we shall look at how to use WMI query language for WMI event queries.

WMI Event Queries: Introduction

Just as there is a WMI class that represents each type of system resource that can be managed using WMI, there is a WMI class that represents each type of WMI event. When an event that can be monitored by WMI occurs, an instance of the corresponding WMI event class is created. A WMI event occurs when that instance is created.

Windows PowerShell v2 provides a cmdlet — *Register-WMIEvent* – to consume WMI events. There are a couple of different ways we can use this cmdlet. You can either use -*Class* or -*Query* parameters to create a temporary[6] event consumer. When using -*Class* parameter, we need to specify a WMI event class. So, what happens if the value provided to the -*Class* parameter isn't a WMI event class? Yes, PowerShell complains about it.



```
PS H:\> Register-WmiEvent -Class Win32_Process
Register-WmiEvent : Class is not an event class.
At line:1 char:18
+ Register-WmiEvent <<<<  -Class Win32_Process
    + CategoryInfo          : NotSpecified: (:) [Register-WmiEvent], Managemen
   tException
    + FullyQualifiedErrorId : System.Management.ManagementException,Microsoft.
   PowerShell.Commands.RegisterWmiEventCommand
```

So, from the above error, we understand that *Win32_Process* isn't a WMI event class. But, how do we find what are the WMI event classes?

```
#Get all classes that are WMI event classes
#filter class names for Win32 classes
Get-WMIObject -Query "SELECT * FROM meta_class WHERE (__This ISA '__Event')
AND (__Class like 'win32%')"
```

This will list all WMI event classes that start with Win32 prefix. We will see many more if we remove the second condition in the WHERE clause but for starters, this is good enough. *Win32_ProcessStartTrace* is one of the WMI event classes listed when we execute the above PowerShell command. This class indicates the new process started event. We can use this WMI class to subscribe to all process start events. For example,

```
Register-WMIEvent -Class Win32_ProcessStartTrace -SourceIdentifier "Process
Started" -Action {
                     Write-Host "$($Event.SourceEventArgs.NewEvent.ProcessName)
                     just started"
            }
```

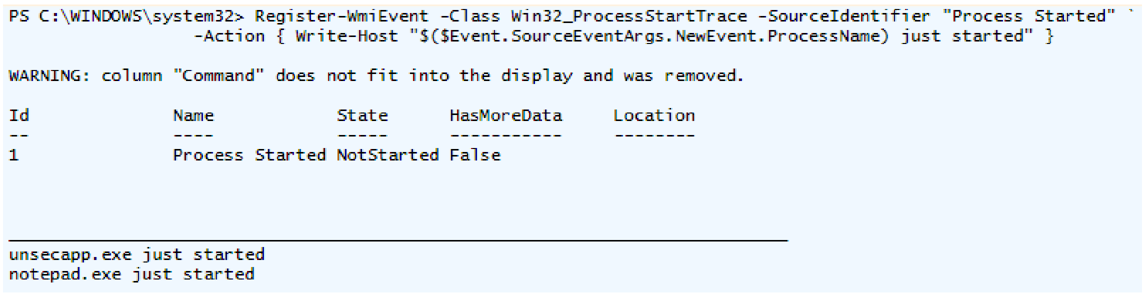
Note

-*Action* parameter can be used to specify the scriptblock used to take some action when the event gets fired. Alternatively, you can use *Wait-Event* and *Get-Event* cmdlets to receive the events.

[6] Event Consumers: http://msdn.microsoft.com/en-us/library/aa393013(VS.85).aspx#event_consumers

Note
You have to open the PowerShell console in elevated mode. Else, we will see an access denied message every time we try using *Register-WMIEvent*.

This command will register an event consumer and display a message with the newly created process name. However, this will result in receiving the messages at the console every time any process starts and not just the one we are interested in.

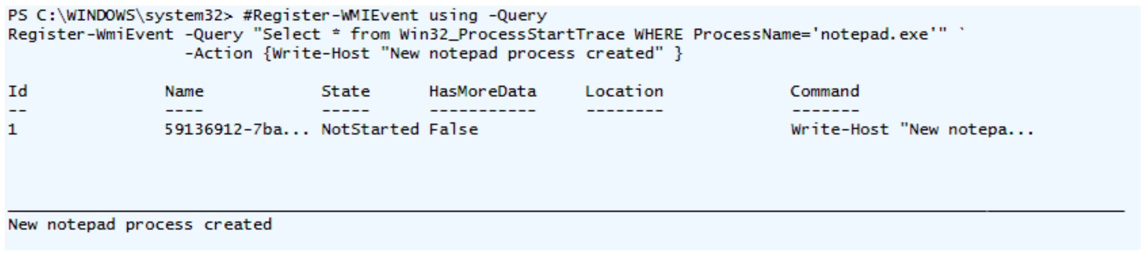

```
PS C:\WINDOWS\system32> Register-WmiEvent -Class Win32_ProcessStartTrace -SourceIdentifier "Process Started" `
                     -Action { Write-Host "$($Event.SourceEventArgs.NewEvent.ProcessName) just started" }

WARNING: column "Command" does not fit into the display and was removed.

Id              Name            State      HasMoreData     Location
--              ----            -----      -----------     --------
1               Process Started NotStarted False


unsecapp.exe just started
notepad.exe just started
```

So, what if we are interested only in one specific process? We could have easily filtered out the unnecessary processes before displaying the process name at the console. But, why even receive the event when we don't need it? This is where *-Query* parameter comes handy. Look at this example:

```
#Register-WMIEvent using -Query
Register-WMIEvent -Query "SELECT * FROM Win32_ProcessStartTrace WHERE
ProcessName='notepad.exe'" -Action {
                        Write-Host "New notepad process created"
}
```

Note
It is possible to register for a remote WMI event using *Register-WMIEvent* cmdlet and the *-ComputerName* parameter. This book, however, looks at the local WMI events only.

The WQL statement we used should be familiar to you by now. This query can be used to narrow down the events to just the ones we need.


```
PS C:\WINDOWS\system32> #Register-WMIEvent using -Query
Register-WmiEvent -Query "Select * from Win32_ProcessStartTrace WHERE ProcessName='notepad.exe'" `
                  -Action {Write-Host "New notepad process created" }

Id              Name            State      HasMoreData     Location             Command
--              ----            -----      -----------     --------             -------
1               59136912-7ba... NotStarted False                                Write-Host "New notepa...


New notepad process created
```

There are many other ways to monitor process creation using WMI events and WQL. What we just saw is one way of doing it. We shall see the other methods in the later sections.

In the earlier section, we saw how to list WMI event classes and how to use *Register-WMIEvent* to subscribe to the events from that WMI class. In the case of a WMI event class, the class itself provides the events. Hence, we can use the class name directly and create an event subscriber.

In the case of non-event classes, WMI does all the hard lifting of monitoring the class instances and providing whenever an instance is created, modified, or deleted. Now, how do we subscribe to the events from these non-event classes? This question brings us to the discussion on types of event queries.

Event Query Types

There are three types of event queries in WMI: intrinsic events, extrinsic events, and timer events. Here is a brief discussion on each of these event types:

Intrinsic Events

Intrinsic events are used to monitor a resource represented by a class in the CIM repository. In other words, the intrinsic events occur in response to a change in the standard WMI data model. WMI creates intrinsic events for objects stored in the WMI repository. A provider generates intrinsic events for dynamic classes, but WMI can create an instance for a dynamic class if no provider is available. WMI uses polling to detect the changes. There are many system classes that WMI uses to report intrinsic events. However, the ones that are most interesting and useful are *__InstanceCreationEvent*, *__InstanceModificationEvent*, and *__InstanceDeletionEvent*. Hence, monitoring resources on a system involves monitoring of these system classes. The later chapters in this book dig deep in to the intrinsic events and show examples of subscribing to these events.

Extrinsic Events

Extrinsic events represent events that do not directly link to standard WMI model. For example, Windows registry defines extrinsic events for all registry change events. For intrinsic events, having a WMI provider isn't mandatory. This is mostly because they are defined within the standard WMI model and WMI takes care of these if there is no WMI provider for a given resource in the standard WMI model. However, since extrinsic events are outside of the standard WMI model, having a WMI provider is mandatory. The later sections of this book will provide an in-depth discussion on the Windows registry events and show a few examples.

Timer Events

Timer events are a specialized kind of intrinsic event. WMI uses preconfigured event timers within the repository to generate timer events. In the pre Windows 2003 days, we had to use *__AbsoluteTimerInstruction*[7] and *__IntervalTimerInstruction*[8] to create absolute and interval timer instructions, respectively. However, with the Win32_LocalTime class, this isn't necessary any more. We shall see how to use this class to create timer events and examples of how to use the timer events to create complex scheduling tasks in the later parts of this book.

[7] __AbsoluteTimerInstruction class: http://msdn.microsoft.com/en-us/library/aa394620(v=vs.85).aspx
[8] __IntervalTimerInstruction class: http://msdn.microsoft.com/en-us/library/aa394654(v=VS.85).aspx

Before we dig into the event types, it is important to understand the WMI query syntax used for the above query types. The WMI query syntax for event queries is a bit different and deserves a discussion.

WQL Syntax for event queries

As we discussed earlier, we use SELECT statement for event queries too. We can combine this with other keywords such as WITHIN, HAVING, and GROUP to change how we receive these WMI events. Here is the general syntax for WQL event queries:

```
EVENT-WQL = "SELECT" <PROPERTY-LIST> "FROM" /
<EVENT-Class-NAME> <OPTIONAL-WITHIN> <EVENT-WHERE>

        OPTIONAL-WITHIN = ["WITHIN" <INTERVAL>]
        INTERVAL = 1*DIGIT
        EVENT-WHERE = ["WHERE" <EVENT-EXPR>]

        EVENT-EXPR = ( (<INSTANCE-STATE> "ISA" <CLASS-NAME> <EXPR2>) /
        <EXPR> )
        ["GROUP WITHIN" <INTERVAL>
        ( ["BY" [<INSTANCE-STATE> DOT] <PROPERTY-NAME>]
        ["HAVING" <EXPR>]] )

        INSTANCE-STATE = "TARGETINSTANCE" / "PREVIOUSINSTANCE"
```

In the above syntax specification, we know the SELECT, FROM, and WHERE keywords. There are other keywords such as WITHIN, GROUP, BY, and HAVING. Let us look at each one of these keywords now.

WITHIN

WITHIN keyword is used to specify the polling interval or grouping interval (when used with GROUP clause) for the events. A polling interval is the interval that WMI uses as the maximum amount of time that can pass before notification of an event must be delivered. The general syntax to specify the polling interval,

```
SELECT * FROM EventClass WITHIN interval WHERE property = value
```

The polling interval value is specified as number of seconds and is a floating point number. So, we can specify values smaller than one second. However, specifying a polling interval smaller than one second (for example, 0.1) may cause system slow down due to the resource intensive nature of event queries. The recommended values for the polling interval really depend on the event class. Never use a small value here unless you really need the event notification to be delivered immediately.

Let us look at an example:

```
#Build a WMI query for receiving an event
$query = "SELECT * FROM __instanceCreationEvent WITHIN 10 WHERE
TargetInstance ISA 'Win32_Process'"

#Register the event
Register-WMIEvent -Query $query -Action {Write-Host "New Process created"}
```

GROUP

Using GROUP clause causes WMI to generate a single notification to represent a group of events. When used in a WMI event query, this returns an instance of *__AggregateEvent* that contains an embedded object of one of the instances received during the grouping interval and the number of such events received. These two are represented by '*Representative'* & '*NumberOfEvents'* properties respectively.

The grouping interval specifies the time period, after receiving an initial event, during which WMI should collect similar events. The GROUP clause must contain a WITHIN clause to specify the grouping interval and can contain either the BY or the HAVING keyword, or both. The GROUP clause is placed after the WHERE clause if the query contains a WHERE clause. Here is the syntax:

SELECT * FROM EventClass [WHERE property = value] GROUP WITHIN interval

This is especially useful when we don't want to receive an event notification every time the event fires. For example, when monitoring a system, we may not want to receive a notification every time a specific event log gets generated. Instead, we can use GROUP clause to specify a grouping interval and receive only one notification for all the desired event logs generated within the grouping interval.

Take a look at this example:

```
#Build a WMI query for receiving an event
$query = "SELECT * FROM __instanceCreationEvent WHERE TargetInstance ISA
'Win32_NTLogEvent' AND TargetInstance.EventCode=1980 GROUP WITHIN 300"

#Register the event
Register-WMIEvent -Query $query -Action {Write-Host "Eventlog Arrived"}
```

This is different from how polling interval works. When using polling interval, WMI waits until the time specified is elapsed and sends all the events that were generated during the polling interval. However, in case of grouping interval, as mentioned above, one event will be received by the subscriber but the number of events occurred during the grouping interval will be known.

HAVING

In the above example, we will receive an event notification for all the events received within the grouping interval. But, what if we want to receive the event notification only when ten such event logs are generated within the grouping interval? This is where HAVING plays an important role. Let us look at how we can modify the above sample code to use the HAVING keyword.

```
#Build a WMI query for receiving an event
$query = "SELECT * FROM __instanceCreationEvent WHERE TargetInstance ISA
'Win32_NTLogEvent' AND TargetInstance.EventCode=1980 GROUP WITHIN 300 HAVING
NumberOfEvents > 10"

#Register the event
Register-WMIEvent -Query $query -Action {Write-Host "Eventlog Arrived"}
```

I mentioned earlier that by using GROUP returns a property called *NumberOfEvents* that contains the number of events received during the grouping interval. Now, we use that property along with HAVING keyword to filter event notifications. Here is the syntax:

SELECT * FROM EventClass [WHERE property = value]
GROUP WITHIN interval HAVING NumberOfEvents operator constant

So, using the HAVING query as shown above, an event notification is delivered only when WMI receives more than 10 events in the grouping interval.

BY

You can use the BY keyword along with the GROUP clause to group events by one or more properties of the event class. The general syntax for using BY is as follows:

SELECT * FROM EventClass [WHERE property = value]
GROUP WITHIN interval [BY property_list]

For example, the following example groups all events received in 300 seconds by the TargetInstance.SourceName property.

```
#Build a WMI query for receiving an event
$query = "SELECT * FROM __instanceCreationEvent WHERE TargetInstance ISA
'Win32_NTLogEvent' GROUP WITHIN 300 BY TargetInstance.SourceName
HAVING NumberOfEvents > 10"

#Register the event
Register-WMIEvent -Query $query -Action {Write-Host "Eventlog Arrived"}
```

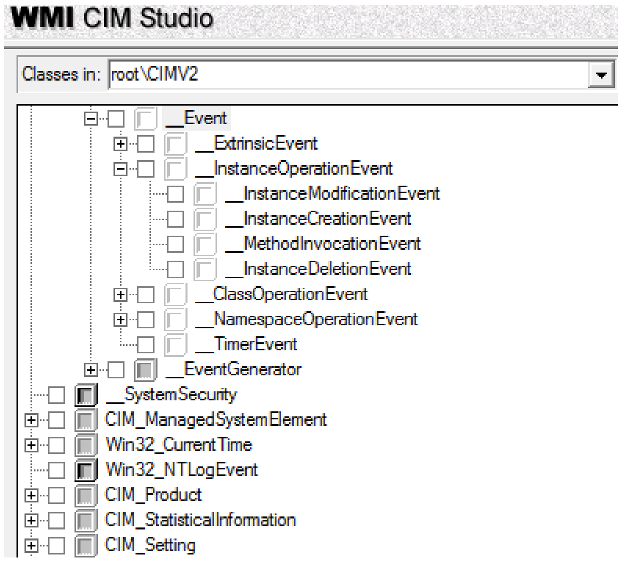
In the above example, the events are grouped by the TargetInstance.SourceName property and an event notification is delivered only if the number of events received during the grouping interval exceeds 10.

So far, we looked at all the important keywords that can be used while building WMI event queries. But, if you have observed, I kept using the words ***EventClass*** and ***TargetInstance***. What are these? This is the topic of discussion in the next few chapters.

Intrinsic Event Queries

Intrinsic events are used to monitor a resource represented by a class in the CIM repository. In other words, the intrinsic events occur in response to a change in the standard WMI data model. WMI creates intrinsic events for objects stored in the WMI repository. A provider generates intrinsic events for dynamic classes, but WMI can create an instance for a dynamic class if no provider is available. WMI uses polling to detect the changes.

There are many system classes[9] that WMI uses to report intrinsic events. However, the ones that are most interesting and useful are *__InstanceCreationEvent*, *__InstanceModificationEvent*, and *__InstanceDeletionEvent*. Hence, monitoring resources on a system involves monitoring of these system classes. These classes are derived from the *__InstanceOperationEvent* class which is derived from the *__Event system* class under *root\Default* namespace. The following screen capture of WMI CIM Studio shows this hierarchy.



```
__Event
  __ExtrinsicEvent
  __InstanceOperationEvent
    __InstanceModificationEvent
    __InstanceCreationEvent
    __MethodInvocationEvent
    __InstanceDeletionEvent
  __ClassOperationEvent
  __NamespaceOperationEvent
  __TimerEvent
__EventGenerator
__SystemSecurity
CIM_ManagedSystemElement
Win32_CurrentTime
Win32_NTLogEvent
CIM_Product
CIM_StatisticalInformation
CIM_Setting
```

The general WQL syntax for WMI intrinsic event queries is:

```
SELECT Property_List FROM EventClass WITHIN PollingInterval
        WHERE TargetInstance | PreviousInstance ISA WMIClassName
        AND TargetInstance.WMIClassPropertyName = Value
```

This is something similar to what we saw in the earlier discussion about WQL syntax for event queries. The EventClass can be any of the system classes such as *__InstanceCreationEvent*, *__InstanceModificationEvent*, *__InstanceDeletionEvent*, and *__InstanceOperationEvent*.

[9] WMI System classes: http://msdn.microsoft.com/en-us/library/aa394583(v=vs.85).aspx

Note
The *PreviousInstance* and *TargetInstance* represent the state of the object prior and after the event respectively. However, make a note that the *PreviousInstance* property is available only when using __*InstanceModificationEvent* class.

So, when do we use each of these system classes mentioned above?

__InstanceCreationEvent

This class is used when we want to receive a notification upon creation of an instance. For example, we can use this event class when you want to receive an event notification every time a new process gets created. This can be done by,

```
#Query for new process events
$query = "SELECT * FROM __InstanceCreationEvent WITHIN 10 WHERE
TargetInstance ISA 'Win32_Process'"

#Register WMI event
Register-WMIEvent -Query $query -Action {Write-Host "New Process Created. Do
something useful here"}
```

__InstanceDeletionEvent

This class is used when we want to receive a notification upon deletion of an instance. For example, we can use this class when we want to receive an event notification every time a process is terminated. For example,

```
#Query for new process events
$query = "SELECT * FROM __InstanceDeletionEvent WITHIN 5 WHERE TargetInstance
ISA 'Win32_Process'"

#Register WMI event
Register-WMIEvent -Query $query -Action {Write-Host "A Process terminated. Do
something useful here"}
```

__InstanceModificationEvent

This class is used when we want to monitor changes to an existing instance or a resource. For example, we can use this class when we want to receive an event notification when the processor utilization on a system goes beyond a specified usage threshold. For example,

```
#Query for new process events
$query = "SELECT * FROM __InstanceModificationEvent WITHIN 5 WHERE
TargetInstance ISA 'Win32_Processor' AND TargetInstance.LoadPercentage > 80"

#Register WMI event
Register-WMIEvent -Query $query -Action {Write-Host "Processor utilization is
more than 80%. Do something useful here"}
```

Using the above query, we will know whenever the CPU load goes beyond 80%. Now, what if we want to receive the event only when the change in load percentage is around a specified range? This is where the *PreviousInstance* will play a role. As discussed earlier, the *TargetInstance* and *PreviousInstance* are the embedded objects of *__InstanceModificationEvent*. So, we will have access to both the instances.

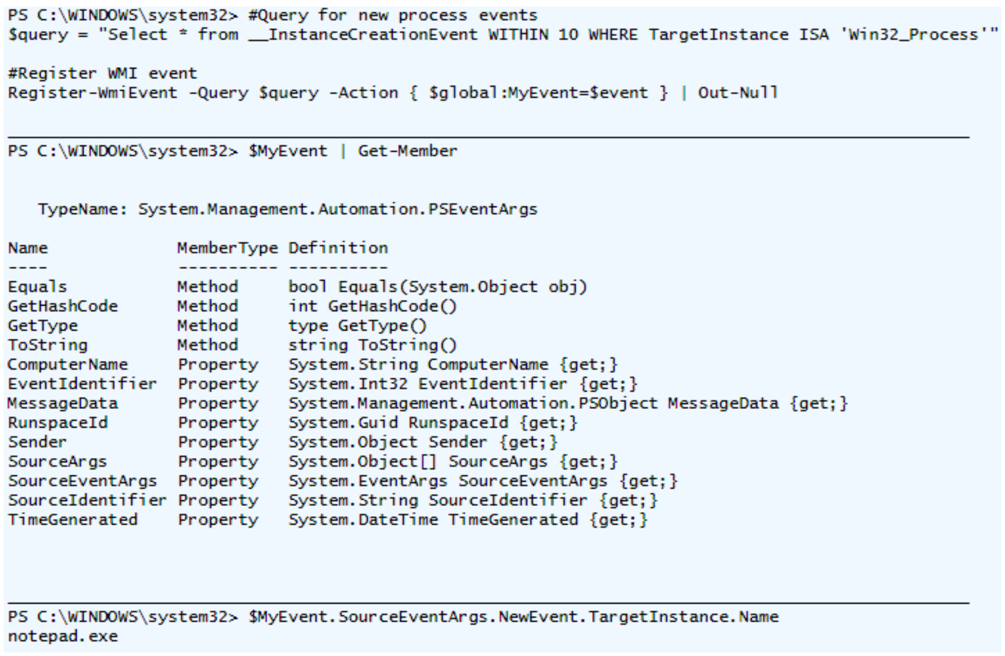
Now, going back to the example:
```
#Query for new process events
$query = "Select * FROM __InstanceModificationEvent WITHIN 5
            WHERE TargetInstance ISA 'Win32_Processor'
            AND (TargetInstance.LoadPercentage >= 80
            AND PreviousInstance.LoadPercentage >= 70)"

#Register WMI event
Register-WMIEvent -Query $query -Action {Write-Host "Processor utilization is
more than 80%. Do something useful here"}
```

All of the examples above just displayed a message when the event notification was received. Instead, we can do something useful within the script block.

For example, in the *__InstanceCreationEvent* example, we are just displaying that a new process was created but not the process name that just got created. So, how do we access that information in the script block and tell a user the name of the process that was created?

Simple, PowerShell creates an automatic variable called $event and stores the last event received in that variable. And, this automatic variable can be accessed in the *-Action* scriptblock you specify during a WMI event registration. Let us see an example:

```
#Query for new process events
$query = "SELECT * FROM __InstanceCreationEvent WITHIN 10 WHERE
TargetInstance ISA 'Win32_Process'"

#Register WMI event
Register-WMIEvent -Query $query -Action { $global:myEvent=$event } | Out-Null
```


```
PS C:\WINDOWS\system32> #Query for new process events
$query = "Select * from __InstanceCreationEvent WITHIN 10 WHERE TargetInstance ISA 'Win32_Process'"

#Register WMI event
Register-WmiEvent -Query $query -Action { $global:MyEvent=$event } | Out-Null

________________________________________________________________________________
PS C:\WINDOWS\system32> $MyEvent | Get-Member


   TypeName: System.Management.Automation.PSEventArgs

Name             MemberType Definition
----             ---------- ----------
Equals           Method     bool Equals(System.Object obj)
GetHashCode      Method     int GetHashCode()
GetType          Method     type GetType()
ToString         Method     string ToString()
ComputerName     Property   System.String ComputerName {get;}
EventIdentifier  Property   System.Int32 EventIdentifier {get;}
MessageData      Property   System.Management.Automation.PSObject MessageData {get;}
RunspaceId       Property   System.Guid RunspaceId {get;}
Sender           Property   System.Object Sender {get;}
SourceArgs       Property   System.Object[] SourceArgs {get;}
SourceEventArgs  Property   System.EventArgs SourceEventArgs {get;}
SourceIdentifier Property   System.String SourceIdentifier {get;}
TimeGenerated    Property   System.DateTime TimeGenerated {get;}



________________________________________________________________________________
PS C:\WINDOWS\system32> $MyEvent.SourceEventArgs.NewEvent.TargetInstance.Name
notepad.exe
```

If you see in the above example, we made an event registration for process creation events and in the *-Action* script block, assigned the $event variable to a variable in the global scope ($myEvent). This is essential because we cannot access the $event variable outside the -Action script block. Once the registration was done, I opened notepad application. This will fire the *__InstanceCreationEvent* and $myEvent should have the details around the event. So, I tried looking at all the members of this event. After exploring that a bit, I figured out that ***$myEvent.SourceEventArgs.NewEvent.TargetInstance.Name*** has the name of the new process which is notepad.exe. This is precisely what we see in the last line there.

$Event.SourceEventArgs.NewEvent.TargetInstance will have the instance of the newly created process. Remember, $event automatic variable won't be available outside of the *–action* Scriptblock. And, that is the reason we assigned it to global variable called $myEvent. Go explore[10] more about this.

As we have seen in the system class hierarchy, the above three classes are derived from *__InstanceOperationEvent* class. We can use this class as well for our event subscriptions. Here is how we do it:

```
$query = "SELECT * FROM __InstanceOperationEvent WITHIN 10 WHERE
TargetInstance ISA 'Win32_Process'"

Register-WMIEvent -Query $query -Action {
    $processName = $Event.SourceEventArgs.NewEvent.TargetInstance.Name
    Switch ($event.SourceEventArgs.NewEvent.__CLASS) {
        '__InstanceCreationEvent' {
                Write-Host "New Process created: $($processName)"
        }
```

[10] Eventing – Scout the Action block's Automatic variables: https://robertrobelo.wordpress.com/2010/10/15/eventing-scout-the-action-blocks-automatic-variables/

```
        '__InstanceDeletionEvent' {
                Write-Host "A Process terminated: $($processName)"
        }
        '__InstanceModificationEvent' {
                Write-Host "Process was modified: $($processName)"
        }
    }
}
```

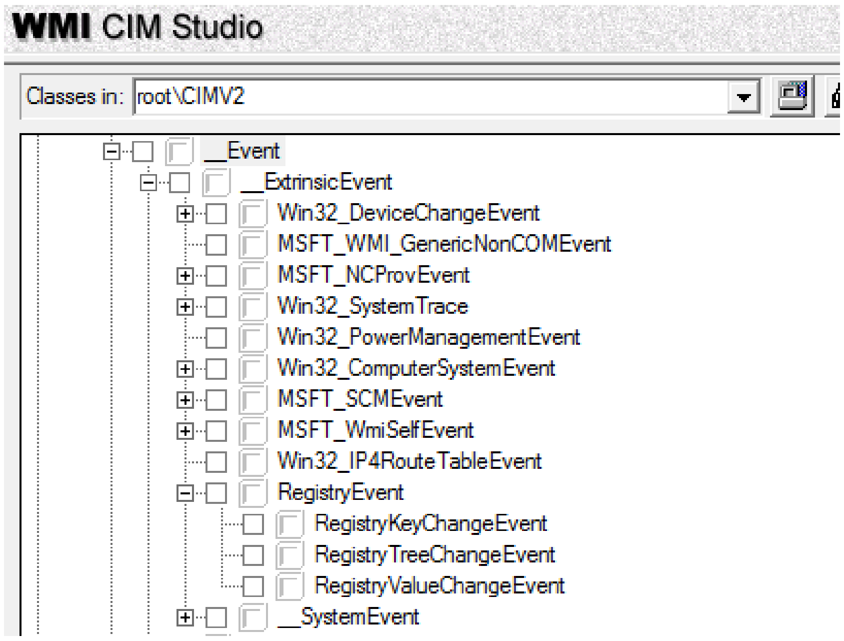
If we look at the *-Action* scriptblock, we are actually checking for the actual instance class type again using a switch statement. This is required because *__InstanceOpeartionEvent* can be any of the three types. It does not have an instance of its own. This example is to just show you how to use the *__InstanceOperationEvent* class*.* This, in fact, results in a flood of events because every time a process working set is changed, it triggers a *__InstanceModificationEvent*.

This is it about intrinsic events and how to access or subscribe to them. In the next chapter, we shall see what extrinsic events are and how events from Windows registry provider can be used to monitor changes to registry.

Extrinsic Event Queries

Extrinsic events represent events that do not directly link to standard WMI model. For example, Windows registry defines extrinsic events for all registry change events. For intrinsic events, having a WMI provider isn't mandatory. This is because they are defined within the standard WMI model and WMI takes care of these if there is no WMI provider for a given resource in the standard WMI model. However, since extrinsic events are outside of the standard WMI model, having a WMI provider is mandatory.

When creating a WMI query for extrinsic events, we need to use a class derived from __*ExtrinsicEvent* class. Take a look at this CIM Studio screen capture.



```
__Event
  __ExtrinsicEvent
    Win32_DeviceChangeEvent
    MSFT_WMI_GenericNonCOMEvent
    MSFT_NCProvEvent
    Win32_SystemTrace
    Win32_PowerManagementEvent
    Win32_ComputerSystemEvent
    MSFT_SCMEvent
    MSFT_WmiSelfEvent
    Win32_IP4RouteTableEvent
    RegistryEvent
      RegistryKeyChangeEvent
      RegistryTreeChangeEvent
      RegistryValueChangeEvent
    __SystemEvent
```

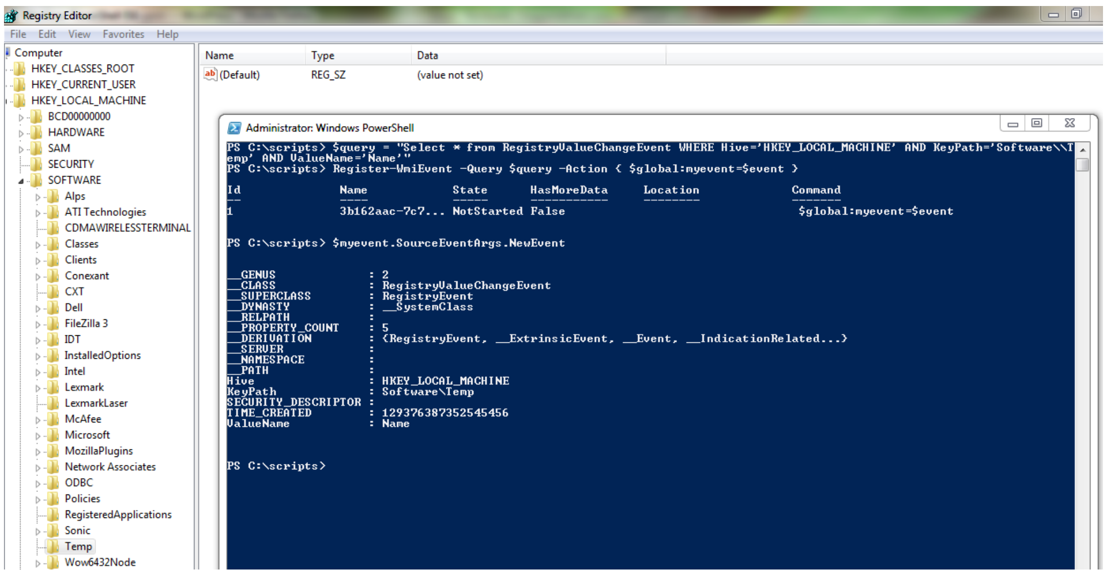
As you see, there is registry provider that provides extrinsic events. The subsequent sections in this chapter will use the extrinsic event classes from registry provider to show we can create event notifications for extrinsic events.

Monitoring registry value change events

We use ***RegistryValueChangeEvent*** to monitor changes to registry values. Here is how we write a WMI query for registry value change events:

```
$query = "SELECT * FROM RegistryValueChangeEvent WHERE
Hive='HKEY_LOCAL_MACHINE' AND KeyPath='Software\\Temp' AND ValueName='Name'"
Register-WMIEvent -Query $query -Action { Write-Host "Value changed" }
```

Yes, that is it. So, when the monitored value changes, we will see the message "value changed" on the screen. However, there is one drawback here. The event notification only tells us that the value has been modified but it won't return the new value! Also, deleting a value is also considered modification and we receive a notification. But, again, the notification won't tell us that the value was deleted. See the next screen capture.



```
PS C:\scripts> $query = "Select * from RegistryValueChangeEvent WHERE Hive='HKEY_LOCAL_MACHINE' AND KeyPath='Software\\T
emp' AND ValueName='Name'"
PS C:\scripts> Register-WmiEvent -Query $query -Action { $global:myevent=$event }

Id              Name            State      HasMoreData     Location             Command
--              ----            -----      -----------     --------             -------
1               3b162aac-7c7... NotStarted False                                 $global:myevent=$event


PS C:\scripts> $myevent.SourceEventArgs.NewEvent


__GENUS            : 2
__CLASS            : RegistryValueChangeEvent
__SUPERCLASS       : RegistryEvent
__DYNASTY          : __SystemClass
__RELPATH          :
__PROPERTY_COUNT   : 5
__DERIVATION       : {RegistryEvent, __ExtrinsicEvent, __Event, __IndicationRelated...}
__SERVER           :
__NAMESPACE        :
__PATH             :
Hive               : HKEY_LOCAL_MACHINE
KeyPath            : Software\Temp
SECURITY_DESCRIPTOR :
TIME_CREATED       : 129376387352545456
ValueName          : Name



PS C:\scripts>
```

Although we deleted the value, the resulting event notification does not tell us anything about that value deletion. So, in the -Action script block, we will have to verify the presence of registry value we are monitoring and then display the new value. For example,

```
$query = "SELECT * FROM RegistryValueChangeEvent WHERE
Hive='HKEY_LOCAL_MACHINE' AND KeyPath='Software\\Temp' AND ValueName='Name'"

Register-WMIEvent -Query $query -Action {
            If ((Get-item HKLM:\SOFTWARE\Temp).GetValue("Name")) {
                Write-host (Get-item HKLM:\SOFTWARE\Temp).GetValue("Name")
            } else {
                Write-host "The registry value was deleted"
            }
}
```

Monitoring registry key change events

RegistryKeyChangeEvent can be used to monitor modifications to a registry sub key. Similar to ***RegistryValueChangeEvent***, this event notification also won't give you any information beyond sub key modification. Here is how we use it:

```
$query = "SELECT * FROM RegistryKeyChangeEvent WHERE
Hive='HKEY_LOCAL_MACHINE' AND KeyPath='Software\\Temp'"

Register-WMIEvent -Query $query -Action {Write-host "Something changed"}
```

Once again, you need to use some technique similar to what is shown above to retrieve the "real" modification that happened.

Monitoring registry tree change events

RegistryTreeChangeEvent can be used to monitor sub-tree level modifications. Similar to other two event classes in the registry provider, this event class provides a notification that a change occurred but won't tell us about what had changed. So, we got to use a method of our own to detect what change generated the event notification.

```
$query = "SELECT * FROM RegistryTreeChangeEvent WHERE
Hive='HKEY_LOCAL_MACHINE' AND RootPath=''"

Register-WMIEvent -Query $query -Action { Write-host "Something changed" }
```

There are many vendor provided extrinsic event classes.

For example, take a look at how Intel ME WMI provider uses extrinsic events for firmware state notifications:
http://software.intel.com/sites/manageability/AMT_Implementation_and_Reference_Guide/default.htm?turl=WordDocuments%2Fwmievents.htm

This is it about extrinsic events in WMI. Go and explore these using the examples we used throughout. We shall look at WMI timer events in the next chapter.

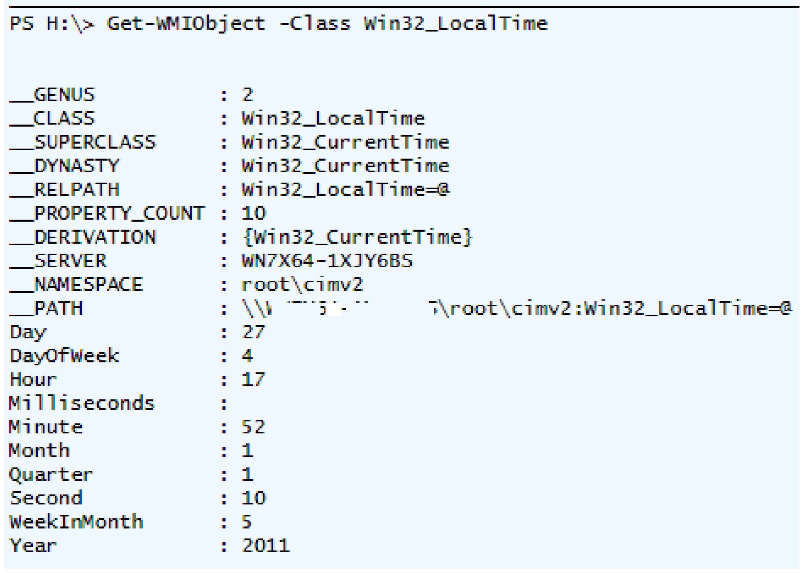
Timer Events

WMI timer events are generated by WMI infrastructure at a point in time. In the good old days of Windows 2000, we had to capture these by creating an instance of classes derived from the *__TimerInstruction* class. Also, there are two types of timer events.

1. Absolute Timer Events occur on a specific day at a specific time.
2. Interval Timer Events are events that occur periodically at specified intervals. For example, every 30 minutes.

Prior to Windows 2003, we would use *__AbsoluteTimerInstruction* class and *__IntervalTimerInstruction* class to define these events. However, that isn't necessary anymore with the introduction of *Win32_LocalTime* and *Win32_UTCTime* WMI classes. With these two classes, we can use the standard WMI eventing model to consume timer events.

Let us take a look at *Win32_LocalTime*[11]. This class represents an instance of *Win32_CurrentTime*.



```
PS H:\> Get-WMIObject -Class Win32_LocalTime


__GENUS          : 2
__CLASS          : Win32_LocalTime
__SUPERCLASS     : Win32_CurrentTime
__DYNASTY        : Win32_CurrentTime
__RELPATH        : Win32_LocalTime=@
__PROPERTY_COUNT : 10
__DERIVATION     : {Win32_CurrentTime}
__SERVER         : WN7X64-1XJY6BS
__NAMESPACE      : root\cimv2
__PATH           : \\[redacted]\root\cimv2:Win32_LocalTime=@
Day              : 27
DayOfWeek        : 4
Hour             : 17
Milliseconds     :
Minute           : 52
Month            : 1
Quarter          : 1
Second           : 10
WeekInMonth      : 5
Year             : 2011
```

As shown in the above screenshot, an instance of this class represents the current date and time. There are a few special properties that tell us which quarter of the year, week in the current month, etc. Do refer to the MSDN link for this class and read about these properties. Especially, DayOfWeek property. It always indicates Sunday as zero irrespective of the system culture and language settings.

[11] Win32_LocalTime on MSDN: http://msdn.microsoft.com/en-us/library/aa394171(v=VS.85).aspx

Since, this class supports standard WMI eventing model, we can use an intrinsic event query to consume these events. For example,

```
#Setup WQL query
$query = "SELECT * FROM __InstanceModificationEvent WHERE TargetInstance ISA
'Win32_LocalTime'"

#Register WMI Event
Register-WMIEvent -Query $Query -SourceIdentifier "Event-Every-Second"
-Action {Write-Host "Event Received"}
```

The above code will result in displaying "Event Received" message in the console every second. This is because the event fires every second and that is the smallest time segment supported. Now, this won't be very useful unless we want to flood the consumer with a bunch of useless messages every second. Let us look at another example:

```
#Setup WQL query
$query = "SELECT * FROM __InstanceModificationEvent WHERE TargetInstance ISA
            'Win32_LocalTime' AND TargetInstance.Hour=14
                              AND TargetInstance.Minute=30
                              AND TargetInstance.Second=0"

#Register WMI Event
Register-WMIEvent -Query $Query -SourceIdentifier "Event-Every-Day-14.30PM"
-Action {Write-Host "Event Received"}
```

In this example, we use WQL query to register for a WMI event that fires every day at 14.30 hours. Note that the hour's value is in 24 hour format. Now, this is useful. We can use the -*Action* script block to do something better like running a backup script when this event fires. Similarly, we can also create event registrations for events occurring on specific day of every quarter (Quarter) or specific week (WeekInMonth) of every month.

However, here is the catch: there is a bug that causes DayOfWeek to stop triggering any timer events.

```
#Setup WQL query for DayOfWeek
$query = "SELECT * FROM __InstanceModificationEvent WHERE TargetInstance ISA
            'Win32_LocalTime' AND TargetInstance.DayOfWeek=3
                              AND TargetInstance.Hour=12
                              AND TargetInstance.Minute=16
                              AND TargetInstance.Second=0"

#Register WMI Event
Register-WMIEvent -Query $Query -SourceIdentifier "Event-Every-Tuesday-12-
16PM" -Action {Write-Host "Start backup"}
```

The above example will never trigger a timer event because of the bug I mentioned. This is a known bug at this time which may be fixed in next release of the operating system. No promises!

And, here is a workaround:
We just need to dupe WMI with a second DayOfWeek condition in the WHERE clause and this is how we do it:

```
#Here is the workaround. We just dupe WMI by putting a dummy DayOfWeek
#Condition
#WMI seems to be firing the timer event properly when there are multiple
#DayOfWeek Conditions
$query = "SELECT * FROM __InstanceModificationEvent WHERE
          (TargetInstance ISA 'Win32_LocalTime') AND
          (TargetInstance.Quarter=4) AND
          (TargetInstance.WeekInMonth=1) AND
         (TargetInstance.DayOfWeek=5 OR TargetInstance.DayOfWeek=9) AND
          (TargetInstance.Hour=19) AND
          (TargetInstance.Minute=41) AND
          (TargetInstance.Second=0)"

Register-WMIEvent -Query $query -Action {Write-Host "Event Arrived"}
```

In the above timer event query, we used DayOfWeek multiple times in the WHERE clause. And, if you observe closely, we used '9' as one of the values for DayOfWeek property. This has no impact on the outcome of the query since there is no 9th day of week. We just use this dummy value to make the query work.

So, what are these timer events used for? I have not come across a real need for timer events in system administration. However, one use is to be able to create complex scheduling tasks that are not possible using the regular task scheduler. For example, check the above code itself. Using that, we are subscribing to the timer event that occurs ***on every Thursday of every fourth week of a month in the third quarter of every year.*** This type of scheduling isn't possible with the regular scheduler.

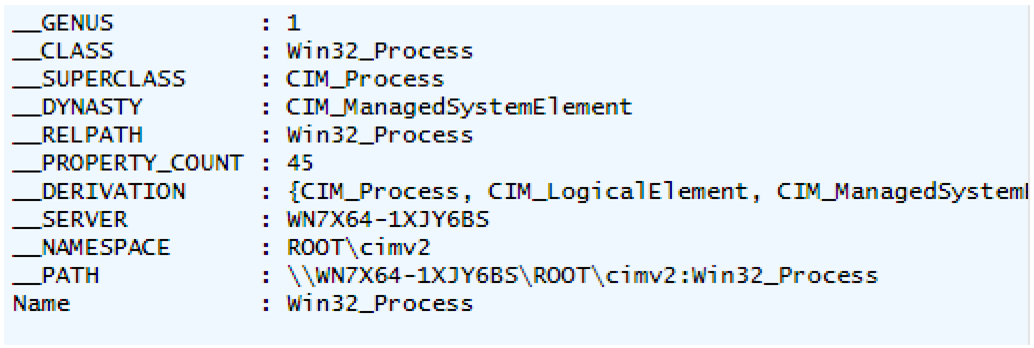
This brings us to end of discussion on the different types of WMI event queries. We have seen, so far, how to use WMI query language for creating WMI event queries and subscribe to WMI events.

WMI Schema Queries

Schema queries are used to retrieve class definitions (rather than class instances) and schema associations. In simple words, if you need to find out what type of information (this is what schema really means) a specific class holds, you use schema queries. Here is an example of a schema query:

```
$query = "SELECT * FROM meta_class WHERE __this ISA 'Win32_Process'"

Get-WMIObject -Query $query | Format-List
```

And, this is what you would see when you execute the code:



```
__GENUS          : 1
__CLASS          : Win32_Process
__SUPERCLASS     : CIM_Process
__DYNASTY        : CIM_ManagedSystemElement
__RELPATH        : Win32_Process
__PROPERTY_COUNT : 45
__DERIVATION     : {CIM_Process, CIM_LogicalElement, CIM_ManagedSysteml
__SERVER         : WN7X64-1XJY6BS
__NAMESPACE      : ROOT\cimv2
__PATH           : \\WN7X64-1XJY6BS\ROOT\cimv2:Win32_Process
Name             : Win32_Process
```

In one of the earlier discussions, we looked at retrieving class definitions with ASSOCIATORS OF and REFERENCES OF keywords. So, how are the schema queries different from the data queries using these two keywords?

Well, the above keywords return class definitions only when there are instances of those classes present. Using a schema query, we can retrieve the class definitions even when there is no instance present.

To understand what I just said, take a look at this example that shows how a WMI query was built when using REFERENCES OF.

```
$query = "REFERENCES OF {Win32_Process=$pid} WHERE ClassDefsOnly"
Get-WMIObject -Query $query | Format-List
```

See the {Win32_Process=$pid} part of the query. We have to specify some identifier so that we can get an instance. In the above example, we used $PID automatic variable. $PID is the process ID of the PowerShell host. If we don't specify a PID or some other identifier to get the instance, we end up with an error while executing the query. Now, go back and take a look at how we wrote the schema query. We did not specify any kind of an identifier or property anywhere and we were still able to get the class definition. That is the difference.

Let us dig a bit in to the schema query syntax.

We are familiar with the SELECT keyword. When building schema queries, only "*" is supported. Unlike other queries, you cannot do something like SELECT xyz FROM abc. It has to be SELECT * always. And, the use of meta_class specifies that we are building a schema query. The only way to narrow down the results, when using schema queries, is to use a WHERE clause. Let us now look at a few ways to narrow the query results.

Using __this

__this is a special property that identifies the target class for the query and using an ISA operator is must. This requests the definitions for the subclasses of the target class. Here is how you use this method:

```
$query = "SELECT * FROM meta_class WHERE __this ISA 'Win32_LogicalDisk'"
Get-WMIObject -Query $query | Format-List
```

This when executed, returns the class definitions of Win32_LogicalDisk and all its subclasses.

Using __Class

Using *__Class*, we can request for the class definitions of a single class and this is how we do it:

```
$query = "SELECT * FROM meta_class WHERE __class='Win32_LogicalDisk'"
Get-WMIObject -Query $query | Format-List
```

This is a brief overview of schema queries. We seldom need to use schema queries and IT Pros will probably never have to use this type of queries. You can also use REFERENCES OF and ASSOCIATORS OF keywords to get schema associations. You can use the SchemaOnly qualifier in the WHERE clause to retrieve the schema associations of a class.

WMI Event Consumers

We have used the PowerShell cmdlet, *Register-WMIEvent*, to create these event subscribers. As mentioned at the beginning of this discussion on event queries, *Register-WMIEvent* creates only temporary event consumers. What are these event consumers? What are the different types of event consumers?

In this chapter, we shall look at types of event consumers and how to create these event consumers. This chapter has no direct relation to learning WMI query language. This is just a bonus chapter that helps you understand WMI event consumer concepts.

Event consumers are applications that request notification of an event and then perform an action in response to that specific event. The PowerShell cmdlet we used in our examples so far creates an event consumer too. WMI just passes the relevant notifications from providers to consumers. Event consumers use an event filter to register with WMI to receive the event notifications and these consumers have no idea if the event providers support intrinsic events or extrinsic events. WMI abstracts this information from consumers.

There are two types of event consumers in WMI.

1. Temporary event consumers
2. Permanent event consumers

Temporary Event consumers

A temporary event consumer can receive event notifications only while it is active. The temporary event registrations are stored in memory and hence, will be lost when the hosting application closes. For example, the *Register-WMIEvent* cmdlet creates a temporary event consumer. However, once we close the PowerShell console, where we registered for the event notifications, the event registration will be lost. We will no more receive any WMI event notifications. We have seen several examples already using *Register-WMIEvent* cmdlet. Hence, this requires no further explanation.

Permanent Event consumers

A permanent consumer[12] is a COM object that can receive a WMI event at all times. A permanent event consumer uses a set of persistent objects and filters to capture a WMI event. Like a temporary event consumer, we set up a series of WMI objects and filters that capture a WMI event. When an event occurs that matches a filter, WMI loads the permanent event consumer and notifies it about the event. Because a permanent consumer is implemented in the WMI repository and is an executable file that is registered in WMI, the permanent event consumer operates and receives events after it is created and even after a reboot of the operating system as long as WMI is running.

[12] Permanent Event Consumers: http://msdn.microsoft.com/en-us/library/aa393013(v=vs.85).aspx#event_consumers

A permanent event consumer can receive event notifications whether it is active or inactive. If an event takes place and a permanent consumer is not currently active, then WMI tries to load the event consumer before forwarding the event. The permanent event consumers are useful in scenarios where we don't need to have consumer active all the time in memory.

A permanent event consumer receives events until its registration is explicitly canceled and unlike temporary event consumers, the permanent consumers have to be unregistered explicitly.

WMI comes preinstalled with a five standard consumer[13] classes. They are:

Event Consumer type	Description
ActiveScriptEventConsumer	Executes a predefined script in an arbitrary scripting language when an event is delivered to it. This consumer is available on Windows 2000, XP and later versions.
LogFileEventConsumer	Writes customized strings to a text log file when events are delivered to it. This consumer is available on Windows XP and later versions.
NTEventLogEventConsumer	Logs a specific message to the Windows NT event log when an event is delivered to it. This consumer is available on Windows XP and later versions.
SMTPEventConsumer	Sends an e-mail message using SMTP each time an event is delivered to it. This consumer is available on Windows 2000, XP and later versions.
CommandLineEventConsumer	Launches an arbitrary process in the local system context when an event is delivered to it. This consumer is available on Windows XP and later versions

Creating a WMI permanent event consumer includes creating an event filter, creating an instance of one of the above consumer classes and binding the filter & consumer together. An instance of the above physical consumer is called a logical consumer. Here is what MSDN documentation[14] says about logical event consumers:

> The properties of a logical event consumer specify the actions to perform when notified of an event, but do not define the event queries with which they are associated. When signaled, WMI automatically loads the COM object that represents the permanent event

[13] Standard consumer classes: http://msdn.microsoft.com/en-us/library/aa393649(v=vs.85).aspx
[14] Logical Event consumers: http://msdn.microsoft.com/en-us/library/aa393014(v=vs.85).aspx

consumer into active memory. Typically, this occurs during startup or in response to a triggering event. After being activated, the permanent event consumer acts as a normal event consumer, but remains until specifically unloaded by the operating system.

Let us now see the steps in creating a permanent event consumer. For this example, we shall use LogFileEventConsumer type.

Note
All the below steps require administrative privileges and hence should be run at an elevated command prompt.

Creating an event filter

An event filter is a WMI class that describes which events WMI delivers to a physical consumer. An event filter also describes the conditions under which WMI delivers the events. Creating an event filter involves creation of an instance of *__EventFilter* WMI class. For example,

```
#Creating a new event filter
$instanceFilter =
([WMICLASS]"\\.\root\subscription:__EventFilter").CreateInstance()
$instanceFilter.QueryLanguage = "WQL"
$instanceFilter.Query = "SELECT * FROM Win32_ProcessStartTrace"
$instanceFilter.Name = "EventFilter1"
$result = $instanceFilter.Put()
$newFilter = $result.Path
```

The above example shows how to create an event filter. At the end of the example, we stored the path to the newly created event filter in $newFilter.

Let us now look at how we can create an instance of **LogFileEventConsumer**.

Creating a logical consumer

Like any other WMI class, we must create an instance of the physical consumer. An instance of a physical consumer class is also known as a logical consumer. The logical consumer represents the physical consumer to WMI. The main purpose of a logical consumer is to provide the physical consumer with the parameters for the activities the physical consumer performs. The below code shows how to create a logical consumer:

```
#Creating a new event consumer
$instanceConsumer =
([wmiclass]"\\.\root\subscription:LogFileEventConsumer").CreateInstance()
$instanceConsumer.Name = "EventConsumer1"
$instanceConsumer.FileName = "C:\Scripts\processLog.log"
$instanceConsumer.Text = "New process has been
created: %TargetInstance.ProcessName%"
$result = $instanceConsumer.Put()
$newConsumer = $result.Path
```

Notice that these physical consumers exist in the *root\subscription* namespace. Also, notice the $instanceConsumer.Text. The *%TargetInstance.ProcessName%* is the name of the process that gets created. So, the value we provide to *$instanceConsumer.text* really depends on what we want to write to the log file. We specified the name of the log file as processLog.Log under C:\Scripts.

So, we have the path to the new LogFileEventConsumer in the $newConsumer variable. Let us now go ahead and bind them together.

Binding Event Filter and Consumer

Binding the event filter and consumer are straightforward. This is how we do it:

```
#Bind filter and consumer
$instanceBinding =
([wmiclass]"\\.\root\subscription:__FilterToConsumerBinding").CreateInstance()
$instanceBinding.Filter = $newFilter
$instanceBinding.Consumer = $newConsumer
$result = $instanceBinding.Put()
```

This is it. Now, we have all the steps complete. Now, whenever a new process gets created we should see an entry in the log file under C:\scripts location.

We now understand how to create a LogFileEventConsumer for permanent event notifications. As shown earlier, there are four more permanent event consumers. Also, as you see in the above examples, the creation of filters, consumers, and binding them together is quite verbose. This can be quite complex too depending what logical consumer you want to create.

To make this process easy, Trevor Sullivan[15] created a PowerShell module called PowerEvents[16]. This module can be used to any of the five permanent standard event consumers.

Introducing PowerEvents

PowerEvents is a Windows PowerShell v2.0 module designed to facilitate the ease of creating, updating, and deleting WMI (Windows Management Instrumentation) permanent event registrations. PowerEvents makes it easy to create WMI event filters (define the events you want to capture) and event consumers (responders to events), and then bind them together to initiate the flow of events. By leveraging permanent event registrations, you can perform advanced monitoring functions on a workstation or server that would otherwise require implementation of an enterprise monitoring product. Because WMI is incredibly vast in the information it provides, very detailed monitoring can be performed using almost any of the WMI objects that exist on a computer.

[15] Trevor's Blog: http://trevorsullivan.net
[16] PowerEvents: http://powerevents.codeplex.com

You can download the module and import the same using *Import-Module* cmdlet. This module has the following cmdlets to support creation of permanent event consumers.

1. Get-WmiEventConsumer
2. Get-WmiEventFilter
3. New-WmiEventConsumer
4. New-WmiEventFilter
5. New-WmiFilterToConsumerBinding

Let us now see how we can use PowerEvents module to create permanent event consumers. We shall create a scheduled task using WMI timer events and WMI command-line event consumer.

Creating an event filter

PowerEvents module provides a cmdlet to create an event filter — *New-WMIEventFilter.*

```
$query = "SELECT * FROM __InstanceModificationEvent WHERE
              (TargetInstance ISA 'Win32_LocalTime') AND
              (TargetInstance.Quarter=3) AND
              (TargetInstance.WeekInMonth=4) AND
              (TargetInstance.DayOfWeek=4 OR TargetInstance.DayOfWeek=9) AND
              (TargetInstance.Hour=12) AND
              (TargetInstance.Minute=0) AND
              (TargetInstance.Second=0)"

$taskFilter = New-WmiEventFilter -Name "WQL for 3rd quarter timer event" -
Query $query
```

This is it. You have the event filter created.

Creating an event consumer

The cmdlet for creating an event consumer is *New-WMIEventConsumer*. Let us see how to create a command-line event consumer. The idea is to invoke a backup script when the event fires.

```
$cmdConsumer = New-WmiEventConsumer -Verbose -Name "bkConsumer1" -
ConsumerType CommandLine -CommandLineTemplate "cmd.exe /c
`"C:\Scripts\backup.bat`""
```

The -*CommandLineTemplate* takes the path to the backup script. Also, note that the ConsumerType is set to CommandLine in this case.

Now that we have both filter and consumer, let us bind them together.

Binding Event filter and consumer

We need to bind the event filter and the consumer together so that the backup script gets invoked when the timer event gets triggered on the specified date & time. To do this, we will use *New-WMIFilterConsumerBinding* cmdlet.

```
New-WmiFilterToConsumerBinding -Filter $taskFilter -Consumer $cmdConsumer -
Verbose
```

This is it. The backup script gets triggered once the timer event gets fired. This is just one example of creating complex scheduling tasks using WMI timer events. And, using PowerEvents makes it easy to create permanent event consumers. Go and explore!