

## Getting Started with PowerShell by Guy Thomas

### Chapter Headings:

Windows PowerShell – Our Mission .....	3
Windows PowerShell Introduction .....	4
GS1: Download your copy of PowerShell .....	9
GS2: Three Ways to Execute a PowerShell Command.....	11
GS3: Three Key PowerShell Commands.....	14
GS4: Get-Member .....	18
GS5: PowerShell's Cmdlets .....	21
GS6: PowerShell Aliases.....	25
GS7: Controlling PowerShell's Results with Out-File .....	30
GS8: PowerShell's   Where {\$_.property -eq statement} .....	33
GS9: PowerShell Loops.....	36
GS10: PowerShell's WMI Techniques .....	41
GS11: Creating a PowerShell Function.....	47
GS12: Mastering Windows PowerShell's Profile.ps1 .....	50
Twelve Real Life Tasks for PowerShell .....	53
RL1: Checking the Eventlog with PowerShell.....	54
RL2: Scripting Files with PowerShell's - Get-Childitem (gci) .....	57
RL3: Finding text with Select-String .....	61
RL4: Deleting Temp Files.....	66
RL5: Listing the Operating System's Services with Get-Service .....	70
RL6: Starting an Operating System's Service with Start-Service.....	74
RL7: Checking your Disk with Win32_LogicalDisk.....	78
RL8: PowerShell: More flexible than Ipconfig.....	80
RL9: Scripting PowerShell's ComObject and MapNetworkDrive .....	83
RL10: Scripting - COM Shell Objects (Run Applications) .....	86
RL11: Active Directory and PowerShell.....	90
RL12: Creating an Exchange 2007 Mailcontact.....	96
Syntax Section .....	98
SN1: Type of Bracket.....	99
SN2: Conditional Operators .....	101
SN3: Format-Table (ft) .....	104

SN4: PowerShell's -f Format .....	108
SN5: Group-Object .....	112
SN6: PowerShell's If Statement.....	114
SN7: Windows PowerShell's Switch Command .....	116
SN8: PowerShell's Top Ten Parameters, or -Switches .....	119
SN9: Pipeline Symbol ( ) or ( ; ).....	122
SN10: PowerShell Quotes .....	124
SN11: PowerShell's Variables.....	126
SN12: PowerShell -whatif and -confirm.....	131

## Windows PowerShell – Our Mission

My purpose in writing this book is to two fold. Firstly, to encourage a new generation of script writers to learn PowerShell; secondly I want to provide experienced scripters with a bank of example scripts so that they can convert to PowerShell.

To newbies I want to say: 'It really will be easy to get started. You will be surprised by how many tasks on your Windows computer benefit from a PowerShell script'. To the experienced scripters I say, 'I want to build a reference library for that moment when you have forgotten the syntax for a particular PowerShell command'.

To help you discover the benefits of PowerShell, I will not only provide real life tasks, but also I will add Learning Points. One of my greatest joys is when you modify my code to fit your situation.

In September 2007 Microsoft released Powershell 1.0 RTW (Release to Web). In 2008 Microsoft will release version 2.0. To check which version you have, launch Powershell and type these five characters:

\$host

**Result: (See screenshot)**

Version : 1.0.0.0

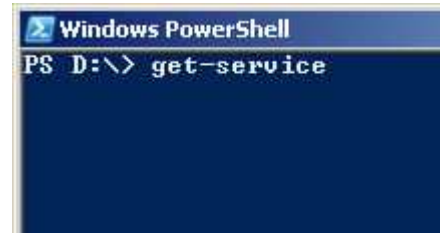


### I divided this PowerShell book into 3 sections:

- Getting Started (GS) - 12 basic techniques to assist you take those first faltering steps.
- Real life (RL) - 12 examples of PowerShell undertaking tasks on your computer.
- Syntax (SN) - A selection of 12 common punctuation and grammar constructions for your scripts.

## Windows PowerShell Introduction

My goal is to persuade Windows administrators that Microsoft's PowerShell is a friendly scripting language, which is well worth learning.



Using my examples and Learning Points, you will soon be able to interrogate the server's operating system. So get started, by downloading and installing Windows PowerShell, together with .Net Framework.

## Introduction to Windows PowerShell Topics

- Who is PowerShell designed for?
- Twelve PowerShell Techniques
- My Challenge - Start your PowerShell journey

### Who is Windows PowerShell designed for?

What impresses me about PowerShell is that even as an amateur, you can get started easily and you will soon be creating interesting scripts. The hidden message is that PowerShell works at many levels, moreover, you can quickly rise through these levels. As your journey proceeds, PowerShell teaches you about scripting theory, and scripting theory teaches you which constructions to look for in PowerShell.

The vision behind PowerShell is to provide cmdlets (scripts) which automate repetitive tasks. There is a feeling, at least in Exchange 2003/7, that people cannot find settings because they are hidden underneath so many levels of sub-menus. What I find is that it soon becomes as fast to issue a few commands in the Microsoft Shell, as to configure the same settings through a GUI.

Admins who have always wanted a command line shell for interactive scripting are going to enjoy PowerShell. The most amazing feature of PowerShell is that it lets you think in previous language, while you figure out its native ways of operating. Using the built-in Aliases, or creating new Aliases is the key to this transition.

### Twelve PowerShell Techniques

I have sequenced the following twelve PowerShell techniques. Thus if you are new to PowerShell I suggest that you start by downloading PowerShell from Microsoft's site, then go to item: 2. How to execute basic commands.

1. Download the correct version of PowerShell
2. How to execute basic commands
3. Master PowerShell's key commands
4. Get-Member command
5. Create PowerShell cmdlets  
Use PowerShell's built-in cmdlets
6. Aliases - Types
7. Out-file

8. Where Statement: `$_name`
9. Loops
10. WMI Techniques
11. Functions - Create
12. Enable your 'Profile'

### **Contents of my PowerShell technique pages**

Each of the twelve technique has its own 'how to' instructions and also 'Learning points'. As a result you will be able to modify my examples to suit your situation. The only pre-requisite is that you must download the correct version of PowerShell and .Net Framework for your operating system. Microsoft's site has separate versions for XP, Vista and Windows Server 2003.

### **Getting started with PowerShell**

No other scripting language is so easy for the newcomer, yet offers so much sophistication for the experienced script writer. One reason that you can start learning PowerShell NOW is you can still use all your old commands such as: `cd`, `dir`, and even `Ipconfig`. My vision is that little by little, you can progress from commands you already know, e.g. `dir`, to PowerShell's `get-ChildItem`.

Now I don't pretend that it's easy to become an expert at PowerShell, I merely want to emphasise that it's easy to begin that scripting journey. Just as in golf, most people can get the ball airborne with a sand wedge, but only a professional can hit a ball out of the rough, clear a water hazard, then hold the ball on a down-slope. PowerShell is like golf in that anyone can play, but few will have the effortless control of a professional. To stretch my analogy to the limit, just as we can play along with our favourite golfers from the comfort of our armchairs, so we can emulate PowerShell's experts by copying and pasting their code.

**To get you started, here are five examples to type at the PS prompt type:**

- `get-command`
- `get-command *event`
- `get-Eventlog system`
- `get-Eventlog system -newest 100`
- `get-Eventlog system -newest 100 | where {$_eventid -eq 20}`

### **My Challenge - Start your PowerShell journey**

My aim is to get you writing your own PowerShell cmdlets. Meanwhile, here is my challenge to you: in ten minutes you will be able to perform all the tasks that you currently achieve at the CMD prompt. Moreover, at the end of half an hour you will be issuing PowerShell commands that are just not possible with `cmd.exe`.

My assumption is that you have successfully installed PowerShell and .NET Framework 2.0 (or 3.0). Begin by clicking Start, Run, and type `powershell` in the dialog box; alternatively, create a short cut to powershell in the Quick Launch area of the Taskbar.

## Try out a few PowerShell commands

Normally, when people move to another scripting language their old trusted commands don't work in the new shell. Well, you will be pleasantly surprised that when you migrate from cmd.exe, old favorites such as `cd` and `dir` are available thanks to PowerShell's Aliases. In a short time you will adapt to the new ways of doing old tasks; for example, `set-location` is the equivalent of `cd`, and `get-childitem` achieves the same result as `dir`. In addition, PowerShell allows you to run all the native Windows executables such as `Ipconfig`, `NetSh` and `NetDiag`.

Thanks to this built-in library of Aliases, those coming from UNIX, perl, or .NET can also make an easy transition to PowerShell. I don't want you to underestimate the body of knowledge required to be productive in PowerShell, I just want to make the point that it's easy to take that first step and build on familiar commands from your existing scripting language.

**Principle 1:** At the heart of every PowerShell command is a verb-noun pair. You start with verb, add a hyphen and then finish with a noun; for example, `set-location`, which is the equivalent of `cd` (change directory) in DOS. If you find exceptions that only require the noun, this is because PowerShell's intelligence adds the default verb 'get'. For example if you type, 'childitem', PowerShell assumes you mean `get-childitem`. More good news, advanced scripters can create their own aliases.

## PowerShell Introduction - Simple Examples

I realize that these are Windows rather than Exchange 2007 examples but I want to get you up and running:

Preamble, Start, Run, PowerShell. In the shell type:

```
get-process  
or  
get-Eventlog system
```

**Principle 2:** Create lots of scripts or cmdlets. What I mean is save your commands in a text file with a .ps1 extension, for example `ProcessAll.ps1`. Moreover, save that file into PowerShell's working directory. Now you are ready to execute these cmdlets from within this Microsoft Shell, for example: `.\ProcessAll`. Your two key characters are the dot and the backslash.

The secret is get into the rhythm: dot, backslash then filename. There is no need to add the .ps1 extension, also avoid spaces.

```
.\cmdlet_filename
```

## The Knack

The knack is to store all your cmdlets in a folder which you can easily reach from the PowerShell prompt. At the risk of teaching my grandfather to suck eggs, I suggest that you adopt one of these strategies.

- a) Save all cmdlets to the default PowerShell directory.
- b) Once PowerShell launches, immediately change directory to the folder containing your cmdlets  
`cd C:\cmdlets.`
- c) There is an advanced technique to re-program the default PowerShell directory to a folder of your choice.

If all else fails just type the complete path:

`C:\Documents and Settings\GUYT\Documents\PSConfiguration\ProcessAll`  
(Or `D:\scripts\ProcessAll`)



### Auto-completion:

To save typing, and reduce the risk of making a spelling mistake, invoke auto-completion by pressing tab.

Auto-completion examples

`get-ev [tab]` auto-completes to `--> get-Eventlog`

`set-lo [tab]` auto-completes to `--> set-location`

## Example of a PowerShell Cmdlet

```
# This cmdlet generates a report about memory in active processes
# Memory usage (>1000000000)"
Write-Host Report generated at (get-date)
Write-Host ""
get-process | where-Object { $_.VirtualMemorySize -gt 1000000000 }
```

## PowerShell's Built-in Help

PowerShell's help is clear, concise and full of useful examples. Put your faith in Powershell's help. Cast aside negative experiences of other help files. Overcome your pride and ask for help. All that you need to query the vast help information store, is either precede the item with 'help', for example, `help get-Member`; or else append `-?` for example, `get-process -?` (hyphen question mark). The executable powershell.exe even has its own help, try this at the PS prompt, type: `powershell -?`

Another useful sister cmdlet is: `get-command`, which lists the built-in cmdlets. Better still, try the `*` wildcard:

`get-command *service.`

## **Summary of PowerShell Introduction**

PowerShell is Microsoft's new scripting language. It is designed to take the place of VBScript and eventually to replace the 'Dos box'. What makes it easy to get started is that all the old DOS commands still work at the PowerShell command line.

Trust me; it really is easy to get started with PowerShell verb-noun commands. Therefore make sure that you download a version for your operating system, and start learning from the PowerShell script examples that I will supply on the other pages.



## GS1: Download your copy of PowerShell

One of my themes with Windows PowerShell is that nothing is too simple. Therefore, let me start by checking that you have downloaded and installed the correct PowerShell files. If indeed, you have PowerShell installed and functioning correctly then go to the next section.

### Download PowerShell Itself

Key point, in autumn 2007, you need to download an operating specific version of PowerShell. In 2008, look out for version 2.0 of PowerShell. For now, Microsoft will provide you with slightly different version of PowerShell 1.0 RTW (Release to Web) for:

- Vista
- Windows Server 2003
- XP
- For Windows Server 2008, there is a simpler technique: install PowerShell by launching the 'Add a Feature' option in Server Manager.

### Download PowerShell from Microsoft's site

<http://www.microsoft.com/windowsserver2003/technologies/management/powershell/download.msp>

### Download .NET Framework

PowerShell is an object based scripting language; consequently, it needs .NET Framework for the definitions of the Windows objects. PowerShell works fine with either .NET Framework v 2.0, or v 3.0. Search Microsoft's site for:

#### Microsoft .NET Framework 2.0 Redistributable Package (x86)

#### Microsoft .NET Framework 3.0 Redistributable Package

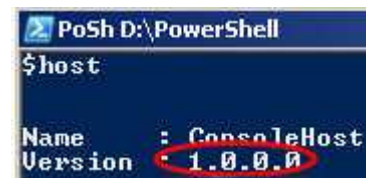
### PowerShell Installation

Here are the three stages before you get up and running with PowerShell:

- Obtain, then install .NET Framework 2.0
- Get a copy of PowerShell, for example v 1.0 (2.0 in 2008)
- Start, Run, PowerShell.

Once you have installed PowerShell (and .NET), try this simple command:

\$Host



Here opposite is a screen shot of what you should see: Version 1. x. x. x

As usual, there are two ways of doing everything, you could try.  
get-Host

## **The Situation with PowerShell and the Operating System**

Previous scripting programs came 'built-in' to the operating system. DOS has its cmd.exe, while VBScript has cscript.exe as its scripting engine. As of 2007, no commercial operating system, not even Vista, has PowerShell 'built-in'. Fortunately, installation is one of Microsoft's strengths, thus obtaining both PowerShell and .NET Framework is an uncomplicated one-off task.

PowerShell Runs on these systems: Windows Server 2003, Windows XP and even Windows Vista, however each has their own version of PowerShell.

## GS2: Three Ways to Execute a PowerShell Command


Here are basic instructions to help you execute a PowerShell command. You have a choice of three strategies; firstly, the time-honoured method of copying other people's examples, and then pasting their code into your PowerShell session. Secondly, creating cmdlets (my favorite), and thirdly, simply typing the instructions at the PowerShell command line.

### Topics - How to Execute a PowerShell Command

- Method 1 Copying and Pasting (Easiest)
- Method 2 Cmdlet (Best)
- Method 3 Type at the command line (Simplest)

#### Method 1 Copying and Pasting (Easiest)

This is how to copy and paste PowerShell instructions at the command line.

- Launch Windows PowerShell
- Copy all the lines of code into memory
- Right-click on the PowerShell symbol 
- Edit --> Paste
- Check the menus on my screenshot to the right
- Press 'Enter' to execute your pasted code



#### 'Vehicle' for Executing a PowerShell Command

Since the aim is to learn a technique, the practice code does not matter. Most people use 'Hello World', as their test 'vehicle'; however, I prefer to choose a real example. For instance, here is a cmdlet which gets the operating system processes, and then groups them by company name. Incidentally, the code is a work-in-progress where I am trying to output the data to a file called ProcessCompany.txt.

#### Example 1: To list Processes running on your machine

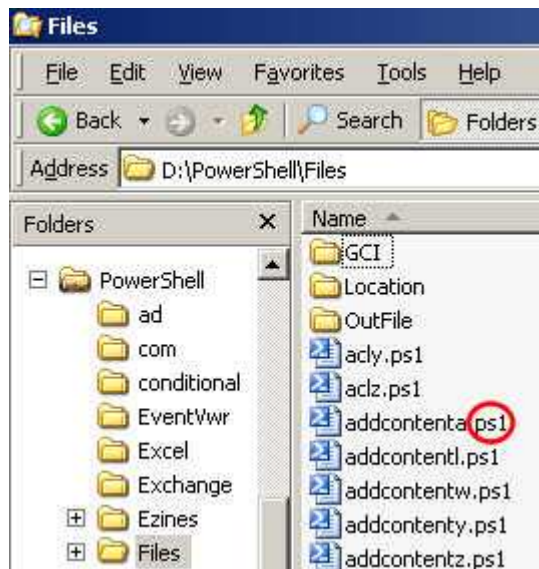
```
# PowerShell cmdlet to group Processes by company
$Path = "C:\PowerShell\ProcessCompany.txt"
$ProSvc = get-Process | sort company | ft -groupby company
$ProSvc
# $ProSvc | out-file $Path
```

**Result:** You should see a list of processes grouped by Company name.

## Method 2 Cmdlet (Best)

Because it saves your instructions permanently into a text file, this cmdlet method is an improvement over copy and paste. Creating cmdlets is my favorite technique because it is ideal for making amendments, then re-running the commands. As a bonus, cmdlets keep a record of what I have done.

- Copy the code in the 'Vehicle' above into a text file
- Save the file with a .ps1 extension, for example: addcontenta.ps1
- In Powershell, navigate to the folder where you saved addcontenta.ps1 (D:\PowerShell\Files in the screen shot below)
- Issue this command:  
.\addcontenta  
(dot backslash filename)



**Tip:** For each of my PowerShell projects, I launch Windows Explorer and then create a subfolder. Once I have a cmdlet that works, I store it in that subfolder. Thereafter my technique is to call for: File (menu), SaveAs, create a new file. Then I work with the new file and try to improve on that original version. At various points I call for SaveAs again, thus creating a family of cmdlets, for example: addcontenta, addcontentz, addcontenty etc.

My reason for employing this cmdlet technique is twofold, to cater for that moment when my code gets into a tangle, and I think: 'If only I could go back in time to when my script WAS working'. Well, thanks to saving lots of intermediary examples, I can revert to a working backup. Secondly, producing cmdlets means that I can return to my research months later and pick up from where I left off, instead of starting the project from scratch.

You may like to combine methods 1 and 2 by copying other people's code then pasting, not to the command line, but into a cmdlet text file.

### **Method 3 Type at the command line (Simplest method)**

Because PowerShell commands are so efficient, and thus short, I have no qualms about recommending that you simply type them at the command line. To test method 3, I have different examples or 'Vehicles'. You could start by typing this at the command line: `get-childitem c:\windows`.

Indeed, if you are new to PowerShell there is nothing like typing to give you a 'feel' for the syntax. As you type simple commands, so you get into the rhythm of the verb-noun pair. Another bonus of typing is that you understand when to use a dash (-) and when to precede a dash with a space. Here are three examples of what I mean:

`get-Eventlog -list` (Correct: Space before -list)

`get-help eventlog` (Correct: No space before eventlog)

`get-Eventlog system | get-Member` (Correct: No space between each verb-noun)

`get -eventlog -list` (Wrong: Space between get and -)

`get-help -eventlog` (Wrong: 'Overthink' eventlog is not a parameter or a switch, it is a positional argument and thus does not need a dash.

There are occasions when even experienced PowerShell scriptwriters resort to typing commands. As for me, I love creating cmdlets, but I do type commands especially when I want a list of an object's properties, for example,  
`abc-xyz | gm`. (gm is an alias for get-Member).

### **Summary of How to Execute a PowerShell Command**

Typical Microsoft, there are always at least two ways of executing PowerShell code. By all means start with the time-honoured method of copying and pasting, but for the long term, my advice is take the time to master the cmdlet method.

## GS3: Three Key PowerShell Commands

Here are three simple, yet key commands, which are designed to get you started with PowerShell. As you study, my example scripts keep in mind the golden rule, verb-noun, e.g, get-PSPProvider.

### Three Key PowerShell Commands or Cmdlets

1. Get-Command
2. Get-Help
3. Get-Member

#### 1. Get-Command

Let us begin by testing get-Command. What this instruction does is list all of PowerShell's noun-verb pairs; incidentally, these are also referred to as built-in cmdlets. Assuming that you have opened a PowerShell session, then you should see a prompt like: PS > Now type just one hyphenated phrase: get-Command

To filter the list, employ the famous star \* wildcard; here are three examples:

```
get-Command out*
```

```
get-Command add*
```

```
get-Command get-*
```

Let us experiment with a variation of this wildcard theme which displays only cmdlets beginning with the verb 'set':

```
get-Command set* -commandType cmdlet
```

It is possible to tweak the display of your output columns with ft (Format-Table). My hidden agenda here is to give you practice with PowerShell's Pipe symbol (|), try:

```
get-command |ft name, definition -auto
```

At the moment we are just 'playing', testing, or feeling our way, thus do feel free to experiment with your own variations of my suggestions. Once you have seen the long list of all possible commands, chose one example for further research, for example:

**Get-PSPProvider** (Or plain: PSPProvider)

This is what happened when I typed just: get-Psprovider <carriage return>

```
Name Capabilities Drives
-----
Alias ShouldProcess {Alias}
Environment ShouldProcess {Env}
FileSystem Filter, ShouldProcess {C, D, E, H... }
Function ShouldProcess {Function}
Registry ShouldProcess {HKLM, HKCU}
Variable ShouldProcess {Variable}
Certificate ShouldProcess {cert}
```

**Challenge:** try

`PSPProvider | get-Member`

### Another Command PSSnapin

What PSSnapin does is reveal the sources for the built-in cmdlets:

`Get-PSSnapin`

Or

`Get-PSSnapin |ft name, description -autosize`

Note how every PowerShell noun is singular, PSSnapin, Command and PSPProvider. Also note how a Pipe (|) followed by ft means format the output as a table, as opposed to format the results as a list (fl). Any words which follow 'ft' are names of the fields, each separated by a comma. At the end of the above command is the switch -autosize, this extra parameter tells PowerShell to close-up the width of the columns. When ever you use format-Table, or ft, try appending -autosize, or the shorter version: -auto.

In the example below, I have used ft to omit the Description field and just displayed the name:

`Get-PSSnapin |ft name:`

```
Name
----
Microsoft.PowerShell.Core
Microsoft.PowerShell.Host
Microsoft.PowerShell.Management
Microsoft.PowerShell.Security
Microsoft.PowerShell.Utility
```

## 2. Get-Help

Avoid arrogance, put aside pride, and call for PowerShell's built-in help. We all have to learn somewhere and only you know what you type in the privacy of your PowerShell command line.

Perhaps what puts us off trying built-in help is a bad experience with the stilted help of an old DOS system. Dare I suggest that experience with internet search techniques makes us more willing to try a modern application's own help?

PowerShell's help has some interesting switches, such as: -full and -example. Incidentally, -examples also works, a rare case of a plural PowerShell noun.

### Get-help get-wmiobject

Note: Get-help does not require the pipe symbol. In fact, the pipe (|) only gets in the way of get-help; Powershell does its best to interpret:

`Get-help | get-wmiobject cim_chip` But it still results in an error, therefore stick with plain simple:

`Get-help get-wmiobject`

### PowerShell's Hidden 'About' files

In the PowerShell folder, referenced by: \$PSHome \*\*, you will find a whole family of About help files. In these files, which all begin with About\_, you will discover information on topics such as Foreach, If and Elseif. My point is that you cannot get assistance by typing: get-help foreach, yet you can find a wealth of information if you read the file at: \$PSHome\about\_foreach.help.txt \*\*.

Here is a cmdlet that reveals the names of these About files:

```
# List all the About help files
$i=0
$Homes = get-ChildItem "$PSHome\about*. *" -recurse
foreach ($About in $Homes) {$About.name; $i++}
"There are $i about files"
```

**Result:** A list of 55 files. (The precise number vary with each version of PowerShell)

\*\* On my system \$PSHome translated to: C:\WINDOWS\system32\WindowsPowerShell\v1.0

### 3. Get-Member

Learn from mistakes my mistakes, and take a minute to imprint the format of this get-Member command into your brain. The key point is that the object you want information about comes at the beginning, and not at the end of the command. My second error was (is) forgetting the pipe symbol.

Here is the correct format:

Get-process | get-Member

Wrong

get-Member | get-process (sequence incorrect)

get-process get-Member (forgot the pipe symbol)

Worth a try:

Get-process | Get-Member -membertype **property**

Get-process | Get-Member -membertype **method**

Call me lazy, or call me showing that commands are not case sensitive. My point is that get-Process, Get-Process and get-process are all interpreted identically. In PowerShell, capitalization of command statements has no effect on successful execution.

### Tab Completion

Talking of being lazy, PowerShell also supports a tab auto-complete feature. Once you have typed enough of a command to make it unique, you can press tab and PowerShell completes the rest of the command.

get-process get-mem(tab) automatically expands to: get-process get-Member. As you can probably guess, this tab completion works for all commands not just this one. Incidentally, when you employ



PowerShell's parameters you can take advantage of a similar automatic completion, for example -auto instead of -autosize and -f instead of -filter. Once again, there are many more such auto-completions.

### **Summary of PowerShell's Commands**

Talk to an experienced PowerShell user, or talk to a teaching guru, they will each tell you that that the secret of success is simple, keep returning to the basics until you gain mastery. Those basics are:

get-command

get-help

get-Member

## GS4: Get-Member

While I have already introduced get-Member, this page goes into more detail. Get-Member is an essential command for discovering more about PowerShell's objects. Because you are at the command line, you cannot right click an object and check its properties; instead, what you can do is type: `get-ObjectXYZXYZ | get-Member`.

PowerShell is sometimes referred to as a self-describing language. Indeed, it is thanks to get-Member that we can see the properties and methods associated with any given object.

### Topics for PowerShell's get-Member

- The Concept behind get-Member
- The Basics of `get-ObjectXYZ | get-Member`
- Examples of get-Member
- Filter with `-membertype`
- Getting Help for get-Member
- Summary of PowerShell's get-Member

### The Concept behind get-Member

The reason that I use get-Member at every opportunity is so that I can discover 'handles', which help me achieve a particular scripting task. These 'handles' or memberTypes are split into methods and properties. Suitable objects to learn more about get-Member include, service, process, eventlog, or WmiObject.

Employing get-Member is a useful tactic in the bigger game of scripting specific properties. For example, if your goal is to stop, or to start a service, then you need to investigate the scripting properties, and possible values, for that service. A little research with get-Member will reveal a property called 'Status', with values of 'Running' or 'Stopped' - perfect for our task.

Here are three examples to illustrate my research technique:

1) `get-Service`

Results in a long list of Windows services

2) `get-Service | get-Member`


Results in a rich list of properties and methods

3) `get-Service messenger`

Reveals that the service Messenger has a property called 'Status' with a value of 'Stopped' (or 'Running').

3a) `get-Service messenger | get-Member`

Reveals more properties specific to the messenger service

 Try judicious use of 'tab' and invoke Auto-completion, for example try, get-p [Tab]. If you press tab again PowerShell cycles through commands beginning with get-p, for example get-Process, get-PSDrive

## The Basics of get-ObjectXYZ | get-Member

Naturally, the phrase get-Member never varies, a case of learn once and apply to many objects. Just remember the hyphen, and also remember that there are no spaces in verb-noun pairs. Please note that get-ObjectXYZ is just a generic name that I made up to illustrate get-Member in action.

Correct Syntax: get-Member

**Incorrect Syntax:** get member (no hyphen), or get -member or even get- member (spaces)

As I mentioned earlier, when you use get-Member in conjunction with other commands such as get-ObjectXYZ, remember the 'Pipe' or 'Pipeline' symbol. This vertical line | is ASCII 124 and looks like this at the PS Prompt |

**Correct Sequence:** get-ObjectXYZ | get-Member


**Incorrect Syntax:** get-Member | get-ObjectXYZ (wrong sequence of verb-noun pair)

**Trap:** get-Member get-ObjectXYZ (Forgot the pipe |)

## More Examples of get-Member

get-WmiObject Win32\_processor | get-Member

Note: get-Member is even more useful with WmiObject because this type of object varies more than objects such as Service, Process or Eventlog.

 Try aliases, for example gwmi for get-WmiObject. Many people use gm instead of get-Member.

get-Process | get-Member

get-Eventlog system | get-Member

Note if you don't tell PowerShell which eventlog you want, the command does not complete.

## Filter with -membertype

Results from the simple command: get-Member, may produce too many MemberTypes; if so, then you can filter the output with a -membertype argument, for example:

get-Process | get-Member -membertype property

or

get-Process | get-Member -membertype method

Appreciate that the results of the parameter -MemberType are grouped into at least four categories: AliasProperty, Method, Property and ScriptProperty. Once you have researched -MemberType, you may see new applications for the properties that it reveals, for example  
get-process | group company

Strictly speaking, the above command should be, 'group-Object company', however, since 'group-Object' has an alias, you can shorten the command to plain 'group'.

Here is a method for expanding the company information:

```
get-Process | sort company | format-Table -group company name, description -autosize
```

Incidentally, this whole page of -member examples gives valuable experience of when to use the hyphen - also called a dash, and sometimes referred to as a minus sign. For example -membertype takes the hyphen prefix, whereas **property**, as in -membertype **property** does not need any prefix.

## Getting Help for get-Member

For a complete list of filters supported by the get-Member command, call for help thus: help get-Member.

This may be just me, but I have the urge to call this -method instead of -member. I mention this because you always learn more from mistakes. The answer lies in, a) Reading the error message! b) Trying an example you know works, for example, I was trying this:

Here is a mistake that I made:

```
get-WmiObject win32_computersystem | get-method
```

When I read the error message it said: 'get-method is not recognised'. Hmmm. . . I thought, let me try an old friend the process object.

When I tried

```
get-process | get-method
```

 and this also failed, I realized that I was having a 'senior moment'.

Fortunately I woke up and read the error message slowly. 'get-method is not recognised'. Ah ha, get-method is what's wrong, why don't I try get-Member. Perfect, it worked just as I wanted. Thank you error message.

get-help and get-Member working in tandem.

Remember that get-help and get-Member are complimentary. On the one hand, get-help will disclose information about parameters, or switches that you can employ with your command: for example -recurse with get-childitem -path and -pattern with Select-String.

On the other hand, get-Member will disclose the properties and methods, for example .extension .fullname, both of which are useful with get-childitem. My point is that get-help and get-Member each provide different information; they are complimentary rather than interchangeable. If you are complete beginner, investigate both get-help and get-Member; when you are an intermediate don't get fixated on one, and forget all about the other.

## Summary of PowerShell's get-Member

When you need to investigate the methods and properties available to a PowerShell object, then call for get-Member. You will soon get used its hyphen and associated 'Pipe' symbol (|), just remember the correct sequence:

```
get-ObjectXYZ | get-Member.
```

## GS5: PowerShell's Cmdlets

With PowerShell you have a choice, you can either type commands directly at the shell's command line, or else you can store the same commands in a text file. Naturally you then call that script file, known as a cmdlet, from the PowerShell command line.

As with other script languages, notepad is an adequate vehicle for writing or editing scripts, the only PowerShell specific requirement is that you must save the cmdlet with a .ps1 file extension. (Note this differs from the Monad Beta where the extension was .msh). Writing scripts has two extra benefits, it documents your commands, so that you don't forget them, and also, these cmdlet files provide a proud record of your PowerShell achievements.

### PowerShell Cmdlet Topics

- Cmdlet (Command Let)
- PowerShell Execution Policy Adjustment
- Filename and .ps1 extension
- Calling the filename
- Copy and Paste Into PowerShell
- Summary of Cmdlets in PowerShell

### Cmdlets (Command Lets)

#### First Meaning of Cmdlet

Cmdlet has two meanings in Powershell; the first meaning of cmdlet is a synonym with a PowerShell script. A cmdlet is a series of commands, usually more than one line, stored in a text file with a .ps1 extension. It is this scriptlet meaning of cmdlet that I feature on this page.

#### Second Meaning of Cmdlet

In PowerShell circles there appears to be a second meaning for the word cmdlet. For example , Microsoft say: **cmdlet means a built-in PowerShell command**, which corresponds to a single verb-noun pair. Such a cmdlet is often accompanied by an alias, for example: get-Member with its alias of gm.

#### Advantages of Cmdlets

Once you have experimented with a few basic instructions at the PowerShell command line, you may wish to store your code in its own cmdlet file. The benefit is that you can replay these perfect lines of instructions later. This strategy becomes more and more useful as the code grow in complexity. My tactic is once a cmdlet achieves my basic objective; I call for 'SaveAs', and then use the copy for further development. Seven times out of ten the 'development' goes hopelessly wrong, at which point I am so grateful that I saved a working copy.

I hope that you are getting the idea of a cmdlet. Spend time perfecting the PowerShell commands, then save them in a text file. This technique saves you typing in lines of code and the command line; instead, you call the cmdlet by typing: dot slash and followed by its filename, for example .\memory. Incidentally, building cmdlets fits in with my strategy of assembling scripts in sections.

## Cmdlets - Three Quick Instructions

Creating these PowerShell cmdlets is straightforward, and most rewarding. Here are three essential tasks to ensure that the instructions in these script files execute correctly.

1. For security, the operating system won't run PowerShell scripts by default. Thus we need to adjust the ExecutionPolicy to allow PowerShell scripts to run. The better method is to issue this instruction at the PowerShell command line: `set-executionpolicy RemoteSigned`.  
Alternatively you could edit the registry.
2. Make sure that you save the filename with a `.ps1` extension.
3. Learn the knack of calling the file from the PowerShell command line, either by typing the full path:  
D: \scripts\filename, or if you were already in the D: \scripts folder, just type:  
`.\filename` (note the rhythm: dot slash filename).

**Note 1:** Let me explain, this is how I like to call cmdlets from a subdirectory. D: \scripts is my main script directory, however, I create cmdlets in subdirectories such as: D: scripts\wmi\32proc.ps1 . Assuming that I am at the PowerShell command line in the D: \scripts folder, all that I type at the prompt is: `.\wmi\32proc`.

**Note 2:** It took me ages to deduce that all I needed was plain `.\filename`. Avoid over-think; when you call the cmdlet file you don't need to add the extension. Adding `.ps1` is not necessary `.\filename` will suffice.

**Note 3:** While this dot slash (`.\`) method of executing the cmdlet script seems cumbersome, Microsoft decided on this method for security reasons. Hackers, phishers and the like could trick people into executing a rogue PowerShell script by clicking on it. However, nothing happens - unless you proceed the script with `.\` this safety feature offers a measure of protection from such mischief makers.

## Cmdlets Detailed Instructions

The following instructions are the same as those above, but with extra step-by-step directions.

### 1a) PowerShell's executionpolicy command

I prefer this method, which employs PowerShell's own commands to control the script Execution Policy. At the PS prompt type:

```
get-executionpolicy
```

Now try: `set-executionpolicy -?`

Here is the crucial command:

```
set-executionpolicy RemoteSigned
```

N. B. `get-executionpolicy` is available in PowerShell but not in the Monad beta.

## 1b) PowerShell Registry Adjustment

For security, and by default, Microsoft prevent PowerShell from executing cmdlet scripts; consequently we need to change a specific registry setting to allow our cmdlets to execute. If you don't make my amendment you may get the following error message when you call for a cmdlet script. 'The execution of scripts is disabled on this system'.

Our task is to open the registry and amend the value of the REG\_SZ ExecutionPolicy, specifically change Restricted to RemoteSigned. There are two additional values called Unrestricted and AllSigned. However, RemoteSigned is the best because it allows you to run scripts locally, while preventing people from hacking you from other machines e.g. the internet. To check the setting launch regedit and navigate to:

HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell

(In some versions of Powershell / Monad the path maybe slightly different.

HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.Management.Automation.ps1 )

Change this registry key:

REG\_SZ ExecutionPolicy RemoteSigned.

## 2a) Filename and .ps1 extension

Filename and .ps1 extension

When you create your cmdlet with notepad, the filename must have a .ps1 extension for example, RunningProcess.ps1 . One way of creating such a file is to save your PowerShell commands in notepad, then from the file menu: Save As, Type, All Files, RunningProcess.ps1. Just to be doubly sure, you could put the filename in double quotes, for example: "RunningProcess.ps1 ". I maybe paranoid, but please check the file is not called RunningProcess.txt or RunningProcess.ps1 .txt.

What you put in filename.ps1 are the same commands that you type in the PowerShell box. You could start by creating a file with a simple verb-noun pair, for example just:  
get-process.

This may seem too simple, but my philosophy is keep things straightforward and build on success.

Here is a more advanced set of instructions just to give you the idea of the power of the cmdlets.

```
# RunningServices.ps1
# This script generates a report about Running Services
# Guy Thomas September 2007
# Version 1. 5
"" # Insert a blank line
"Report generated at " + (get-date)
"" # Insert blank line

"Services that are running"
get-Service | where-Object { $_.status -eq "Running"}
```

## Learning Points

**Note 1:** The key command in this example is: get-service. Observe the singular noun service, furthermore, PowerShell specializes in consistency, thus nouns are always singular.

**Note 2:** Let us dissect the where clause. {\$\_, is a special variable to indicate, in the current pipeline. The dollar sign indicates we are using a variable and the underscore has the special meaning, a value in this stream. The process object has many properties, amongst them is .status. -eq means the left side equals the right side value. Observe the cmdlet and see that in this instance, we are looking for a value held by \$\_.status equal to "Running" and not equal "Stopped".

**Trap:** { \$\_.status = "Running"} Remember that PowerShell introduces comparisons with a hyphen and not an equal sign, thus we have -eq -match, -contains.

## 2b) Calling the filename

Assumptions: You saved the cmdlet script files into a directory called D:\ scripts. Also, in my example, I assume that the actual file is called RunningServices.ps1 .

When you execute the cmdlet by calling the filename from the PowerShell command line, you don't need to add the .ps1 extension. However, you need to pay attention to the path. Start at the command line by typing the full path, for example, D:\ scripts\RunningServices.

When that works, build on success. Navigate in PowerShell to the D:\ scripts folder, to achieve that, you could try this command

set-location d:\ scripts

Now issue the shorter command: .\RunningServices (Rather than D:\ scripts\RunningServices)

Note: Observe the dot and the slash in front of the filename, incidentally the slash can be either a backslash or a forward-slash, therefore, ./runningservices works.

## Copy and Paste Into PowerShell

If, for what ever reason, you do not wish to use a cmdlet, the alternative is to copy instructions from notepad and then paste them into the PowerShell interface.

## Summary of Cmdlets in PowerShell

When you execute PowerShell commands you have a choice, either just type the instructions at the Microsoft Shell command line, or create cmdlets scripts which you then call from the PS > prompt. Remember before you start experimenting with PowerShell scripts, change the executionpolicy to RemoteSigned. To enable scripts to run locally, type this command:  
set-executionpolicy RemoteSigned.

As a result the PowerShell registry will now permit .ps1 files to execute at the PowerShell command line.



## GS6: PowerShell Aliases

Powershell's built-in Aliases will help you make the transition from another command shells. For example, in DOS the 'cd' command means change directory, in PowerShell there is an Alias which maps the native set-location to DOS's cd. The result is that in PowerShell you can type cd and it works just as it did in DOS.

There is a second family of PowerShell Aliases, these are designed simply to reduce your keystrokes, for example, instead of typing thirteen keystrokes for get-childitem, just type three letters: gci.

### PowerShell Alias Topics

- Check Aliases
- List of Built-in Aliases
- Create your own PowerShell Alias
- How to Permanently Save your Alias
- Summary of Alias in PowerShell

### Check Aliases with get-alias

Let us start with the master command which lists PowerShell's built-in aliases. Launch PowerShell and type:  
get-alias.

You can also filter get-alias, for example, [a-g]\* lists all the aliases beginning with letters 'a' through 'g'. Incidentally, this simple example demonstrates how PowerShell employs wildcards, and also alerts you to the significance of different types of brackets.

As ever, you can refer to help, for example type: help get-alias. There is also: help new-alias, or even help delete-alias. I have deliberately not emphasised creating your own aliases. My reasoning is this, any aliases you create will not work if you send scripts containing such aliases to other people. This could cause confusion and thus is best avoided especially if you are just starting to learn PowerShell.

### List of PowerShell's Built-in Aliases

As you check this list see if you can detect two types of PowerShell Aliases, those convenience Aliases that simply save key strokes, for example, gci (get-childitem) and those Aliases that help people transition from other languages, for example, cd (set-location).

Alias	Definition	Alias	Definition
ac	add-content	kill	stop-process
cat	get-content	lp	out-printer
cd	set-location	ls	get-childitem

chdir	set-location	mi	move-item
clc	clear-content	mount	new-drive
clear	clear-host	move	move-item
cli	clear-item	mp	move-property
clp	clear-property	mv	move-item
cls	clear-host	nal	new-alias
clv	clear-variable	ndr	new-drive
copy	copy-item	ni	new-item
cp	copy-item	nv	new-variable
cpi	copy-item	oh	out-host
cpp	copy-property	popd	pop-location
cvpa	convert-path	ps	get-process
del	remove-Item	pushd	push-location
dir	get-childitem	pwd	get-location
echo	write-Object	r	invoke-history
epal	export-alias	rd	remove-Item
epcsv	export-csv	rdr	remove-drive
erase	remove-Item	ren	rename-item
fc	format-custom	ri	remove-Item
fl	format-list	rm	remove-Item
foreach	foreach-Object	rmdir	remove-Item
ft	format-Table	rni	rename-item
fw	format-wide	rnp	rename-property
gal	get-alias	rp	remove-property
gc	get-content	rv	remove-variable
gci	get-childitem	rvpa	resolve-path

gcm	get-command	sal	set-alias
gdr	Get-psdrive	sasv	start-service
ghy	get-history	sc	set-content
gi	get-item	select	select-Object
gl	get-location	set	set-variable
gm	get-Member	si	set-item
gp	get-property	sl	set-location
gps	get-process	sleep	start-sleep
group	group-Object	sort	sort-Object
gsv	get-service	sp	set-property
gu	get-unique	spps	stop-process
gv	get-variable	spsv	stop-service
h	get-history	sv	set-variable
history	get-history	type	get-content
icm	invoke-command	where	where-Object
ihy	invoke-history		
ii	invoke-item		
ilal	initialize-alias		
ipal	import-alias		
ipcsv	import-csv		

## Create your own PowerShell Alias

The command to add your own alias is: `set-alias`. You follow this command with your choice of aliasname and then complete the instruction by specifying an existing PowerShell verb-noun, for example:

```
new-alias xcopy copy-item  
or  
set-alias xcopy copy-item works just as well.
```

The new alias is called `xcopy` and what it does is the equivalent of `copy-item`. To double check type:

```
get-alias xcopy
```

N. B. Don't go mad with aliases, stick to one verb and one noun. I tried this:  
`set-alias eventvwr get-Eventlog application`, however it did not work as I had hoped - too many arguments.  
`set-alias eventvwr` is ok, but rather disappointing because it prompts you for the name of the log.

### Function - An alternative to Alias

If you really need a more complex set of commands then consider a function, which you can then save to the Function 'drive', investigate with: `get-psdrive` or `get-psdrive` function.

### Sequencing - Aliases come first

Suppose you have an Alias called 'eventvwr' and also a cmdlet with the same name 'eventvwr.ps1', but with different instructions. What would happen is the Alias would be processed first. In fact PowerShell checks the Aliases before it looks for functions or cmdlets.

## How to permanently save your Alias

If you just create an alias at the command prompt it is desperately disappointing that PowerShell does not remember your aliases the next time you logon. Good news, you can add your aliases to `Profile.ps1`, the benefit is they will now be available for each and every session.

My friend 'Barking' Eddie even has a file just for his aliases, he calls it `profile_alias.ps1`. His trick is to call this dedicated file from within `Profile.ps1` with this line:

```
. "C:\Documents and Settings\EDDIE\My Documents\PSConfiguration\profile_alias.ps1 "
```

If you try this at home, then substitute your username for EDDIE, or else it will not work.

N. B. That's not dirt on the screen or a Guy error, there really is a full stop or 'period' in front of `."C: \ "`.

Some say Eddie is Barking mad, but actually he is from Barking in Essex, either way, he takes this linking idea further and creates a `profile_function.ps1` file for his special functions.

### Aliases in cmdlets?

Can you call an Alias from a cmdlet? The answer is yes, why ever not, just remember that cmdlets mimic keystrokes you type at in the Microsoft Shell.

### **Problems caused by custom PowerShell Aliases**

My old friend 'Barking' Eddie has produced some great PowerShell scripts and at present they work fine. However, Eddie has created a time bomb, the problem is that his cmdlets have so many bizarre aliases that no-one else can understand them. Eddie has no scruples, he is hoping that the people who employ him today, will pay him big bucks to come out of his retirement and fix those cmdlets that no-one else can decipher.

The lesson is this, if you only have yourself to please, then create all the aliases that you want. However, if others need to understand or troubleshoot your scripts, then filling them with aliases will only store up problems that will return to haunt you.

### **Summary of PowerShell Aliases**

As far as I can see, using Aliases in PowerShell has these three benefits:

- To reduce typing.
- To smooth the transition from cmd.exe or other scripting languages to PowerShell.
- To create your own shortcuts for commands that you use often.

## GS7: Controlling PowerShell's Results with Out-File

Viewing a script's output on screen is all well and good, but often it's more convenient to save PowerShell's results to a file. Enter PowerShell's command: out-File. Here is an instruction that you simply bolt-on to an existing script..... | out-File "Filename.txt".

### Out-File Topics

- Introduction to Out-File
- Research Out-File with Get-Help
- Example 1 - Starting with Out-File
- Example 2 - Putting Out-File to Work
- Example 3 - More parameters (-Append and -NoClobber)

### Introduction to Out-File

Appending out-File to an existing command is easy. The biggest danger is 'over-think', just remember that PowerShell takes care of opening and closing the file automatically. Consequently, there is no need to waste time looking for non-existent file-open, or file-save commands. If the file specified by out-File does not already exist, PowerShell even creates it for you.

This is how the command works. Assuming the first part of the script delivers the results, redirect the output into a file with a command such as: | out-File c:\logs\result1.txt.

### Example 1 - Starting with Out-File

This example is purely to concentrate on the out-File command. In fact, the sooner we move on to example 2, the sooner we can do some real work.

```
# Powershell script to list the files in C:\Windows\System32
Get-ChildItem "C:\Windows\System32" | out-File "D:\Files\Sys32.txt"
```

**Note 1:** While out-File creates the file, you have to make sure that the path exists because out-File cannot create folders. In this instance, the alternative is to adjust D:\files to C:\PS, or an existing folder on your machine.

### Research Out-File with Get-Help

Once I get a command to work - and I like it, I want to know more. Get-Help always reveals at least one parameter that I had taken for granted, forgotten, or previously overlooked.

Get-Help out-file -full

(help out-File -full) also works. Be aware that there is no need for a pipe (|) with help.

If you append the -full switch, then PowerShell's help reveals useful parameters, for example, -filepath (taken for granted) -append (forgotten) -NoClobber (previously overlooked).

## Example 2 - Putting Out-File to Work

This example has extra features, which make the script more useful. Thanks to the -recurse parameter, the script is designed to drill down into the subdirectories. The question mark '?' introduces a 'Where' statement, its purpose is to filter just the dll files. Also, the output contains only the two file properties that we are interested in: name and creationtime.

```
# Powershell script to list the dll files under C:\Windows\System32
$DllFiles = gci "C:\Windows\System32" -recurse |? {$_.extension -eq ".dll"} `
| Format-Table name, creationtime -auto | out-File -filepath "D:\files\dll.txt"
```

**Note 1:** Spot the tiny backtick symbol ` at the end of line 2. This plain backtick tells PowerShell that the command continues on the next line.

**Note 2:** The parameter -filepath is optional. If you omit -filepath, PowerShell's intelligence deduces from the sequence of commands that "D:\ps\files\dll.txt" is the place to save the file. If there are no spaces in the path, then you can even omit the speech marks.

**Note 3:** If you get an error message 'Cannot find part of the path', then amend D: \files to a real folder on your machine.

## Example 3 - More parameters (-Append and -NoClobber)

If you run the script for a second time, have you noticed how out-File over-writes information? If your intention was to add data at the end of the file, then one solution would be to replace out-File with add-Content. However, because of its superior formatting, I prefer to stick with out-File and add the -append parameter.

-Append

```
| Out-File -filepath "D:\Files\dll.txt" -append
```

-NoClobber

What can 'No Clobber' mean? Lost your clothes? Not so, NoClobber means don't over-write the file. It seems to me that you would incorporate the -noclobber in circumstances where your intention was to save lots of files with slightly different names, and you did not want to risk losing their contents by over-writing.

```
| Out-File -filepath "D:\Files\dll June.txt" -NoClobber
```

**Note 1:** If you insist on running the script again, the -noclobber parameter causes PowerShell to generate an error message and thus protects the original file from being replaced.

## **Content Family**

Out-File is an honorary member of the 'Content' family, here are the cousin commands. I don't often use these other commands because they don't format the data as I would like.

Add-Content (Appends)

Clear-Content

Get-Content

Set-Content (Replaces existing content)

## **Out-Printer**

Out-File has a sister command called out-Printer. If you try:

Help out-Printer

Then you can see that you don't need to specify the name of the printer, PowerShell automatically selects the default printer, just as any application would.

You could experiment by substituting out-Printer for any of the above out-File commands

## **Summary of PowerShell's Out-File**

Trust me; you will make great use of PowerShell's out-File for saving results to a text file. This is a straight-forward command to bolt-on to existing commands. The main danger for newbies is looking for non-existent commands; remember that PowerShell takes care of both file open and file close.



## GS8: PowerShell's | Where {\$\_.property -eq statement}

I started my computing career in 1982 with a spreadsheet called SuperCalc. One of the most useful commands that I mastered was the 'If' statement, it was brilliant at filtering data. Now, although PowerShell also supports the 'If' statement, I find that PowerShell's 'Where' construction is much more versatile for filtering the output. Let us learn more about 'Where' by examining the following practical examples.

### PowerShell Topics for the Where statement

- Instructions to run the PowerShell code
- 'Where' examples which filter lists of files
- 'Where' examples which filter WMI objects

#### 'Where' examples which filter lists of files

A good time to add a 'Where' statement is when you need to filter a list. What we are going to do is get a list of files with 'get-childitem', and then pipe the output into a 'Where' clause, which filters according to this condition: file extension equals .exe.

Example 1a Where clause to find executable files

```
# Powershell script to list the exe files C:\program Files
get-childitem "C:\Program Files" -recurse | where {$_.extension -eq ".exe"}
```

**Note 1:** Perhaps this is a foible that only effects me, but whenever I create a 'Where' statement from scratch, I forget to introduce 'where' with a | pipe symbol.

**Note2:** # (hash) means: 'Here follows a comment'

#### Example 1b Where replaced with '?'

```
# Powershell script to list the exe files C:\Program Files
$PathDir = "C:\Program Files"
$FilesExe = gci $PathDir -recurse
$List = $FilesExe | ? {$_.extension -eq ".exe"}
$List | sort-Object -unique | format-Table name
```

#### Learning Points for PowerShell's Where

**Note 1:** Many PowerShell scriptwriters prefer to type the single question mark character, '?' instead of 'Where'. However, I prefer to stick with the full word 'Where' because it makes the script easier to read especially when the rest of the script is new or complex.

**Note 2:** If you like abbreviations, PowerShell has lots of aliases for common commands, for example, gci for get-ChildItem. In Example 1b we could also use 'ft' instead of format-Table. Also, I mostly use plain 'sort' rather than sort-Object.

**Note 3:** I cannot emphasise enough, always remember to introduce the where statement with a pipe, hence,  
..... | where {\$\_.name -eq "ReadMe"}

**Note 4:** Observe how sorting and formatting can improve the output. To see what I mean, compare Example 1a with Example 1b.

**Challenge 1:** Change ".exe" to ".dll"

**Challenge 2:** Change: where {\$\_.extension -eq ".exe"} to where {\$\_.name -eq "ReadMe"}

**Challenge 3:** Try changing the location from C:\program files to a folder of your choice. If you accept this challenge, also consider changing the file extension.

### 'Where' examples which filter WMI objects

Another situation that benefits from a 'Where' statement is when we research PowerShell's objects. To take a network example, imagine that our goal is to display the TCP/IP properties, but we have a problem - what is the WMI object called? Let us begin our quest by researching WMI objects. We already know that we can use, wmiobject -list, but let us refine our search by adding a where statement.

My idea behind providing three examples of achieving the same goal is to give you perspective, and to illustrate that there is no one 'right' way of coding PowerShell.

#### Example 2a

```
get-wmiobject -list | where {$_.name -match "Network"}
```

#### Example 2b

```
gwmi -list | ? {$_.name -match "Network"}
```

#### Example 2c

```
$objNetwork = get-wmiobject -list | where {$_.name -match "Network"}  
$objNetwork | Format-Table name
```

### Learning Points for PowerShell's Where

**Note 1:** Example 2b shows us that you can use a question mark (?) to replace 'Where'. This example also employs the alias gwmi for get-wmiobject.

**Note 2:** Most 'Where' statements employ the '{\$\_.xyz}' construction. What dollar underscore (\$\_.xyz) does is say: 'Take the xyz from the current input'.

The second half of the statement is concerned with an evaluation, and this is what achieves the filtering we desire. In these examples I use -match, but you could substitute, -eq (equals), -like, or any other of PowerShell's comparison operators.

**Note 3:** One lesson that I am for ever re-learning is that every PowerShell word, symbol, or even bracket is loaded with meaning. For instance, where {requires braces, and not elliptical brackets}.

### **Summary of PowerShell's where clause**

One of PowerShell's greatest assets is the ability to pipe the output of one command into another command. The 'Where' clause provides a suitable vehicle for testing this technique of piping and then filtering the output. The skill is to experiment until you get just the list that you need.

Incidentally, mastering the 'Where' command gives you an insight into the modular nature of PowerShell. Once you master of the rhythm of the command: Output | (pipe) where {\$\_.property - condition "comparison"}, then you can apply the same construction to numerous other PowerShell scripts.

## GS9: PowerShell Loops

Automating repetitive tasks is what scripting is all about. Therefore, each scripting language needs at least one method for cycling, or looping through a block of instructions. PowerShell provides a rich variety of looping techniques. However, because loops can go spectacularly wrong, I recommend you test a simple loop before graduating to more complex loops in your production script.

### Types of PowerShell Loops

- While Loops
- Do While Loop
- ForEach Loop (Three Examples)
- For Loop (Also known as the 'For statement')
- PowerShell Loop Output trick
- Summary of PowerShell Loops

### While Loops

The 'While' loop is the easiest loop to master, and also the most flexible. Each type of PowerShell loop has specific syntax rules; in the case of the 'While' loop there are only two elements (condition) and {Block Statement}. As so often with PowerShell, the type of brackets is highly significant; (Parenthesis for the condition) and {curly braces for the command block}.

The way that the loop works is that PowerShell evaluates the condition at the start of each cycle, and if it's true, then it executes the command block.

Here is a simple loop to calculate, and then display, the 7 times table.

```
$i = 7 # Set variable to zero  
while ($i -le 85) {$i; $i += 7}
```

### Learning Points

**Note 1:** Observe the two types of bracket (while) {Block Calculation}

**Note 2:** -le means less than or equals.

**Note 3:** +=7 increments the variable \$i by seven on each loop.

Alternatively, we could use a semi colon to join the two statements to form one line.

```
$i = 7; while ($i -le 85) { $i; $i += 7 }.
```

### Do While Loop

In the 'Do ... While' loop, PowerShell checks the (condition) is at the end of each loop. One feature of the 'Do While' loop is that the {Command Block} is always executed at least once; this is because the 'While' comes after the 'Do'.

```
$i = 7; do {$i; $i += 7} while ($i -le 85)
```

## Do Until

There is a variation of this style of loop, 'Do .. Until'. The layout of the components is the same as 'Do While', the only difference is that the logic is changed from, do while condition is true, to, do until condition is true.

```
$i = 7; do {$i; $i +=7} until ($i -gt 85)
```

## Foreach Loop in PowerShell (3 Examples)

The 'Foreach' loop is more complex, and has more arguments than the 'for' and 'Do While'. The key feature is that loop interrogates an array, known as a collection. In addition to the position, and the type of bracket, observe the tiny, but crucial keyword, 'in'.

Here is the syntax: `Foreach ($item in $array_collection) {command_block}`

### Example 1 - Math

```
foreach ($number in 1,2,3,4,5,6,7,8,9,10,11,12) { $number * 7 }
```

```
foreach ($number in 1.. 12) { $number * 7 }
```

```
$NumArray = (1,2,3,4,5,6,7,8,9,10,11,12)
```

```
foreach ($number in $numArray) { $number * 7 }
```

```
foreach ($number in 1,2,3,4,5,6,7,8,9,10,11,12) { $number * 7 }
```

```
$NumArray = (1.. 12)
```

```
foreach ($number in $numArray) { $number * 7 }
```

### Learning Points

**Note 1:** By creating five variations, my aim is to give you perspective and experience of the foreach loop.

**Note 2:** (1.. 12) is a convenient method of representing a sequence.

### Example 2 - To display files

Here is an example of a 'foreach' loop which displays the filename and its size. In order to get this example to work, create a cmdlet by saving the following into a file, and give it a .ps1 extension. Call the cmdlet from the PS prompt. If the file is called ListDoc.ps1 , and assuming that the file is in the current directory, then at the PS prompt type:

```
.\listdoc
```

Alternatively, from the PS prompt, type the full path D:\scripts\listdoc.

### Cmdlet 1

```
foreach ($file in get-childitem)
{
$file.name + " " + $file.length
}
```

In Cmdlet 2 (below), we can employ a simple 'if' statement to filter .txt files. Naturally, feel free to alter -eq ".txt" to a more suitable extension.

### Cmdlet 2

```
"File Name " + "`t Size" + "`t Last Accessed"
foreach ($file in get-childitem)
{if ($file.extension -eq ".txt")
{
$file.name + "`t " + $file.length + "`t " + $file.LastAccessTime
}
}
```

### Learning Points

**Note 0:** If all else fails, copy the above code, and paste to PowerShell prompt, and then press 'Enter'.

**Note 1:** `t means Tab.

### Example 3 - Active Directory

This example conforms to the ForEach (condition) {Code Block}; however, there is a preamble of 16 lines where the script connect to Active Directory. Moreover, the {Code Block} is spread over several lines.

N. B. Find \$Dom on line 7 and change the value to that of your domain, including the extension.

```
# ForEach_AD.ps1
# Illustrates using ForEach loop to interrogate AD
# IMPORTANT change $Dom 'value'
# Author: Guy Thomas
# Version 2. 5 November 2006 tested on PowerShell RC2

$Dom = 'LDAP://DC=YourDom;DC=YourExt'
$Root = New-Object DirectoryServices. DirectoryEntry

# Create a selector and start searching from the Root of AD
$selector = New-Object DirectoryServices. DirectorySearcher
$selector.SearchRoot = $root

# Filter the users with -like "CN=Person*". Note the ForEach loop
$adobj= $selector.findall() `
| where {$_.properties.objectcategory -like "CN=Person*"}
ForEach ($person in $adobj)
{
    $prop=$person. properties
    Write-host "First name: $($prop.givenname) " `
    "Surname: $($prop.sn) User: $($prop. cn)"
}
write-Host "`nThere are $($adobj.count) users in the $($root.name) domain"
```

### Learning Points

**Note 1:** ` on its own means word-wrap. `n means new line and `t means tab.

### For Loop (Also know as the for statement)

One use of the 'For loop' is to iterate an array of values and then work with a subset of these values. Should you wish to cycle through all the values in your array, consider using a foreach construction.

Here is the syntax: for (<init>; <condition>; <repeat>) {<command\_block>}

### Example

```
for ($i = 7; $i -le 84; $i+=7) {$i}
```

PowerShell Loop Output trick

I have not found it possible to pipe input into loops. Obtaining output was nearly as difficult; however, I have discovered this trick to assign the output to a variable, which we can then manipulate.

```
$NumArray = (1.. 12)
$(foreach ($number in $numArray ) { $number * 7}) | set-variable 7x
$7x
# Option research properties by removing # on the next line
# $7x | get-Member

Dejan Milic's Method (Better)

$NumArray = (1.. 12)
$7x = @()
foreach ($number in $numArray ) { $7x+=$number * 7}
$7x
```

/V/o/V/s Method (Fantastic)

```
$7x = 1.. 12 |% {$_ * 7 }
$7x
```

I (Guy) envy /V/o/V/s ability to write tight code. That % sign means 'foreach'. If you (readers) see anything on the internet by /V/o/V/, then you can be sure that it's top draw code.

### Summary of PowerShell Loops

PowerShell supports a variety of different loops, for example, 'While' and 'Foreach'. Brackets play a vital role in defining the elements syntax, (use parenthesis) for the condition and {use braces} for the command block. Take the time to master loops and thus automate repetitive tasks.



## GS10: PowerShell's WMI Techniques

One of the most useful jobs for PowerShell is creating a bank of WMI based scripts. Furthermore, scripting WMI with PowerShell is much easier and more efficient than using the combination of VBScript and WMI.

Skills that you develop through experimenting with PowerShell and WMI will help you manage your Exchange 2007 server. With practice, you will also be able to adapt these techniques to Active Directory and Longhorn. My point is that while PowerShell is clearly has a future, you can get started here and now by creating WMI cmdlets.

### WMI and PowerShell Topics

- WMI Perspective
- get-WmiObject
- WmiObject get-Member
- WmiObject - Help with Parameters
- WmiObject \$variables
- Summary of WMI and PowerShell

### WMI Perspective

To appreciate the beauty of a crystal, you should examine the facets from different angles. The same applies to the diamond that is WMI; I recommend that to gain perspective we investigate WMI from these five angles.

1. Imagine WMI as a database, which keeps information about a computer's components such as the: BIOS, services and network settings.
2. Regard WMI as a method for collecting data about a machine's hardware and software.
3. View WMI as a pipe, which magically connects to the core of any Microsoft operating system (post 2000).
4. Think of WMI as a having its own PowerShell dialect, for example the WQL select clause.
5. Treat WMI as a microscope, and use it to probe the operating system's objects and their properties.

Whenever I think about WMI for any length of time, it hits me: the operating system must know everything that's going on! Therefore, as long the PowerShell script has sufficient rights, it can use WMI and tap into that vast fountain of operating system knowledge. Windows Server 2003 must know 'stuff' such as how much memory each process is using, how much free space there is on each partition, which devices are on which Bus. It is even possible to manipulate or 'set' values on some of these properties and thus achieve configuration via scripts rather than GUIs.

### get-WmiObject

For me, my interest in PowerShell took off when I discovered this command:  
`get-WmiObject win32_computersystem`

The result was:

```
Domain : cp. mosel
Manufacturer : VIAK8T
Model : AWRDACPI
Name : BIG-SERVER
PrimaryOwnerName : Guy
TotalPhysicalMemory : 2146910208
```

I was curious to discover what other WMI Objects were available for scripting; then I remembered the -list switch from another PowerShell command (get-Eventlog -list). Thus I tried:

```
get-WmiObject -list
```

Next, I redirected the output from the screen to a file by appending 'out-File':  
out-File WmiObject.txt. To make:

```
get-WmiObject -list | out-File WmiObject.txt
```

My next problem was the list was too long, therefore I added a 'Where' filter

```
get-WmiObject -list | Where-Object {$_.name -match "Win32"}
```

And even better:

```
get-WmiObject -list | Where-Object {$_.name -match "Win32"} `
| out-File D:\wmi\win.txt
```

### Learning Points

**Note 1:** The tiny backtick (`) tells PowerShell that the command continues on the next line.

**Note 2:** On other pages I use plain 'Where', or even '?' instead of the full 'Where-Object'.

**Note 3:** I expect you have guessed that PowerShell commands are case insensitive. At present I am not sure which is best, WmiObject, wmiObject or WmiObject - they all produce the same results. Another minor point, since the verb 'get' is the default, you can shorten the command to: WmiObject win32\_computersystem. Or if you like aliases: gwmi win32\_computersystem

### WmiObject get-Member

Wouldn't it be useful to get a list of all the properties on a PowerShell object? The answer is get-Member. Here is an example of applying the get-Member command to examine the Win32\_Logical Disk:

### Example 1

```
get-WmiObject win32_LogicalDisk | get-Member
```

Sample Output (Heavily truncated)

```
TypeName: System. Management. ManagementObject#root\cimv2\Win32_LogicalDisk

Name MemberType Definition
-----
add_Disposed Method System. Void add_Disposed(EventHandler value)
Clone Method System. Object Clone()
.....
Access Property System. UInt16 Access {get;}
Availability Property System. UInt16 Availability {get;}
BlockSize Property System. UInt64 BlockSize {get;}
Caption Property System. String Caption {get;}
Compressed Property System. Boolean Compressed {get;}
.....
```

### Example 2

Here is another example, this time PowerShell interrogates the WMI BIOS property list:

```
get-WmiObject win32_BIOS | get-Member
```

### WmiObject - Help with Parameters

When I am in 'let us get started' mode, I gloss over the optional PowerShell commands. However, when we need to troubleshoot, then the secret of success is knowledge of a command's parameters. To reveal the full list of parameter, let's call for help:

```
get-Help get-Wmiobject -full
```

### Five useful Wmiobject Parameters

- -class
- -namespace
- -computerName
- -filter
- -query

#### -class

When you employ get-Wmiobject the first thing you want to specify is the class of the object, for example: Win32\_LogicalDisk. If this class name follows directly after get-Wmiobject there is no need to explicitly use the -class parameter. PowerShell assumes from the first position that the word defines the type of class you wish to script.

```
get-Wmiobject -class Win32_WmiSetting
```

Abbreviated version of the same cmdlet

```
get-Wmiobject Win32_WmiSetting
```

### **-namespace**

The reason that the -namespace parameter is optional because Wmiobject has a default value of: root\cimv2. Thus the time to include this parameter is when you wish to use a different namespace, such as: root\directory\ldap.

```
get-WmiObject -namespace "root\cimv2" -list
```

A different namespace: "root\directory\ldap"

```
get-WmiObject -namespace "root\directory\ldap" -list
```

The full syntax to specify the class and the namespace:

```
get-Wmiobject -class Win32_WmiSetting -namespace "root\cimv2"
```

**Note 1:** Interestingly, when you explicitly define the namespace, the command displays more properties, than if you omit the parameter and rely on the default.

### **-computerName**

As expected, when you use get-Wmiobject PowerShell defaults to the current machine; unlike VBScript, you don't have to add "." However, if you want to run the script against another machine you need to add the -computerName parameter. Incidentally, the target machine must have WMI, but does not need PowerShell. As all Microsoft Computers after Windows 2000 have WMI, there should be no problem; your only real concern could be firewalls. Knowledge of the -computerName parameter helps when you want a script which loops through multiple hostnames, or IP addresses.

```
get-Wmiobject -class Win32_WmiSetting `  
-namespace "root\cimv2" -computerName bigserver
```

**Note 1:** If you would like to run my cmdlet on your network, then please amend 'bigserver' to the name of a real machine on your subnet.

**Note 2:** To cope with word-wrap I added the tiny backtick ` command. This tells PowerShell that there is one command but it is split over two lines.

**Note 3:** As far as I can see, the shorter **-computer** works just as well as the longer **-computerName**. In fact, this is an example of PowerShell's intelligence, as soon as the word that you are typing becomes unique, then PowerShell automatically fills in the missing letters. To see what I mean try -comput, or even -comp.

### **-filter**

To illustrate the -filter parameter, let us set the scene: you wish to interrogate the logical disk, but you only want information about the 'c:' drive. Incidentally, -filter is much easier than comparable constructions in VBScript.

```
get-WmiObject Win32_LogicalDisk -filter "DeviceID = 'c:' "
```

**Note 1:** Pay close attention to the quotes. One set of double quotes surrounding "Device ID = ". And one set of single quotes around the disk drive 'c:'.

**Note 2:** PowerShell's usual comparison operators employ -eq. However, here -filter requires the equals sign.

**Note 3:** To concentrate on the main feature -filter, I omitted the optional parameters that we covered previously.

### **-query**

In this example, imagine that we need information about the hard disk, but we don't want the results cluttered with data about floppy drives, CD or DVD drives. If you are familiar with any dialect of SQL you may recognise the select statement below:

```
get-WmiObject -query "select * from Win32_LogicalDisk where DriveType = '3' "
```

**Note 1:** As with -filter, you need to be careful with syntax of the -query parameter. Observe the quotes, and how I have separated the single quote from the double quote with a space: = '3' ". In fact, as 3 is a numeric value, we could remove these single quotes and simplify the expression to: where DriveType = 3.

## **WmiObject \$variables**

Let us kill two birds with one stone. Firstly, I want to introduce variables and secondly, I want to filter the properties for a WMI Object. This is the idea, let us set a variable called \$disk equal to get-WmiObject Win32LogicalDisk. Now we can call individual properties, for example \$disk.name or \$disk.Size.

There is one additional script structure we must master, the foreach construction. Since there is more than one drive we need the foreach loop below. Observe how this construction requires a placeholder (\$drive in \$disk).

### Simple Example

```
$disk= get-WmiObject Win32_LogicalDisk  
foreach ( $drive in $disk ) { "Drive = " + $drive.name }
```

### Extra Maths Example

```
$disk= get-WmiObject Win32_LogicalDisk  
"Drive Letter Size GB"  
foreach ( $drive in $disk ) { "Drive = " + $drive.name + `  
"Size = " + [int]($drive.Size/1073741824)}
```

### Learning Points

Results (Your values for Size will be different!)

```
Drive Letter Size GB  
Drive = A: Size = 0  
Drive = B: Size = 0  
Drive = C: Size = 49  
Drive = D: Size = 29  
Drive = E: Size = 85
```

To recap, we begin with the variable \$disk. We set its value = get-WmiObject Win32\_LogicalDisk. We want to display the name \$drive.name and the size \$drive.Size. Because the raw disk size is in bytes, we best convert to gigaabytes (GB), thus we need to divide by 1024 x 1024 x 1024 (1073741824). [int] displays the number as an integer.

With the brackets it is always worth paying attention to detail. PowerShell is particular about the style of bracket, for example {Int} or even (int) draw an 'unexpected token' error message. Equally, the foreach construction needs the simple elliptical brackets (), while the properties are encased in curly {} brackets.

### Summary of PowerShell Get-WmiObject

Get-WmiObject is a good bellwether for PowerShell. Contrast the ease with which PowerShell displays WMI objects, with the struggle which VBScript achieves the same result. I say ease of PowerShell, there is still plenty to learn about brackets, variables and foreach loops. My advice is begin learning PowerShell by experimenting with a simple object such as 'Process' or 'Eventlog' before tackling the WmiObject family of objects.

## GS11: Creating a PowerShell Function

In some ways this is an advanced topic for a 'getting you started' manual. Thus you may wish to put functions on the back-burner until you have created a few cmdlets of your own.

As you may expect from a top-notch scripting language, PowerShell supports functions. There are several advantages of investing in the time needed to create functions. One advantage of a function is that once you get it working, it's easy to call the commands later in the same script, moreover, once perfected, the code works consistently. Another advantage of functions is that they help organize a long script into manageable chunks.

### PowerShell Function Topics

- Our Practical Task - Enumerate svchost
- A function called plist
- Here is the code for the plist function
- Summary of PowerShell Functions

#### Our Practical Task - Enumerate svchost

The task that I have set for our function is to enumerate which services are in each of the generic svchosts processes that you see in Task Manager. Incidentally, the reason for multiple svchosts is that certain processes 'fight' and thus must be kept separate. The solution is for the operating system to create multiple svchosts; for example, RemoteRegistry cannot co-exist with TermService.

#### Preliminary Commands

To get the idea of what we want to achieve, try these two commands individually:

```
get-process * | sort ProcessName
```

And then

```
get-wmiobject win32_service | sort ProcessId | group-Object ProcessId
```

Our mission is to create a function which combines both of the above commands and thus achieves a single list of all the svchosts with their corresponding services.

This is the output of our goal; we want our function called plist to produce this output:

```
Name    Id  service
----- --  -
svchost 740 {ERSvc}
svchost 916 {TapiSrv}
svchost 1120 {RemoteRegistry}
svchost 1392 {DcomLaunch}
svchost 1712 {RpcSs}
svchost 1772 {Dnscache, Dhcp}
svchost 1832 {LmHosts, W32Time}
```

```
svchost 1904 {TrkWks, WZCSVC, AeLookupSvc}
svchost 2884 {W3SVC}
svchost 3904 {TermService}
```

### An Example of a PowerShell Function called plist

When you declare a function it requires as a minimum: Functionname {Block of Work} The actual work is done by the PowerShell statements between the required {braces}. Functions may include optional parameters, these are enclosed in (parenthesis) and are introduced after the function's name and before it gets to work with the {}.

Here is the code for the plist function

```
function plist([string]$name="*")
{

$Svc = get-wmiobject win32_service | sort ProcessId | group-Object ProcessId
$ps = @(get-process $name | sort Id)
$i=0
$j=0
while($i -lt $ps.count -and $j -lt $svc.count)
{

    if($ps[$i].id -lt $Svc[$j].name)
    {

        $i++;
        continue;
    }
    if($ps[$i].id -gt $svc[$j].name)
    {
        $j++;
        continue;
    }
    if($ps[$i].id -eq $svc[$j].name)
    {
        $ps[$i] | add-member NoteProperty service $Svc[$j]. group;
        $i++;
        $j++;
    }
}
$ps;
}
```

# Here is the plist function in action against the svchost processes:  
 plist svchost\* | format-Table -autosize name,Id,service



### Learning Points from the Function Example

**Note 1:** Plist will be a string function and not an integer and is declared thus:

```
function plist([string]$name="*")
```

**Note 2:** \$Name="\*" returns all the names of the objects that get processed.

**Note 3:** Let us consider the instructions inside the {Braces}, starting with the variable \$Svc. What this does is get the wmi win32\_service. Here is the command:

```
$Svc = get-wmiobject win32_service | sort ProcessId | group-Object ProcessId
```

**Note 4:** The loop is covered by this While construction, the key is lt (less than):

```
while($i -lt $ps.count -and $j -lt $svc.count)
```

**Note 5:** The process name is controlled by:

```
$ps = @(get-process $name | sort Id)
```

**Note 6:** This is the clever line that appends all the services to each individual svchost

```
$ps[$i] | add-member NoteProperty service $Svc[$j]. group;
```

### Footnote - Simpler code

It is possible to replace the plist function (above) with more efficient code, nevertheless, remember that the purpose of this page is to introduce PowerShell functions.

Here is the alternative code if you just wish to check the instances of svchost

```
function plist([string]$a)
{
    $FormatEnumerationLimit = 100
    gwmi win32_service | ? { $_.PathName -match 'svchost' -and $_.ProcessId -ne 0 } | group ProcessId |
    ft
}
Plist
```

### Summary of PowerShell Functions

The purpose of this page is to understand how a function is constructed. Take it one line at a time. My goal was to break down a complex task into a series of single commands. The vehicle for our example task was drilling down into the SVCHOST processes that you see in the task manager.

## GS12: Mastering Windows PowerShell's Profile.ps1

Old timers like my friend 'Barking' Eddie can remember configuration files such as AutoExec.bat and Config.sys. More modern Microsoft operating systems need boot.ini; all these files control the startup environment. Well the purpose of this page is to configure the equivalent start-up file in PowerShell namely: Profile.ps1.

(Note 1: The extension is ps, followed by numeric one, making ps1 (not ps!).

(Note 2: The name of this file is 'profile' - singular)

### PowerShell Profile.ps1 Topics

- Mission to enable a basic Profile.ps1 file
- Enabling PowerShell Scripts
- Locating the path for the Profile.ps1 file
- Alternative Locations for Profile.ps1
- Summary of Profile.ps1

### Mission to enable a basic Profile.ps1 file

When I installed my Windows PowerShell v 1.0 it did not configure the Profile.ps1 file automatically. A little research revealed that the key folder is WindowsPowerShell. I want to spare you my grief in troubleshooting Profile.ps1 and instead, I want to concentrate on telling you how, and where, to control this configuration file.

### Enabling PowerShell Scripts

Profiles.ps1 is a script file; by default you cannot run any scripts, this is because at install the executionpolicy is set to Restricted. Without making the following configuration change you may get an error message: 'The execution of scripts is disabled'. Incidentally, this is the same command that allows all PowerShell's scripts to run. My point is that you may have run this command just before you configured your first cmdlet scripts.

The best solution is to employ PowerShell's very own commands to manipulate the registry:

```
set-executionpolicy Unrestricted.
```

Or to be more secure:

```
set-executionpolicy RemoteSigned
```

**Note 1:** Investigate with set-executionpolicy -?

PowerShell Registry

Alternatively, you could change a registry value called REG\_SZ ExecutionPolicy from Restricted to UNRestricted (or better RemoteSigned). Here is the full path to enable . PowerShell cmdlets in general and Profile.ps1 in particular.

HKLM\Software\Microsoft\PowerShell\1\ShellIds\Microsoft.Management.Automation.PowerShell

## Locating the path for the Profile.ps1 file

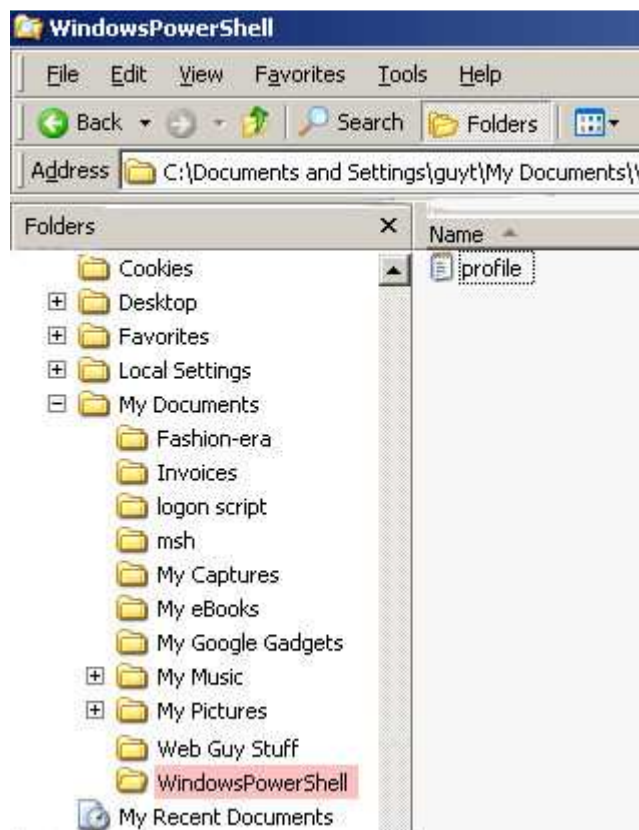
My main problem with configuring Profile.ps1 was locating the precise path shown below. In truth, it was my fault for not paying attention to detail. Note: as with all PowerShell nouns, Profile.ps1 is the singular 'profile' and not the plural 'profiles'.

**Preparation:** Navigate to your Windows directory called, 'Documents and Settings'.

First Task: Find a sub folder called: %username%\My Documents\WindowsPowerShell e.g.

C:\Documents and Settings\YourName\My Documents\WindowsPowerShell

N. B. In version 1.0 (and RC2) This folder is now called WindowsPowerShell (not PSConfiguration)



**Second Task:** Launch notepad and create a file called precisely Profile.ps1. Make sure that you have the correct file extension, .ps1 and not .txt. This is the position; you should now have see the file and folder in the above screen shot.

C:\Documents and Settings\%username%\My Documents\WindowsPowerShell\Profile.ps1

**Third Task:** In the file called Profile.ps1, add instructions to configure your command shell. I added a phrase to prove that it's my Profile.ps1 which is active and not the default Profile.ps1. "You are now..... ". PowerShell also supports the \$env:Username variable.

```
set-location d:\powershell
# welcome message
"You are now entering PowerShell : " + $env:Username
```

**Fourth Task:** Now launch PowerShell.

Once you have succeeded in placing your Profile.ps1 in the correct path, then you should see a similar command line to the screen shot to the right.

**Note:** As ever, my mission is to get you started, however, if you are interested, you can research more flashy commands to enter into this startup configuration file: Profile.ps1.



### Alternative Locations for Profile.ps1

It is possible to configure the Profile.ps1 at other locations:

%windir%\system32\WindowsPowerShell\v1.0\Profile.ps1

The above profile applies to all users and all shells.

%windir%\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell\_Profile.ps1

The above profile applies to all users, but only to the Microsoft PowerShell shell.

### Summary of Windows Powershell Profile.ps1

As you are likely to spend a great deal of time at the PowerShell command line, it makes sense to spend time configuring the startup folder and appearance of shell. The first step is to locate the location(s) where the key file called: Profile.ps1 is kept. Once you have found the correct folder, then you can add instructions to the Profile.ps1 file.

## **Twelve Real Life Tasks for PowerShell**

1. Checking the Eventlog
2. Scripting Files with get-Childitem
3. Finding text with select-String
4. Deleting Temporary Files
5. Listing Services
6. Starting a Service
7. Checking your Disk With Win32\_LogicalDisk
8. PowerShell and WMI - better than Ipconfig
9. Scripting PowerShell's - ComObject and MapNetworkDrive
10. Com – Shell Objects ('Run' Applications)
11. Active Directory
12. Creating an Exchange 2007 Mailcontact

## RL1: Checking the Eventlog with PowerShell

Let us begin by taking stock of the operating system's event logs. In our hearts, we know that we should be looking at these logs more often. We also know that when we see those red dots in the logs, we should take action to correct the corresponding error message.

Thus we have a task for PowerShell; in fact, we have a marriage made in heaven. PowerShell will help us review the system, application and other logs, while the eventlogs themselves will act as a vehicle for learning more about PowerShell's benefits, capabilities and syntax.

### PowerShell Eventlog Topics

- Example 1: Eventlog -list
- Example 2: Display error messages from your System log
- Example 3: Find Errors in the System log
- Summary of Eventlog

#### Example 1: Eventlog -list

Our first task is to discover how many different logs there are on your machine. Therefore, to discover if your computer has 3, 6, or more logs, append the -list parameter to the get-Eventlog command:

```
# Powershell script to list the event logs.  
get-Eventlog -list
```

#### Learning Points

**Note 1:** You may have guessed that the hash # symbol is PowerShell's way of introducing a comment.

**Note2:** -list is correct, please note that you do need that dash.

**Action Point:** Launch the Event Viewer, visit the actual logs and adjust the 'Retain' time and the Overflow action.

#### Example 2: Display error messages from your System log

```
# Powershell script to find Error messages in the System eventlog.  
get-Eventlog system -newest 2000 | where {$_.entryType -match "Error"}
```

#### Learning Points

I realize that most people will just copy and paste my script, but for those want to look behind the script, or those who get stuck, I provide '**Learning Points**'. My greatest joy is if you would experiment with my code, for example, change 2000 to 10000; or more adventurously, change get-Eventlog system to get-Eventlog application.

**Note 1:** You could simplify the script further and just type:

```
get-Eventlog system
```

**Note 2:** Each word, and indeed every symbol, has deep meaning to PowerShell. (|) pipes the output of the first clause into the 'Where' statement. As a result the output is filtered so that you see only error messages, and not information or warning messages.

**Note 3:** PowerShell supports a whole family of conditional statements, for example, -like, -contains, or even plain -eq (Equals), but for this job, I chose -match.

### Trusty Twosome (Get-Help and Get-Member)

Whenever you discover a new PowerShell command, it benefits from being surveyed with what I call the 'Trusty Twosome'. In this instance, if you research a verb-noun command with Get-Help and Get-Member, then you are sure to unearth new scripting possibilities. To see what I mean, apply these two commands to get-Eventlog :

#### 1) get-Help get-Eventlog

If you want to see examples of the get-Eventlog in action try:

```
help eventlog -full
```

Get-Help displays useful parameters such as: -list, -logname, and -newest. Indeed, the first thing to remember about get-Eventlog is that it needs the name of the log, for example: get-Eventlog system. Remember that PowerShell is looking for a positional argument, thus 'system' is the name of the log and is an argument, and not a parameter. To determine this difference, PowerShell expects a parameter to be introduced with a -dash, whereas an argument is preceded by only a space.

Other names of logs that you can substitute for 'system' are: Application, Security and even PowerShell itself has a log. Windows Server 2003 is likely to have yet more logs, for example, Directory Service and DNS Server.

#### 2) get-Eventlog system | get-Member -memberType property

If you wish to filter gm try: get-Eventlog system | gm -membertype property.

The above command reveals a list of properties that you can then use in the output, for example, category and source.

### Example 3: Errors in the System log

This example produces a very similar result to Example 1 above. The whole point of the extra code is to give us more control over the output. There are numerous ways that you could achieve the same list of events; indeed, many of them are technically superior to mine. However, while we are in learning mode, as opposed to production mode, I feel strongly that this script should demonstrate useful PowerShell features such as: \$Variables, pipeline, format-Table and the tiny ` backtick.

```
# Cmdlet to find latest 2000 errors in the System eventlog
$SysEvent = get-Eventlog -logname system -newest 2000
$SysError = $SysEvent | where {$_.entryType -match "Error"}
$SysError | sort eventid | `
Format-Table EventID, Source, TimeWritten, Message -auto
```

### Learning Points

**Note 1:** Guy loves variables. In PowerShell you just declare variables with a \$dollar sign. There is nothing else you need to do.

**Note 2:** The first example employed one pipeline (|), whereas this script has three (|)s. This technique of using the output of the first clause as the input of the second clause, is a characteristic benefit of PowerShell.

**Challenge 1:** I chose to sequence the data with: sort eventid. Now, I challenge you to sort on TimeWritten.

**Challenge 2:** In my opinion, it's not necessary to include entryType in the Format-Table statement, but I challenge you to add it, and then see if I am right, or see if I am wrong to omit this property.

**Challenge 3:** I used the backtick ` to run one command over two lines. You could try removing the backtick and making fewer, but longer lines. Other experiments that you could try with backtick include putting a ` at a different point in the script. Even better, try for one long but efficient command, perhaps use only one variable.

### Summary of Eventlog

I believe that PowerShell has a future. My mission is to get you started using this scripting language. What suits my learning style is concrete examples, where we learn by doing. It is my belief that a good way to begin is by employing PowerShell to tackle everyday tasks such as reviewing the eventlogs.

Just by issuing a few variations of the command 'get-Eventlog system', you will soon get a feeling of the abilities of PowerShell. Moreover, as a bonus you will soon obtain useful information about events in your operating system. The command: 'get-Eventlog application' has a wide range of switches, for example -list and -newest. What is always instructive with any PowerShell command, is get-Member, for example:  
get-Eventlog system | get-Member.



## RL2: Scripting Files with PowerShell's - Get-Childitem (gci)

Sooner or later you need a script which lists the files in a folder. In DOS we would type: 'DIR'; the nearest equivalent in PowerShell is gci. The full name behind the alias of gci is the informative, get-ChildItem. You can take the comparison further, dir /s in DOS, translates to get-ChildItem -recurse in PowerShell.

### Get-ChildItem Topics

- Trusty Twosome (get-Help and get-Member)
- Example 1 - List files in the root of the C:\ drive
- Example 2 - List ALL files, including hidden and system
- Example 3 - Filter to list just the System files
- Example 4 - The famous -recurse parameter
- Problems with -recurse, and how to overcome them

### Trusty Twosome (get-Help and get-Member)

When you discover a new PowerShell command, it benefits from being probed with what I call the 'Trusty Twosome'. Experiment with get-Help and get-Member and unlock new scripting possibilities for get-childitem. To see what I mean try these two commands:

#### 1) get-help get-childitem

(help gci) If you prefer abbreviations.

(help gci -full) If you like examples.

get-help unearths useful parameters such as -recurse, -exclude, it also helps to reveals hidden files with: -force.

#### 2) get-childitem | get-Member

(gci | gm) If you enjoy aliases.

(gci | gm -membertype property) If you are fond of filters.

get-Member reveals properties that you don't normally see in explorer, for example, CreationTime.

### Example 1 - List files in the root of the C:\ drive

Here is an example which lists all the files in the C:\ root.

```
# Powershell script to list the files in C: root
get-childitem "C:\"
```

**Note 1:** In this instance C:\ is the root and get-childitem lists any files in this directory.

**Note 2:** get-childitem "C:\" works equally well without the speech marks. For example, get-childitem C:\ However, you do need the backslash after the colon.

## Example 2 - List ALL files, including hidden and system

```
$i=0
$GciFiles = get-childitem c:\ -force
foreach ($file in $GciFiles) {$i++}
$GciFiles |sort |ft name, attributes -auto
Write-host "Number of files: " $i
```

**Note 1:** The key addition is the parameter -force. What this does is include hidden and system files.

**Note 2:** The additional commands enable us to count the files, this makes easier to see prove that -force really makes a difference. Double check what I mean by running the script with, and without, the -force switch.

## Example 3 - Filter to list just the System files

```
# PowerShell cmdlet to list the System files in the root of C:\
$i=0
$GciFiles = get-ChildItem "c:\" -force |where {$_.attributes -match "System"}
foreach ($file in $GciFiles) {$i++}
$GciFiles |sort |ft name, attributes -auto
Write-host "Number of files: " $i
```

**Note 1:** We need to employ the comparison parameter -match "System", rather than -eq "System", this is because System files also have Hidden and other attributes. Consequently, their attribute does not equal "System", although it does contain, or match the word System.

**Note 2:** You probably worked it out for your self, but just to emphasise that the variable \$i is a counter. Moreover, ++ increments \$i each time the 'where' statements makes a match. Incidentally, omitting \$=0 at the beginning produces an unexpected, or undesirable result when should you run the script for a second time.

**Challenge 1:** Repeat the command with and without -force.

**Challenge 2:** Substitute this new 'where' clause on line 3: where {\$\_.attributes -ne "Directory"}. -ne is the opposite of -eq. Thus this command filters out the directory entry.

```
# PowerShell cmdlet to list the System files in the root of C:\
$i=0
$GciFiles = gci c:\ -force |where {$_.attributes -ne "Directory"}
foreach ($file in $GciFiles) {$i++}
$GciFiles |sort |ft name, attributes -auto
# $GciFiles |get-Member
$i
```

## Example 4 - The famous -recurse parameter

Outside of PowerShell, recurse is a little known verb meaning rerun. Inside of PowerShell, -recurse is one of the most famous parameters meaning: repeat the procedure on sub-folders.

The point of this script is to list all the executables in the System32 folder. Moreover it will display the CreationTime, which can help determine if a file is up-to-date. This technique is particularly useful for troubleshooting .dll files.

```
# Powershell script to list the exe files under C:\Windows\System32
$i = 0
$DllFiles = gci "C:\Windows\System32" -recurse | ? {$_.extension -eq ".exe"}
Foreach ($Dll in $DllFiles) {
    $Dll.name + "`t " + $Dll. CreationTime + "`t " + $Dll. Length
    $i++
}
Write-Host The total number of files is: $i
```

**Note 1:** This example uses two aliases; my principle reason was to make line 4 shorter. Gci is an alias for our featured command get-childitem, and the question mark (?) is an alias for 'where'.

**Note 2:** The -recurse parameter comes directly after the name of the directory. For completeness, the location should be introduced by the -path parameter, but as this is optional, I have omitted the -path parameter in this script. However, this is how to include the -path parameter.

```
$DllFiles = gci -path "C:\Windows\System32" -recurse.
```

**Note 3:** If you investigate the file's properties, then you can spot other assets, for example: LastWriteTime, or even Length. Here is how to employ get-Member to list the properties.

```
# Powershell script to investigate file properties
get-ChildItem | get-Member -membertype property
```

## The Rest of the Item Family

- Copy-Item
- get-Item (gi)
- Move-Item
- New-Item
- Remove-Item
- Set-Item

## Problems with -recurse, and how to overcome them

### Case Study

The mission is to list all files containing the word 'Microsoft' in the Windows folder or its sub-folders. For this we use the select-string pattern matching command.

**The problem** is that this script does not work. All that happens is that we get an error: Cannot find path... Path does not exist.

```
$i=0
$Path = "C:\windows"
$Full = get-ChildItem $Path -recurse
$stringText= "Microsoft"
$list = select-string -pattern $stringText $Full
foreach ($file in $list) {$file.Path; $i++}
$i
```

**The solution:** Add the -include parameter

```
$i=0
$Path = "C:\windows"
$Full = get-ChildItem $Path -include *.txt -recurse
$stringText= "Microsoft"
$list = select-string -pattern $stringText $Full
foreach ($file in $list) {$file.Path; $i++}
$i
```

**Note 1:** When we add -include \*.txt the cmdlet works as initially planned. Actually, you could swap the famous \*. \* for \*.txt, thus : -include \*. \*

**Note 2:** -include only works when you also append the -recurse parameter.

**Note 3:** -include applies to the filename and extension. Where I made a mistake was thinking that -include would apply to the path.

### Simplification

There is no reason why you cannot simplify by removing at least two of the variables, especially on a production script. The only reason that I employed \$Path and \$StringText is that I like to isolate and control each step of the script.

```
$i=0
$Full = get-ChildItem C:\windows -include *.txt -recurse
$list = select-string -pattern "Microsoft" $Full
foreach ($file in $list) {$file.Path; $i++}
$i
```

### Summary of -Recurse

-Recurse is a classic switch to instruct PowerShell commands to repeat for sub directories. Once you learn the name -recurse and the position, directly after the directory, then it will serve you well in scripts that need to search repeatedly for information.

## RL3: Finding text with Select-String

Select-String not only opens a file but also checks for a word, a phrase, or in fact, any pattern match.

### Topics for PowerShell Select-String

- Introduction to: Select-String
- Example 1a Select-String (Pure no extra commands)
- Example 1b Select-String using variable \$Location
- Example 1c Select-String (Wildcards \*. \*)
- Example 1d Select-String (Guy's indulgence)
- Traps with Select-String
- Summary of Powershell Select-String

### Introduction to: Select-String

The first objective is to set the -pattern parameter, which defines the string we are seeking. Next comes the -path parameter, and as its name indicates, -path directs select-string to the file's location.

As you are reading this introduction, I expect you are thinking of possible applications for this select-string construction. Perhaps you wish to discover which documents contain a particular word? Alternatively, you may be seeking a more complex search-and-replace operation. My point is that while select-string may be a bit-part in a bigger drama, you still need to master its syntax and switches.

### Example 1 Select-String -path -pattern

The key to understanding select-String is studying the two main switches -path and -pattern. They say to me, 'Where is the input?' and, 'What pattern do you want to match?'

To ensure that my examples work, you and I need to agree on the file location and the pattern to search. To be successful you need to embrace one of two tactics, either mimic my folder structure and patterns, or amend my script to fit in with your environment.

My folder happens to be called : D: \powershell\stuff. My working technique is to navigate in powershell to the folder where I store the files with the cmdlet examples that I am testing. Then I just issue the .\filename command (. Dot slash followed by the name of cmdlet which stores the PowerShell instructions).

Here are three simple scripts which all produce the same result, but each employs a slightly different method. By studying all three you will gain both perspective and ideas for the best method to use in your scripts.

#### Assumptions:

You have a file called gopher.txt.

In gopher.txt is the word Guido.

## Example 1a Select-String (Pure no extra commands)

### Instructions

1. Create the appropriate file in the appropriate folder, for example gopher.txt in D:\powershell\stuff.
2. Put the word that you wish to search for in gopher.txt. My word, is 'Guido', and in the script it comes directly after the -pattern parameter.
3. Type this one line at the PowerShell command line:

```
select-string -pattern "Guido" -path "D:\powershell\stuff\gopher.txt"
```

**Note 1:** Look closely at the dashes '-' PowerShell's verb-noun (select-string) pair are joined by a dash with no spaces. The parameters -path and -pattern are introduced by a space dash then the parameter itself.

## Example 1b Select-String using variable \$Location

### Instructions

This example is more complicated in that we are going to create a cmdlet, copy and paste the code below, then execute the cmdlet from inside PowerShell.

1. Create the appropriate file in the appropriate folder, for example gopher.txt in D:\powershell\stuff.
2. Put the word that you wish to search for in gopher.txt. My word, is 'Guido', and it comes directly after the -pattern parameter in the script.
3. Copy the cmdlet, then paste it into notepad, remember to save with a .ps1 extension, for example: selectstr.ps1 .
4. Then I navigate in PowerShell to the folder where I saved the file, for example D:\powershell\stuff
5. Issue the command .\selectstr
6. If all else fails copy and paste the three lines into the PowerShell command line, then press 'Enter'.

```
# PowerShell cmdlet to find the pattern Guido
$Location = "D:\powershell\stuff\gopher.txt"
select-string -pattern "Guido" -path $Location
```

Expected outcome:

D:\powershell\stuff\gopher.txt:3:Guido is king

:3: Means line number 3

:Guido is king Refers to the line where the Pattern "Guido" occurs.

### Example 1c Select-String (Wildcards \*. \*)

It is often useful to search a batch of files. The simplest method is to use the famous \* or \*. \* wildcard.

Instructions are the same as for example 1a

```
# PowerShell cmdlet to find the pattern Guido. Note wildcard *
select-string -pattern "Guido" -path "D:\powershell\stuff\*. *. *
```

### Example 1d Select-String (Guy's indulgence)

My main idea in Example 1c is to introduce an If... Else clause to cater for instances where the - pattern cannot be found. To prepare for the 'If' logic I have introduced another variable called \$SearchStr. The second time you run this script you may wish to find and amend the "zzz" to a value that will ensure success. If you accept my challenge then you can compare the outcome of the 'If' clause with the outcome of the 'Else' clause.

Instructions are the same as for example 1a

```
# PowerShell cmdlet to find the pattern zzz.
$Location = "D:\powershell\stuff\gopher.txt"
$SearchStr = "zzz"
$Sel = select-string -pattern $SearchStr -path $Location
If ($Sel -eq $null)
{
    write-Host "$Location does not contain $SearchStr"
}
Else
{
    write-Host "Found `n$Sel"
}

Write-Host "end"
```

**Note 1:** Let us study the 'If' test: If (\$Sel -eq \$null) What this says if the value returned by Select-String is nothing (\$Null). Incidentally, there are two ls in \$null. Also, the correct syntax is -eq and not plain old =.

**Note 2:** The purpose of `n is to force a carriage return.

**Note 3:** Once the script runs successfully, amend \$SearchStr ="zzz" to a string that exists in your document.

### Footnote - More parameters:

In addition to -pattern and -path, select-string has more parameters; -include and -exclude which are useful for fine tuning the files to be searched. There is also the -caseSensitive switch to control uppercase and lowercase in the -pattern. Select-String, like most PowerShell commands, is not case sensitive by default, hence the -caseSensitive parameter.

### Traps with Select-String

The biggest trap with all PowerShell's file handling commands is looking for open-file or save-file commands. They simply don't exist. What happens is that PowerShell opens, closes and saves files automatically.

### A real-life example of Select-String

My practical problem was that I wanted to search for instances a particular string. If a file had duplicates, then I needed to know.

The PowerShell problem is how to create a stream of the files, then compare each file with my string value. In the output I needed to know the filename.

To solve the problem, I employed four main commands, which you can see in the following script:

get-ChildItem - recurse

foreach (loop) {what to do block}

if (test) {output block}

select-String -pattern to match my string value.

```
# A real-life example of PowerShell's Select-String
$i=0
$File = get-ChildItem "H:\sports" -include *. htm -recurse
$stringText= "themesguy/google"
foreach ($Entry in $File) {
    $List = select-string -pattern $stringText $Entry
    if ($List.LongLength -gt 1) {
        "{0,-8} {1,-4} {2,18}" -f
        "Files ", $List.LongLength, $Entry.FullName;
        $i++
    }
}
```



### **Learning Point**

**Note 1:** If you want to get the above script to work, focus on the \$File and \$StringVid variables. Specifically change their values to match your file location and search pattern.

**Note 2:** To find out more about the -f formatting switch consult SN4 in the Syntax section.

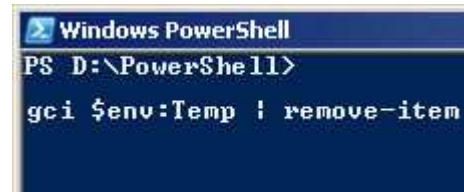
### **Summary of PowerShell's Select-String**

Select-String is a useful instruction for opening files, and then searching their contents for the phrase specified by the -pattern parameter. My advice is to get into the rhythm of the command: for example: verb-noun then -parameter . For example: Select-String -path -pattern. As expected, you can use wildcards in the path.

Once you have mastered the basic construction look real life tasks where you can employ this technique to find words or phrases that are hiding somewhere within your files.

## RL4: Deleting Temp Files

Here are step-by-step instructions to delete a user's temporary files using PowerShell commands. I will also show you how to script environmental variables, such as Temp or windir.



### Topics - Delete Temp Files Using PowerShell

- Our Mission
- Scripting Environmental Variables
- Example 1: PowerShell Script to List the Temp Files
- Example 2: List Temp Files --> Output to a File
- Example 3a: PowerShell Script to Delete Temporary Files
- Example 3b: Delete Temporary Files
- Summary - PowerShell Script to Delete Files

#### Our Mission

Our mission is to create a PowerShell script which deletes all files in a user's temp folder. The point of this exercise is to free up disk space and possibly speed up a machine's performance. What the script does is remove files left behind by programs that are unable to clear up after they close. My idea would be to run this script just after machine startup or just before shutdown.

In XP most, but not all, temp files are stored in the folder which corresponds to:

%USERPROFILE%\Local Settings\Temp

or in Vista:

%USERPROFILE%\AppData\Local\Temp

The key point is that %temp% environmental variable controls the location of this Temp folder in all Microsoft Operating systems. To cut a long story short, script this variable in PowerShell by employing: \$env:temp.

#### Trap

I do not like disclaimers. However, let me say two things: 'Firstly, take extra care with ANY script which deletes data. Secondly, abandon the usual method of mindlessly copying and pasting the code, and instead, take the time to understand each component. '

#### Addendum

Jo Enright wrote in saying that a program that he was trying to install would not work. The reason was that it needed temp files after the reboot. Sadly, the PowerShell script deleted those temp files.

#### Preparation

Download PowerShell from Microsoft's site. One interesting point is that there are different versions of PowerShell for XP, Windows Server 2003 and Vista.

## Scripting Environmental Variables

To make sense of this script you must understand Environmental Variables. Here is how to check your operating system's temp and windir variables:

Press: Windows Key +r (Run dialog box appears):

Type: %temp%.

You can also try: Start, Run, %windir%.

What I would like to do next is make the connection between what happens when you 'Run' these %variables%, with what you see in this location: System Icon, Advanced, Environmental Variables. Note in passing that Temp and Tmp occur under both the User and System variables. Also be aware that the User's %temp%, which we will use in our script, corresponds to %USERPROFILE%\Local Settings\Temp. As you may already know, in XP the %USERPROFILE% is stored under the Documents and Settings folder.

### Example 1: PowerShell Script to List the Temp Files

To gain confidence, and for the sake of safety, I want start by just listing the temporary files.

#### Instructions

**Method 1** - Copy and paste the code below into a PowerShell command box.

Left-click on the PowerShell symbol (Top Right of box), select Edit, Paste and then press Enter twice.

**Method 2 (Better)** - Copy the code below into a text file and thus create a cmdlet. Save with .ps1 extension e.g. C:\scripts\List.ps1 .

Navigate in PowerShell to C:\scripts. Type dir, you should see a file called list.ps1  
(Type) .\list [Dot backslash filename]

```
# PowerShell Script to List the Temp Files
$PathTemp = "$Env:temp"
set-location $PathTemp
$Dir = get-childitem $PathTemp -recurse
$List = $Dir | Where-Object {$_.extension -eq ". tmp"}
foreach ($_ in $List){$_ .name
$count = $count +1}
"Number of files " +$count
```

**Note 1:** It is hard to believe, but the most difficult part of this project was researching the correct syntax for : \$Env:temp.

**Note 2:** Set-location is much like CD (Change Directory)

**Note 3:** get-childitem, with used with the -recurse parameter, is like the dos /s switch.

**Note 4:** Where-Object is just a safety clause to make sure that we only list . tmp files.

**Note 5:** The key to PowerShell's loop syntax is to examine the two types of bracket (condition) {Do Stuff}

### Example 2: List Temp Files --> Output to a File

Here is extra code, which outputs the list to a file. The key point is to pipe (|) the result to out-file. Instructions, see and modify those in Example 1.

```
# List Temp Files --> Output to a File
# Warning C:\ is a silly destination
# Better edit to: "C:\scripts\ListTemp.txt"
$File = "C:\ListTemp.txt"
$PathTemp = "$Env:temp"
set-location $PathTemp
$Dir = get-childitem $PathTemp -recurse
$List = $Dir | Where-Object {$_.extension -eq ". tmp"}
foreach ($_ in $List){$Temp = $Temp + "`t" + $_.name
$count = $count + 1}
"Number of files " + $count
$Temp | out-file -filepath $File
```

**Note 1:** I hope that you changed this line: `$File = "C:\ListTemp.txt"`. It is bad practice to save such files to the root; however, my problem is that I do not know if you have a `c:\scripts` folder, so I dare not save to a non-existent folder.

**Note 2:** ``t` is PowerShell's tab command; the equivalent of `VbTab`.

### Example 3a Delete Temporary Files

As John McEnroe would say, 'You cannot be serious'. How could the tiny script below have the ability to find, and then delete all the temporary files? Part of the answer is: 'PowerShell punches above its weight; every word is loaded with meaning. The other answer is more mundane, the script does not delete ALL the temporary files because some are in use!

#### Instructions

**Method 1** - Copy and paste the code below into a PowerShell command box.

Left-click on the PowerShell symbol (Top Right of box), select Edit, Paste and then press Enter twice.

```
# PowerShell code to delete Temporary Files
get-childitem $env:Temp | remove-item -recurse -force
```

**Method 2 (Better)** - Copy the code same code (above) into a text file. Save with .ps1 extension e.g. `C:\scripts\DelTemp.ps1`.

Navigate in PowerShell to `C:\scripts`. Type `dir`, you should see `list.ps1`  
(Type) `.\DelTemp` [Dot backslash filename]

### Example 3b: Delete Temporary Files

While this script has more code than Example 3a, it is no better. However, it has the interesting feature that it displays the number of temporary files. When I ran Example 1 (Listing) I had 256 tmp files. Once I ran Example 3b, I only had 12 tmp files, and they are in needed by programs running on my machine. Thus the remove-Item command is working.

```
# # PowerShell cmdlet to delete Temporary Files. Note -force
$Dir = get-childitem $Env:temp -recurse
$Dir | remove-Item -force
foreach ($_ in $Dir){$count = $count +1}
"Number of files = " +$count
```

### Conclusion - PowerShell's Scripts to Delete Files

With PowerShell a little code goes along way. Here in RL 4 is a potent script, which is useful for deleting a user's temporary files.

### Summary of Deleting Temporary Files

As usual, I like to split a PowerShell task into chunks. My strategy is to master each chunk of code, then bolt together all the components, thus creating the final script. In the case of PowerShell, these chunks are exceedingly small. The part that took me the longest was to master the special environmental variable: \$env:Temp. Fretting about just these 9 characters looks silly now, but it does emphasise the efficiency of writing code in PowerShell.

Remove-item was another chunk of code that I experimented with in its own testbed. I am still nervous that you could mis-read or mis-apply this command, and consequently delete all of your Windows files. Consequently, do be careful, think about what you are doing and make sure that you understand the implications of the -recurse and -force switches.

## RL5: Listing the Operating System's Services with Get-Service

Our mission for Windows services is to list them, to stop them, and especially to start them. I would like to begin on this page with a thorough grounding both in the PowerShell syntax, and also the properties available to the get-Service command. Once we have mastered the basics of this verb-noun pair, we will investigate tasks for other members of the 'Service' family. For example, I have another page on **start-Service**.

### PowerShell's Get-Service Topics

- Our Mission
- Example 1: Listing all the services on your computer
- Example 2: Manipulating the Output
- Example 3: Filtering the Output with 'Where'
- Summary of PowerShell's Get-Service

#### Our Mission

One key role for a good computer techie is checking, starting, and sometimes stopping the operating system's services. Indeed, expertise with Windows Services is one discriminator between a real network administrator and a 'paper' MCSE impostor. The PowerShell scripts on this page will get you underway with my mission to monitor and control which Services should be running on your servers and workstations.

Let us build-up logically. Firstly, we will list all the services on your computer. Next we will filter the script's output so that it lists only services which are "Stopped". From there we will set about adjusting the start-up type to manual, automatic or disabled. Finally, we can create an advanced script, which will start or stop named services.

Sometimes - like now, it's hard for me to stay focussed on the one item, namely scripting with PowerShell. Instead I get distracted by checking the list of services in case any rogue malware or grayware services have crept onto my computer. Then I have another run-through the list to see if services that should be disabled, are in fact running. However, the good news is that while this sidetracks me from writing code, I am increasing my list of useful jobs to automate with PowerShell.

#### Example 1: Listing all the services on your computer

Scripting Windows Services lends itself to one of my favorite techniques, namely having the GUI open while I execute the code. The advantage of this approach is two-fold; you can check what the script is doing by observing how a value changes in the Services GUI. Also, the GUI's menus and columns give me ideas for creating better scripts. If you like this technique, click on the Windows Start button, Run, type services.msc (do remember the .msc extension).

### Instructions:

Pre-requisite: Visit Microsoft's site and download the correct version of PowerShell for your operating system.

1. Launch PowerShell
2. Copy `get-Service *` (into memory)
3. Right-click on the PowerShell symbol
4. Edit --> Paste
5. Press enter to execute the code.

### Learning Points

**Note 1:** It is said that PowerShell is a self-describing language, what this means is that you can interrogate the properties of an object directly. Don't miss a chance to experiment with my 'Trusty Twosome' of `help` and `get-Member`. In this instance try:

- a) `help get-Service -full`
- b) `get-Service | get-Member`
- c) `get-Service alerter | get-Member -memberType properties`

**Note 2:** I have yet to find a PowerShell noun that is not singular, for example, `Service` (and not `Services`). Realization of this consistency saves me typos.

**Note 3:** PowerShell commands are not case sensitive. `Get-Service` works as well as `get-Service`.

**Challenge:** Try filtering with: `get-Service S*` or use the square brackets and select a range: `get-Service [r-w]*`.

### Example 2: Manipulating the Output

Following research from `get-Service | get-Member`, we can refine the output so that we add extra properties, for example, `CanStop` and `ServiceType`. This example also illustrates how we can 'Sort' on `ServiceType` rather than the 'Name' property.

```
# PowerShell cmdlet to list Windows services
get-Service * | Sort-Object -property ServiceType `
| format-Table name, ServiceType, status, CanStop, -auto
```

### Learning Points

**Note 1:** The tiny backtick ` tells PowerShell that the same command continues on the next line. Without this symbol ( ` ) there is no word-wrap, and thus PowerShell would interpret the new line as a new command. In this instance we want one long command, which happens to spill over two lines.

**Note 2:** The `-auto` parameter produces more sensible column widths.

**Note 3:** When learning, I like to give the full syntax, however, for production scripts `Sort-Object` is normally abbreviated to plain 'Sort', just as `format-Table` is usually written as 'ft'. In fact, you can also omit the `-property` parameter and the script will still work:

```
get-Service * | Sort ServiceType | ft name, servicetype, status, canstop, -auto
```

**Challenge 1:** Research other properties, in particular more members of the Can\* family.

**Challenge 2:** Try an alternative sort criterion, for example Sort-Object Status.

### Example 3: Filtering the Output with 'Where'

In a production network, I see numerous opportunities for a short script to filter which services are running and which services have stopped. From a scripting point of view, this is a job for a 'Where' clause.

```
# PowerShell cmdlet to list services that are stopped
get-Service * | where {$_.Status -eq "Stopped"}
```

#### Learning Points

**Note 1:** The 'Where' command crops up in many PowerShell scripts, therefore it is worth familiarizing yourself with the rhythm of the command:

Begin with a pipe '|'. Next open {Braces (not parenthesis)}, pay close attention to: \$\_. These three symbols translate to 'this pipeline'. In the above example, I have chosen to filter the services based on the value of the property 'Status'. Remember that PowerShell uses -eq and not the equals sign. Finally, we have the criterion: "Stopped", an alternative filter would be, "Running".

**Challenge:** Try changing "Stopped" to "Running".

**For your notebook:** In other PowerShell scripts the structure of the 'Where' statement would be the same, however, you would replace .status with a property to suit your purpose. Each property has its own values which you would research, then substitute for "Stopped".

#### Out-file

If I could digress and hark back to the verbosity of VBScript; because it took so much code to list the services, I feared that it would confuse beginners if I added another 10 lines of VBScript in order to output the list of services to a text file. With PowerShell there is no such worry, all you need to store the results is to append these few words:

```
| out-file "D:\PowerShell\Scripts\services.txt"
```

```
# PowerShell cmdlet to save services that are stopped to a file
get-Service * | where {$_.Status -eq "Stopped"} `
| out-file "D:\PowerShell\Scripts\services.txt"
```

**Note:** As you can see, most of the command is taken up with the path which specifies where you want to create the file; naturally you must amend this path to a location on your computer. Remember to add out-file to your notebook of invaluable PowerShell commands.



### **Summary of Windows PowerShell's Get-Service**

I declare that this page well and truly covers the basics of get-Service. The examples explain how to list all the services, how to filter with a 'Where' clause, and finally, for a permanent record, how to out-file. Next step, see how to Start, Stop and Restart-Service.

## RL6: Starting an Operating System's Service with Start-Service

Our mission on this page is to start a named Windows service. If necessary, we can modify the script to stop or even restart the service. In order to get a grounding in the PowerShell syntax associated with this 'Service' family of commands, I suggest that you begin with my get-Service page.

### PowerShell's Start-Service Topics

- Our Mission
- Example 1: How to Start a Windows Service
- Example 2: How to Stop a Service
- Example 3: How to Restart a Service (Spooler)
- Summary of PowerShell's Start-Service

#### Our Mission

Our mission is to start one (or more) of your operating system's services. We can also adapt the script to stop services, but that is less exciting. It is also worth mentioning that another member of this family is called restart-Service.

The result of a preliminary experiment reveals that it's not possible to start a service whose start-up type is currently set to, 'Disabled'. Good news, a walk-through with the Services GUI reveals that if you switch a service from Disabled to Manual, then you can start it. One extra thing, you need faith in the scripters' maxim: 'Any thing that you can do by clicking in a GUI, you can equal (or exceed) in a PowerShell script'.

#### The Service Family (Each member has a different verb)

get-Service: Useful for listing the services

set-Service: Crucial parameter -startuptype

start-Service: The verb 'start' says it all

stop-Service: Handy for scripts which prevent unwanted services running e.g. Telnet

restart-Service: A nice touch by the creator's of PowerShell, no need to stop then start the service.

#### Example 1: How to Start a Windows Service (Alerter)

I choose the Alerter service for testing, partly because it's relatively harmless service, and partly because its name is near the top of the list!

##### Instructions:

Pre-requisite: Visit Microsoft's site and download the correct version of PowerShell for your operating system.

## Launch PowerShell

1. Copy the four lines of code below (into memory)
2. Right-click on the PowerShell symbol
3. Edit --> Paste
4. Press enter to execute the code.

## Preliminary Script

Let us check the service's status, and also let us 'warm up' with get-Service before we employ other members of the service family.

```
# PowerShell cmdlet to check a service's status
$srvName = "Alerter"
$servicePrior = get-Service $srvName
$srvName + " is now " + $servicePrior.status
```

## Learning Points

**Note 1:** I have decided to introduce the variable \$srvName to hold the value of the service.

**Note 2:** Observe how I mix the "literal phrases" with the \$variables and properties to produce a meaningful output.

## The Main Event - Starting Alerter

```
# PowerShell cmdlet to start a named service
$srvName = "Alerter"
$servicePrior = get-Service $srvName
"$srvName is now " + $servicePrior.status
set-Service $srvName -startuptype manual
start-Service $srvName
$serviceAfter = get-Service $srvName
"$srvName is now " + $serviceAfter.status
```

## Learning Points

**Note 1:** I prepared this script to help you appreciate the factors needed to control a Windows Service. It also reflects my thinking process of how I learn about a command. On the other hand, for a production script you could take a much more ruthless approach and simplify the script thus:

```
set-Service Alerter -startuptype manual
start-Service Alerter
```

**Note 2:** Observe how the speech marks are slightly different in this script:

```
"$srvName is now " + $servicePrior.status compared with
$srvName + " is now " + $servicePrior.status
```

My points are: a) Experiment yourself. b) To some extent, these learning scripts leave traces of my thinking process.

## Example 2: How to Stop a Service (Alerter)

In real life, you may want a script which ensures that services such as: Telnet, Messenger and Routing and Remote Access are Stopped. Just for testing, you may need a script which reverses start-Service, just to be sure that it really is working as designed. Either way, here is a script which stops the service defined by \$srvName.

```
# PowerShell cmdlet to stop the Alerter service
$srvName = "Alerter"
$servicePrior = get-Service $srvName
"$srvName is now " + $servicePrior.status
stop-Service $srvName
$serviceAfter = get-Service $srvName
set-Service $srvName -startuptype disabled
"$srvName is now " + $serviceAfter.status
```

### Learning Points

**Note 1:** Observe how this script is the mirror image of the Start-Service script. It even disables the service once it has stopped. If you remember, when Example 1 wanted to start a service, it must first make sure the -startuptype is set to manual.

In Order to Explain a Trap - I Digress:

What's in a Name? What do these groups of services have in common?

#### Group A

Alerter, Messenger, WebClient

#### Group B

Print Spooler, Telnet, Telephony and Windows Time

More importantly, why won't PowerShell's service family interact with Group B?

**The Answer:** Some services have a 'Display Name' which differs from their 'Service Name', for example Telnet and Tlntsvr. How did I find this out? When I tried to start 'Telnet' or 'Print Spooler', nothing happened. Yet if I had a manual walk-through in the Service GUI, no problem. Then I ran get-Service \* and observed the two columns, Name and also Display Name. What threw me into confusion was Group A, where both names are the same.

Just to emphasise, if you wish to control 'Print Spooler', you need to script the Name - 'Spooler'. If you double-check with the command: get-Service s\* you see Name: Spooler, Display Name: 'Print Spooler'.

### Example 3: How to Restart a Service (Spooler)

A classic service to practice the Restart-Service command is, "Spooler". One reason for choosing this particular service is that the printer gives more trouble than any other piece of hardware, and sometimes restarting the Spooler cures the problem. The inferior, but ruthless method of curing such printer jams is to reboot the computer. However, if the computer is also a server, this method is undesirable.

The real life situation of a jammed printer spooler is not straightforward. The point is that it APPEARS to be running, but in fact it's not working. The smartest solution is to restart the service. As with previous examples, when you are learning, open the services. msc GUI and experiment with the settings. What you will discover is that you can also restart a service that has stopped.

#### Production Script

All you really need is a one-liner:

```
restart-Service "Spooler"
```

#### Learning Script

```
# PowerShell cmdlet to restart the Spooler service
$srvName = "Spooler"
$servicePrior = get-Service $srvName
"$srvName is now " + $servicePrior.status
set-Service $srvName -startuptype manual
restart-Service $srvName
$serviceAfter = get-Service $srvName
"$srvName is now " + $serviceAfter.status
```

#### Learning Points

**Note 1:** My biggest fear is that in a production script I will misspell the name of the service. Thus, check for success by observing this system message:

WARNING: Waiting for service 'Print Spooler (Spooler)' to finish starting...

#### Summary of PowerShell's Start-Service

If your mission is to master the start-Service command, commence with get-Service. Once you have mastered the rhythm of the get-Service verb-noun pair, move on to the Start, Stop, and Restart-Service family of PowerShell commands. For scripting purposes, make sure that you use the true Service **Name**, and avoid the Service's **Display Name**. A final piece of advice, open the Service GUI so that you can double-check that what your script is doing is what you intended.

## RL7: Checking your Disk with Win32\_LogicalDisk

Our mission is to investigate and interrogate a computer's logical disk. Goals include discovering the size of each volume, and how much free space is available. We also have pure PowerShell goals, for example, to examine the 'Select' statement, and to control the display with -groupby, |sort and -auto.

### Topics for PowerShell Disk Check

- Trusty Twosome (Get-Help and Get-Member)
- Example 1a - Display Logical Disk Information
- Example 1b - Display Disk Size and FreeSpace
- Example 1c - PowerShell Innovations
- Example 1d - Where command added

### Trusty Twosome (Get-Help and Get-Member)

When you discover a new PowerShell command, it benefits from being surveyed with what I call the 'Trusty Twosome'. I guarantee that if you research with Get-Help and Get-Member then you will reveal new scripting possibilities for get-WmiObject Win32\_LogicalDisk. To see what I mean try these two commands:

#### 1) get-help get-WmiObject

(help gwmi) If you like abbreviations.

(help gwmi -full) If you prefer examples.

Get-help unearths useful parameters such as -class and -computerName (-computer also works).

#### 2) get-WmiObject Win32\_LogicalDisk |get-Member -memberType property

(gwmi Win32\_LogicalDisk | gm) If you enjoy aliases.

(gwmi Win32\_LogicalDisk| gm -membertype property) If you are fond of filters.

Get-Member reveals a whole host of properties that you don't normally see in Explorer.

### Example 1a - Display Logical Disk Information

Let us start with a simple script to display disk information. We begin by using just the -query parameter combined with an SQL-like select statement.

```
# Powershell cmdlet to display Logical Disk information
```

```
get-wmiobject -query "Select * from win32_logicaldisk" | Ft
```

### Example 1b - Display Disk Size and FreeSpace

In the example below, I introduce a variable called \$Item to control the output. If you would like to refine the output to your specification, then research with get-Member. The parameter -auto helps to 'tighten' the data display.

```
# Powershell cmdlet to display a disk's free space
$item = @("DeviceId", "MediaType", "Size", "FreeSpace")
# Next follows one command split over two lines by a backtick `
get-wmiobject -query "Select * from win32_logicaldisk" `
| Format-Table $item -auto
```

### Example 1c - PowerShell Innovations

- i) This example aggregates the data thanks to the `-groupby` command in the last line.
- ii) However, this example's neatest technical achievement is:  
`Select $([string]::Join(',', $item))`. This means we can use one variable, `$item`, to control both the properties for the select statement, and the output of `format-Table`.
- iii) It also introduces the idea of interrogating other machines on the network, for this it uses the `-computerName`, incidentally, `-computer` works just as well. Naturally, the first thing to alter in the script below is: `-computer YourMachine`.

```
# Powershell cmdlet to display a disk's free space
$item = @("DeviceId", "MediaType", "Size", "FreeSpace")
# Next follows one command split over three lines by a backtick `
get-wmiobject -computer YourMachine -query `
"Select $([string]::Join(',', $item)) from win32_logicaldisk" `
| sort MediaType, DeviceID | Format-Table $item -auto -groupby MediaType
```

### Example 1d - Where command added

The new feature of the example below is the 'Where MediaType' filter. By now I hope that you have started adjusting the script according to your needs.

```
$item = @("DeviceId", "MediaType", "Size", "FreeSpace")
# Next follows one command split over four lines by a backtick `
get-wmiobject -computer YourMachine -query `
"Select $([string]::Join(',', $item)) from win32_logicaldisk `
where MediaType=12" | sort MediaType, DeviceID | `
Format-Table $item -auto
```

### Summary of PowerShell with Win32\_LogicalDisk

The central feature of this page is the WMI command called: `get-WmiObject Win32_LogicalDisk`. From that base we modify the script to select properties such as, `FreeSpace` and disk size. Along the journey we sharpen our skills to filter with 'Where', and also to collate the data with `-groupby`.

## RL8: PowerShell: More flexible than Ipconfig

Scripting WMI objects with PowerShell is a particularly productive area. On this page I will illustrate PowerShell's capabilities by creating a script which is more flexible than Ipconfig.

### Topics for PowerShell: More flexible than Ipconfig

- Our Mission
- Objective 1) - List WMI Objects
- Objective 2) - WMIObject | get-Member
- Objective 3) - PowerShell: More flexible than Ipconfig

#### Our Mission

On this page we will put PowerShell to work, or mission is to extracting TCP/IP information from a machine's network adapters. But first a question: 'Is Guy reinventing the wheel? Can Ipconfig, with its numerous switches, do it all?'

To answer this question is easy, all you have to do is compare Ipconfig with my PowerShell scripts, even better, compare Ipconfig with YOUR scripts. If I could sum up the benefit of PowerShell over Ipconfig, in one word, that word would be flexibility. For example, you can customize which TCP/IP properties to display, and in addition, you can interrogate the network adapters on other machines. Can Ipconfig do that? I have not found away.

#### Powershell Objectives

WMIObject -list (parameter to enumerate possible objects)

WMIObject | get-Member (discover which properties suit our mission)

PowerShell: More flexible than Ipconfig. (Use the 'Where' clause to filter the output)

Incidentally, remember that you can run Ipconfig from inside PowerShell.

#### Guy's Advice

Either work through my learning progression by starting with Objective 1 (recommended), or else if you are in a hurry, cut to the chase, and head for Objective 3 PowerShell: More flexible than Ipconfig.

#### Objective 1) - List WMI Objects

Here is a cmdlet which identifies all the Network WMI objects.

```
$collItems = get-wmiobject -list | where {$_.name -match "network"}  
$collItems | ft name
```

#### Learning Points

**Note 1:** Because PowerShell is so simple, yet so efficient, we need fewer commands than for a corresponding VBScript. Indeed, the bare minimum to get started, and display all the WMI object is



these 8 letters with a dash in the middle:

```
gwmi -list # gwmi is an alias for get-wmiobject
```

**Note 2:** Observe the (|) pipe. You will use this symbol hundreds of times in PowerShell, what happens is the output of the wmiobject list is pumped into the where clause, which pulls out all the entries that contain the word "network". My thinking was gwmi-list produces too many objects.

**Note 3:** I deliberately don't use many aliases in my scripts, but ft (format-Table) is useful for controlling the display of PowerShell's output. Do try the above script with and without the last word, name.

## Objective 2) - WMIObject | get-Member (Discover which properties to use in our mission)

This script identifies the properties available for the object: Win32\_NetworkAdapterConfiguration. Here below is a classic example of the get-Member construction.

```
$collItems = get-wmiobject -class "Win32_NetworkAdapterConfiguration"
$collItems | get-Member -membertype Property
```

### Learning Points

**Note 1:** Observe how -class homes in on the object that we are interested in, namely: Win32\_NetworkAdapterConfiguration.

**Note 2:** If you wished to manipulate the TCP/IP settings, you could investigate which methods are available by substituting 'Method' for the last word, 'Property'. Alternatively, you could omit the -membertype phrase altogether.

## Objective 3) - PowerShell: More flexible than Ipconfig

Here is the main script illustrating PowerShell's ability to display information about your TCP/IP properties.

```
$strComputer = "."
$collItems = get-wmiobject -class "Win32_NetworkAdapterConfiguration" `
-computername $strComputer | Where{$_.IpEnabled -Match "True"}
foreach ($objItem in $collItems) {
    write-Host "MAC Address : " $objItem.MACAddress
    write-Host "IPAddress : " $objItem.IPAddress
    write-Host "IPAddress : " $objItem.IPEnabled
    write-Host "DNS Servers : " $objItem.DNSServerSearchOrder
    Write-Host ""
}
```

**Note 1:** The properties that I have chosen are not important. This is just an example to get you started; it would make my day if you substituted properties that you researched from Example 2: Get-Member, for example, . DefaultGateway or . IPSubnet instead of . MACAddress.

**Note 2:** My old friend 'Barking' Eddie thought that my printer needed cleaning when he saw ` in my script. Actually this character (`), found at the top left of the keyboard, is called a backtick. What the backtick does is tell PowerShell – 'This command continues on the next line'. Just to be clear, this character corresponds to ASCII 096, and is not a misplaced comma!

**Note 3:** To loop through all the network adapters, I chose the ForEach construction. In PowerShell in general, and ForEach in particular, the type of bracket is highly significant; (ellipses for the condition) and {curly for the block command}.

**Note 4:** Most of my scripts employ the Where clause to filter the output. In this case I wanted to discard any virtual network cards.

**Note 5:** I almost forgot, \$strComputer controls which computer you are analysing. Again, it would make my day if altered ". " to the hostname of another machine on your network.

### **Summary of PowerShell: More flexible than Ipconfig**

The aim of this script is to show you that PowerShell can not only mimic Ipconfig, but also exceed its capabilities. In particular, I wanted to show how PowerShell could customise list of TCP/IP properties in its output. It would also be useful to create a script which can display TCP/IP information from other machines on the network.

## RL9: Scripting PowerShell's ComObject and MapNetworkDrive

ComObject, or plain COM, increases the range of PowerShell activities. One way of looking at COM objects is as a mechanism for PowerShell to launch programs, for example, mimicking the RUN command. Another way of looking at ComObjects is performing the role previously undertaken by VBScript. For both those tasks, scripting with COM objects gives you a rich selection of options. The bonus of using PowerShell rather than VBScript is that you need fewer commands.

### Topics for COM Objects

- New-Object -com (MapNetworkDrive)
- Selection of -com Applications
- COM and Shell.Application
- VBScript - MSScriptControl.ScriptControl
- Com and Active Directory
- Summary of Com Objects

### New-Object -com

The secret of manipulating COM objects is the command: New-Object -COM. What comes next depends on which type of object you need. Here are examples of creating, then manipulating ComObject with \$variables:-

#### Launch Windows PowerShell

Type these commands at the PS> Prompt (or copy and paste my examples)

#### 1) MapNetworkDrive

```
$net = New-Object -com WScript.NET work  
$net. mapnetworkdrive("Y:", "\\server\share")
```

**Note 1:** Naturally, you need to amend \\ server\share to the name of a real UNC share on your network.

**Note 2:** To see if your PowerShell script performs as planned, launch Windows Explorer.

#### 2) Internet Explorer

```
$ie = New-Object -ComObject InternetExplorer.Application  
$ie.visible=$true
```

Incidentally, -com and -ComObject appear to be interchangeable.

New-Object -com | get-Member

Let us investigate the properties of the -com object. At first, get-Member appears not work, even help seems unsupportive. Fortunately, all that is missing is the name of the -com object that you wish to research. For example:

```
New-Object -com wscript.NET work | get-Member.
```

**Results:**

Name	MemberType (Method, Property)
AddPrinterConnection	Method
AddWindowsPrinterConnection	Method
EnumNetworkDrives	Method
MapNetworkDrive	Method
RemoveNetworkDrive	Method
SetDefaultPrinter	Method
ComputerName	Property
Organization	Property
Site	Property string
UserDomain	Property
UserName	Property
UserProfile	Property

**Selection of -com Applications**

Create objects with new-Object -com xyz.application

**1) Shell.Application example:**

```
$shell = new-Object -comObject Shell.Application
```

**2) WScript.Shell example**

```
$WSH = New-Object -com WScript. Shell
```

```
$WSH.Run("Explorer.exe")
```

Instead of "Explorer", substitute "Excel", "Word", "Calc" or any other registered application.

**3) MSScriptControl.ScriptControl**

VBscript example

```
$VBScript = new-Object -com MSScriptControl.ScriptControl
```

```
$VBScript.language = "vbscript"
```

```
$VBScript | get-Member
```

**4) InputBox() Example**

```
$VBScript = new-Object -com MSScriptControl.ScriptControl
```

```
$VBScript.language = "vbscript"
```

```
$VBScript.addcode("function getInput() getInput = inputbox(`"Guy Says Hello`",`"Guy's box`") End Function" )
```

```
$Input = $VBScript.eval("getInput")
```

**Note:** This is a complex example because it creates a function, which then does useful work in creating an `inputbox()`.

### **Summary of -COM objects**

The `ComObject` family of commands add important capabilities to PowerShell. For example, creating network objects means that you don't have to revert to VBScript when you need to map network drives. Another way of looking at the `-ComObject` command is as a PowerShell method of accessing the Run dialog box programmatically.

## RL10: Scripting - COM Shell Objects (Run Applications)

On this page I will show you how to create a COM object, which opens and then manipulates Windows Explorer.comObject, or plain COM, is a key PowerShell command that performs many of the jobs previously undertaken by VBScript. For our task, we are going to persuade PowerShell to create a Shell.Application; from there we will manipulate the Explorer programmatically.

### Topics for COM Objects

- New-Object -com
- PowerShell script to Open (launch) the Explorer
- Summary of Com Shell Objects

### New-Object -com

All COM objects are created through the command: New-Object -COM. There are dozens of options and possibilities for New-Object -COM, for our purpose we specifically need a Shell.Application type of object. Let me take you step-by-step through the method.

#### 1) Create the object (Shell.Application)

The first step is to create an object and assign it to a variable. For example:

```
$ShellExp = new-Object -comObject Shell.Application
```

#### 2) Object Methods and Properties

Let us investigate the methods and properties available to our shell object:

```
$ShellExp | get-Member
```

In particular, lookout for the methods: 'Open' and 'Explore', because these are the methods that we are going to apply to our object.

Name	MemberType
-----	-----
AddToRecent	Method
BrowseForFolder	Method
CanStartStopService	Method
CascadeWindows	Method
ControlPanellItem	Method
EjectPC	Method
Explore	Method

ExplorerPolicy	Method
FileRun	Method
FindComputer	Method
FindFiles	Method
FindPrinter	Method
GetSetting	Method
GetSystemInformation	Method
Help	Method
IsRestricted	Method
IsServiceRunning	Method
MinimizeAll	Method
NameSpace	Method
Open	Method
RefreshMenu	Method
ServiceStart	Method
ServiceStop	Method
SetTime	Method
ShellExecute	Method
ShowBrowserBar	Method
ShutdownWindows	Method
Suspend	Method
TileHorizontally	Method
TileVertically	Method
ToggleDesktop	Method
TrayProperties	Method

UndoMinimizeALL	Method
Windows	Method
WindowsSecurity	Method
Application	Property
Parent	Property

### 3) PowerShell script to actually Open (launch) the Explorer

Instructions:

Save the code as a file, make sure that you create a .ps1 extension, for example ShellExplore.ps1

Launch PowerShell

Navigate to the folder where you saved the .ps1 file

At the PS> prompt type: .\ShellExplore (dot backslash filename)

```
# ShellOpen.ps1
# Opening Explorer using PowerShell
# Author Guy Thomas http://computerperformance.co.uk/
# Version 1. 3 - November 2007
# Launches the Explorer
$ShellExp = new-Object -comObject Shell.Application
$ShellExp.open("C:\")
```

#### Learning Points

When I first experimented with this command I tried \$ShellExp.open without the brackets - wrong. Then I tried \$ShellExp.Open() - no good. Finally I remembered that the parenthesis style of brackets needs to enclose a value, \$ShellExp.Open("C:\"). Eureka, success, the Windows Explorer launched anchored at the C:\.

### 4) PowerShell script to Explore with the Windows Explorer

The idea behind a second version of opening Windows Explorer is to give you perspective. By changing a few items, I hope that it gives you extra understanding, also more ideas for your own situation. In the example below I have introduced a variable \$Drive to hold the value for the folder, which you want explorer to view. Note also how I have changed .open("D:") to .explore("C:\windows"). For this script to work, you need to have a \windows folder on your c: drive, fortunately, this is the default location for this system folder.



```
# ShellExplore.ps1
# Opening Explorer using PowerShell
# Author Guy Thomas http://computerperformance.co.uk/
# Version 2.5 - October 2007
# Sets the Drive
$Drive = "C:\windows"
# Launches the Explorer
$ShellExp = new-Object -comObject Shell.Application
$ShellExp.explore($Drive)
```

### Summary of -COM Shell objects

Once you have discovered the straightforward technique of creating com objects, then you can specialise by creating a Shell.Application object. After you have assigned the object to a variable, you can apply methods to perform useful tasks such as opening folders or exploring with Windows Explorer. The secret of this method is adding a value in the brackets at the end of the command, for example, `$ShellExp.Open("C:\")`.

## RL11: Active Directory and PowerShell

What makes scripting Active Directory tricky is that we need so many different skills. However, if you are a beginner don't worry, very little knowledge is assumed. If you are experienced with PowerShell's commands you may prefer to jump straight to Example 4.

### Topics for PowerShell and Active Directory

- ADUC (Active Directory Users and Computers)
- Example 1: Simple Script to Echo the Active Directory Domain
- Example 2: To Count the Objects in Your Active Directory
- Example 3: Adding a Where Clause
- Example 4: Adding a Foreach Loop

### Skills Checklist

- Active Directory Users and Computer (GUI)
- LDAP - ADSI Edit
- PowerShell's | Where clause, Foreach loop and New-Object DirectoryServices

### ADUC (Active Directory Users and Computers)

Scripters are born looking for shortcuts. Their very first action was probably to copy and paste someone else's script. What I cannot understand therefore, is why scripters as a breed are so unwilling to use GUIs. It's as though a GUI is their enemy, or a cheat method that they dare not touch.

I take the view that examining the corresponding GUI compliments my script. Perhaps I am addicted to using both in tandem, because every time I have a walk-through with the GUI, the menus give me ideas for a better script. There is also the point that inspecting the object's properties using ADUC, provides proof that the script has indeed executed as intended. Or more likely, that the script has not worked, but observing the results helps me to troubleshoot a problem with the code.

What ADUC also alerts us to is the true domain name. Is your domain name plain YourCompany, or does it have an extension, for example, YourCompany.com? In passing, could I remind you that in LDAP dc= means domain context, and not domain controller.

A general inspection of a User's property sheet will reveal dialog boxes labelled: First name, Last name and User logon name. It is these fields (or similar) that I recommend we revisit after running the script. The next connection to make is the relationship between say 'Last Name' and the object property called 'sn'. In fact, they are one and the same, but how did I know that? The answer is to spend time researching, or just exploring with ADSI Edit.

## LDAP - As Revealed by ADSI Edit

The ADSI Edit utility, which is found on the Server CD will reveal the connection between property names, for example:

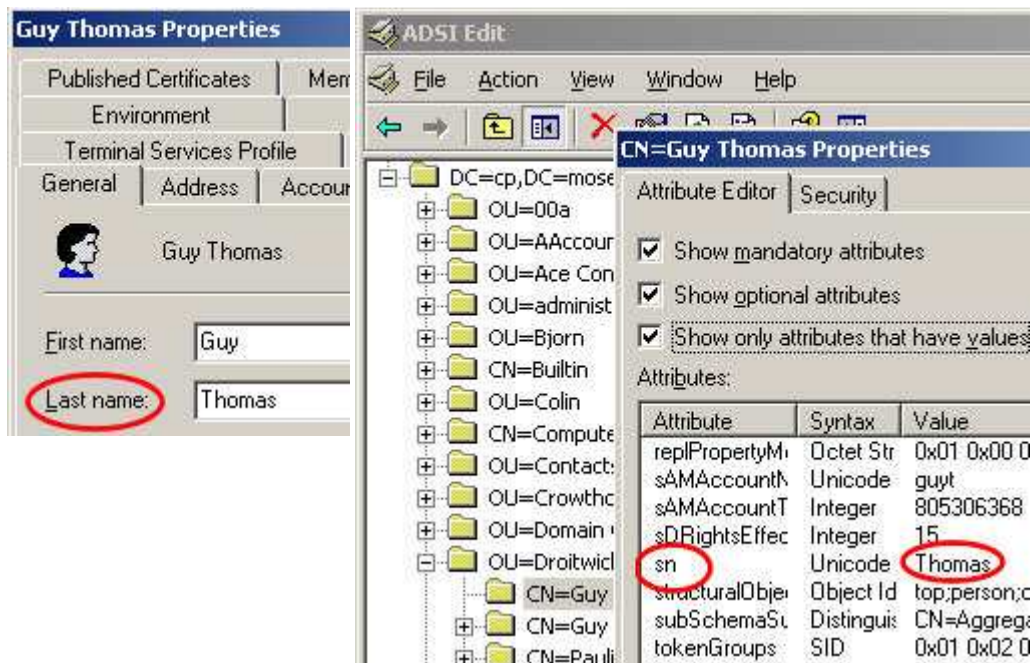
GUI - You see, User logon name:

LDAP Property - You script: sAMAccountName

### Guy's LDAP Learning Technique

My mission, as always, is to get you started. So here is simple technique that I employ when I am unsure of which LDAP name to include in my script. Let us hope that you are practising on a test domain. If you only have a live domain, then at least create a test OU with test users.

The secret is to change a value in the GUI and then see if you can find the very same string in ADSI Edit. If so, then you have learned the equivalence between the GUI menu, and the LDAP property. For example, go to the Last name of your test user, enter 'Thomas', now search through ADSI Edit until you find 'Thomas'. The conclusion is that, Last name: corresponds to 'sn'.



There is an alternative, and that is to get a list of LDAP properties, which you keep by your side when scripting.

### PowerShell Script which connects to Active Directory

#### Launch PowerShell

- Copy the lines of code below (into memory)
- Right-click on the PowerShell symbol
- Edit --> Paste
- Press enter to execute the code.

## 2) Check Your Domain Name

To double-check that your domain is what you think it is, launch ADUC and see whether your domain is one word, or whether it has an additional, .com or .local extension.

### Example 1: Simple Script to Echo the Active Directory Domain

```
# PowerShell Connects to Active Directory
# Connect to hard-coded root
# Author: Guy Thomas
# Version 1. 3 Sept 2007 tested on PowerShell v 1.0 and RC2

$Dom = 'LDAP://DC=cp;DC=mosel'
$Root = New-Object DirectoryServices.DirectoryEntry $Dom
Write-Host "PowerShell connects to domain: $Dom"
```

#### Learning Points

**Note 1:** 'LDAP://DC=cp;DC=mosel'. Rather than using the traditional .local namespace for non-internet domains, I prefer .mosel merely because it happens to be the road where I live! Naturally you changed the value for this \$Dom variable in your live script? Didn't you?

**Note 2:** New-Object is such an insignificant command, yet it is vital for creating objects, which we can use for connecting connect to Active Directory.

**Note 3:** DirectoryServices. DirectoryEntry is one of the key commands to connect to Active Directory. I think of this as a pipeline to the root of my domain's namespace.

**Note 4:** I realize that Example 1 is short. Also from a design point of view it does not achieve much. If you are familiar with PowerShell, then jump to Example 4; else, stick with my master plan to build up gradually and go to Example 2.

### Example 2: To Count the Objects in Your Active Directory

Pre-requisite. Once again, change \$Dom to reflect your domain, and not mine.

```
# PowerShell Counts objects in Active Directory
# Connect to hard-coded root
# Author: Guy Thomas
# Version 1. 5 Sept 2007 tested on PowerShell v 1.0 and RC2

$Dom = 'LDAP://DC=cp;DC=mosel'
$Root = New-Object DirectoryServices. DirectoryEntry $Dom

# Create a selector and start searching from the Root of AD
$selector = New-Object DirectoryServices. DirectorySearcher
$selector.SearchRoot = $root
$adobj= $selector.findall()
"There are $($adobj.count) objects in the $($root.name) domain"
```

## Learning Points

**Note 1:** If you get a result greater than zero, then your script is working. If the number is blank, then check for a typo in your \$Dom domain name. The following result, would mean an error with \$Dom:

There are    objects in the domain. Whereas, 'There are 173 objects in the domain'. Means you edited \$DOM to reflect YOUR domain name.

**Challenge:** One of the best ways of learning is to see if you can alter the script, and still get a meaningful result. My challenge is to amend the script to count only the objects in the Users container. Thus amend, or redefine \$DOM to include CN=Users;.

```
$Dom = 'LDAP://CN=Users;DC=cp;DC=mosel'
```

If you examine ADUC, the yellow Users folder does not have the tiny OU icon / motif, thus is a container object and not actually an Organizational Unit. Hence we use CN=User and not OU=User.

## Example 3: Adding a Where Clause

Pre-requisites

Practice with this clause: | Where{\$\_.xyz} until you understand its syntax.

Research LDAP properties with ADSI Edit. In particular, research the possible values for objectCategory.

```
# PowerShell Counts Person Objects in Active Directory
# Author: Guy Thomas
# Version 2. 3 Sept 2007 tested on PowerShell v 1.0 (RC2)
$Dom = 'LDAP://DC=cp;DC=mosel'
$Root = New-Object DirectoryServices. DirectoryEntry $Dom
# Create a selector and start searching from the Root of AD
$selector = New-Object DirectoryServices. DirectorySearcher
$selector.SearchRoot = $root
$adobj= $selector.findall() | `
where {$_.properties.objectcategory -match "CN=Person"}
"There are $($adobj.count) objects in the $($root.name) domain"
```

## Learning Points

**Note 1:** The key parameter, or switch, is -match. Again, to truly understand how it works, try substituting -like for -match. What you find is that with -like you need to add the wildcard \*, for example, "CN=Person\*".

**Note 2:** To see why I choose 'Person' and not 'User', try a simple substitution, "CN=User"

**Challenge:** To learn more about LDAP properties try this:

```
where {$_.properties.objectclass -match "User"}.
```

My point is that objectClass is different from objectCategory. Bizarrely, 'User' includes Computers as well as User accounts. Furthermore, I truly believe that learning PowerShell will teach you more about Active Directory.

### Example 4: Adding a Foreach Loop

Pre-requisites

Experiment with the foreach loop in isolation so that you understand its mechanism.

Research more LDAP properties. For example the relationship between, GivenName and (First name), and between SN and (Surname), also CN and (Full name).

```
# PowerShell Displays Firstname and surname of Users
# Author: Guy Thomas
# Version 4. 3 Aug 2007 tested on PowerShell 1.0
$Dom = 'LDAP://DC=cp;DC=mosel'
$Root = New-Object DirectoryServices. DirectoryEntry $Dom
$i=0
# Create a selector and start searching from the Root of AD
$selector = New-Object DirectoryServices. DirectorySearcher
$selector.SearchRoot = $root
$adobj= $selector.findall() | `
where {$_.properties. objectcategory -match "CN=Person"}
foreach ($person in $adobj){
$prop=$person.properties
$i++
Write-Host "First name: $($prop.givenname) " `
"Surname: $($prop.sn) User: $($prop.cn)"
}
"Total $i"
```

### Learning Points

**Note 1:** Let us examine what is inside the foreach loop. Firstly, we create a new variable called \$prop to hold the object's properties. Next we write-Host, the properties that we are interested in starting with givenName.

**Note 2:** I added a loop counter, \$i. At the top of the script I set the value to zero, then I increment with \$i++.

**Challenge:** Part of the reason that I added to the \$i loop counter is to compare different scripts. For instance, if you add an 'if' filter this should drastically reduce the number of objects. Here is my challenge, add this code after the \$prop and before \$i++

```
if ($prop. sn -ne $Null){
```

**Important:** Add a balancing closing bracket }. Placing a second bracket before the existing} will do nicely.

### **Summary of PowerShell and Active Directory**

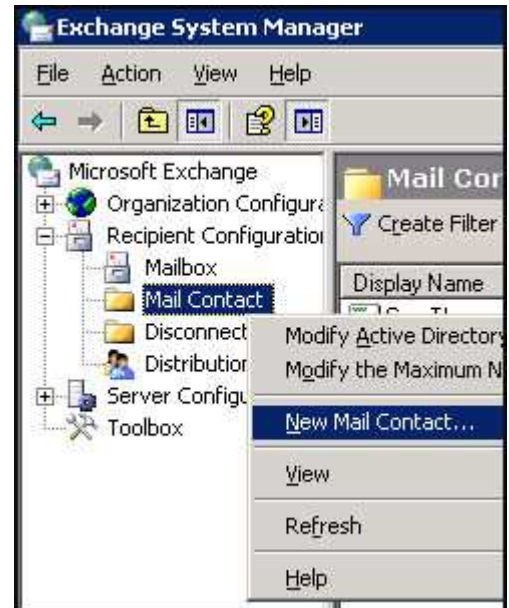
It was with much relief that I discovered that PowerShell supplied a mechanism to query Active Directory. The secret is starting with `New-Object`, and then choosing the following COM objects: `DirectoryServices.DirectoryEntry` and `DirectoryServices.DirectorySearcher`. How easy you find the rest of my script depends on your experience of pure PowerShell techniques, for example `foreach` loops, and `'Where'` clauses.

## RL12: Creating an Exchange 2007 Mailcontact

This is the last page of the 'Real Life' section. My aim is to complete the section with an Exchange 2007 example. If you don't have Exchange 2007, then I suggest that you just skim through the scripts. Hopefully, you will take away the idea that configuring Exchange 2007 with PowerShell will be straightforward, especially as there are likely to be lots of scripts on the internet which you can amend easily to suit your organization.

This example explains how to create Contact and Mailcontact objects in Exchange 2007. If you are looking for a straightforward example to learn Exchange 2007 PowerShell commands, then Mailcontact is an ideal vehicle to practise. The key verbs are: get, new and disable.

Exchange 2007 beta uses the Monad command shell, I believe that in the final product this will be replaced with the PowerShell scripting interface.



### Topics for Mailcontact

- Walk-through
- Get-Mailcontact
- New-Mailcontact
- Disable Mailcontact
- Exchange PowerShell Nouns
- Exchange PowerShell Verbs
- Summary of Mailcontact

### Walk-through with Exchange System Manager

To begin with, I recommend a walk-through with Exchange 2007 System Manager. Firstly it will clarify the objects that we are creating; secondly it will alert you to nuances such as Aliases don't have spaces.

### Get-Mailcontact

As a prelude, open the PowerShell command prompt and type:

```
get-Mailcontact (Only Mail enabled objects)
get-contact (Includes disabled contacts)
```

Type these commands after you have created a new mail object with the GUI, and again after you employ PowerShell to create the Mail Contact.

**Note:** As with so many PowerShell nouns contact is both case insensitive and singular.



## New-Mailcontact

Our mission is to achieve with PowerShell what you could do manually with the Wizard (See Right). Once you issue the plain command New-Mailcontact, you need to provide information about:

ExternalEmailAddress:

Name:

Alias:

OrganizationalUnit:

### Example 1 Create a New Mailcontact called Eddie Jones"

New-Mailcontact

ExternalEmailAddress: ernie@guy-sports.com

Name: Ernie Prost

Alias: ErnieProst

OrganizationalUnit: Droitwich

Once created you should see your new Mailcontact. Try viewing all such objects try:

get-Mailcontact | ft -autosize name, alias, ExternalEmailAddress.

Note: ft is an alias or shorthand for format-Table.



## Disable-Mailcontact

Disabling is different from deleting the object. Disabled Mail Contacts appear in the Exchange System Manger and can be verified by comparing Get-Mailcontact with Get-Contacts. You can also confirm by checking the contents of the Disconnected folder in the Exchange System Manager.

### Example 2 Disable a Mailcontact

Disable-Mailcontact 'Eddie Jones'

Note: For names with spaces, enclose them with single quotes.

Test your command with Get-Contact.compare the results with Get-Mailcontact

## Summary of Mailcontact

Mailcontact is an Exchange 2007 object and a Powershell noun. To control Mailcontact, master the verbs get, new and disable. While the first ten of my 'Real life' examples work on any Windows machine, these last two examples require Active Directory and Exchange 2007.

## Syntax Section

- 1) Brackets
- 2) Conditional Operators
- 3) Format Table
- 4) -f Format
- 5) Group
- 6) If construction
- 7) Out-File
- 8) Parameters and Switches
- 9) Pipeline
- 10) Quotation Marks
- 11) Speech Marks
- 12) WhatIf

Newbies will find useful hints and tips in each of these twelve syntax examples. If you work your way through each topic then you will gain a feel for the rhythm of the commands, and the nuances of the punctuation marks.

For those with intermediate knowledge these examples provide a refresher for that moment when your code is not working because you have forgotten the syntax.

## SN1: Type of Bracket

Did you ever buy a light bulb with a bayonet socket, only to discover that you really needed a bulb with a screw fitting? Well, PowerShell brackets require similar attention to detail otherwise your script will also remain in the dark.

Windows PowerShell employs four types of bracket, (parenthesis, also called curved) {braces, sometimes called curly} and [square]. Occasionally you may also see <angle> brackets. My point is that when you use PowerShell commands, each type of bracket has a particular meaning and significance.

At first sight, one type of bracket seems much like another, but as you gain experience with a variety of PowerShell constructions, you begin to tune-in to the differences. Eventually, you reach a level of expertise where it seems that the very type of PowerShell bracket is trying to tell you something of significance. The bottom line is that if we employ the wrong type of bracket, instead of executing our commands, PowerShell presents us with an error message.

### Types of PowerShell Bracket

- Parenthesis Bracket ()
- Braces Bracket {}
- Square Bracket []
- Summary of PowerShell Brackets

#### 1) Parenthesis Bracket ()

When a PowerShell construction requires multiple sets of brackets, the parenthesis (style) usually comes first. Parenthesis brackets also come first in importance because these (curved) brackets are used for compulsory arguments. Let us take a foreach loop as an example. The (input definition) is the most important element; it comes first and is enclosed by parenthesis. Observe that the {braces} style of bracket, comes second and inside the braces is a {ScriptBlock}, which dictates the code to execute.

#### Example 1: To demonstrate (parenthesis) and {braces} brackets

```
# PowerShell cmdlet to demonstrate brackets
$WmiClass = "Win32_NetworkAdapterConfiguration"
$collItems = get-wmiobject -class $WmiClass
foreach ($objItem in $collItems)
{
    write-Host $objItem.caption
    write-Host $objItem.ipaddress
    write-Host $objItem.macAddress
    write-Host
}
```

**Note 1:** Observe how the parenthesis brackets enclose the compulsory statement (\$objItem in \$colItems), which starts the foreach loop.

**Note 2:** Find the script block, enclosed by {braces}. This clause is important because it determines what to do inside each loop. Where the script block contains multiple lines, each half of the bracket has its own separate line, thus emphasising the action section of the script.

## 2) Braces Bracket { }

A common place to pay attention to the braces style of PowerShell bracket is when you initiate a 'Where' clause.

```
get-Wmiobject -list | where {$_.name -match "win32*"}
```

**Note 1:** Remember to the use of the | pipeline, because this streams the output of 'get-Wmiobject -list', and passes it as input to the 'where' clause.

## 3) Square Bracket [ ]

The word which best sums up PowerShell's square bracket is: optional. Anytime you want to tag on an optional command or parameter, select the [square] bracket.

### Example 3a

List all the processes beginning with the letter 's'

```
get-process [s]*
```

### Example 3b

Wildcards in square bracket can produce unexpected results. It's just a matter of trial and error and also you need to adjust to PowerShell's logic; [s-t] means beginning with 's', or beginning 't'. '[SVC]' means beginning with 'S' or 'V' or 'C' and not beginning with specifically 'SVC.... '.

```
get-process [r-s]*
```

**Note 1:** Experiment with different letter combinations, thus become expert at using the hyphen-filter.

**Note 2:** Pay attention to the wildcard asterisk\*. See what happens if you omit the \*. Try: get-process [r\*-s], or get-process [r-s]\*. Did they produce what you expected? I was disappointed with [r\*-s], but on reflection perhaps it was a foolish request.

## Summary of PowerShell's Brackets

In a nutshell, the type of bracket is highly significant in Windows PowerShell. Take the time to tune-in to the personality of each style of bracket. Parenthesis() come first, both in sequence and in importance. Braces {} play a specific role in 'Where' statements and also in looping constructions. Lastly, PowerShell's square [] brackets are used to control optional parameters. The most important lesson is that each type of bracket has a particular role; you must choose the correct bracket for the particular scripting task.

## SN2: Conditional Operators

I think of PowerShell's conditional operators as filters. Perhaps you are suffering from a common problem - too much information? If so, then these Powershell operators: -match, -like and -contains will help you to distil the key information.

### Introduction to: -match -like -contains.

-Match -like and -contains are all similar PowerShell conditional operators, yet each has a subtle specialization. My best advice is keep experimenting until you find the particular conditional operator that suits your circumstance. Have faith that you will be able to manipulate one of this trio, and thus achieve the degree of filtering that you seek. Also note, these operators are in addition to the ubiquitous, -eq, 'If' and 'elseif' constructions.

### Topics for PowerShell's Conditional Operators

- Example 1 -Match
- Example 2 -Like
- Example 3 -Contains
- Summary of Conditional Operators

#### Example 1 -Match

The 'match' can be anywhere in the string, it have to be at the beginning. Moreover, the pattern does not have to be a complete match. -Match uses regular expressions for pattern matching. Incidentally, -match and the other PowerShell conditional operators all have a negative form, for example -notmatch.

##### Example 1a - Does not have to be at the beginning

```
$Path ="Guy Thomas 1949"  
$Path -match "Th"  
Result PS> True
```

##### Example 1b - Completely wrong name

```
$Path ="Guy Thomas 1949"  
$Path -match "Guido"  
Result PS> False
```

##### Example 1c - Wrong date

```
$Path ="Guy Thomas 1949"  
$Path -match "1948"  
Result PS> False
```

### Example 1d - Wildcard ? Rides to the rescue

```
$Path = "Guy Thomas 1949"  
$Path -match "194?"  
Result PS> True
```

### Example 1e - Wmiobject and Where

```
get-wmiobject -list | where {$_.name -match "cim*"}
```

Negative -notmatch

For negative conditions, there is an alternative form called -notmatch.

### Example 2 - Like

With -like, both sides of the expression have to be the same, fortunately, you can employ the usual wildcards \* and ?.

#### Example 2a - Having only part of the string is no good

```
$Path = "Guy Thomas 1949"  
$Path -like "Th"  
Result PS> False
```

#### Example 2b - Just the start of the string not enough

```
$Path = "Guy Thomas 1949"  
$Path -like "Guy"  
Result PS> False
```

#### Example 2c - Wildcard \* is useful

```
$Path = "Guy Thomas 1949"  
$Path -like "Guy*"   
Result PS> True
```

#### Example 2d - Wildcard \* but the rest has to be correct

```
$Path = "Guy Thomas 1949"  
$Path -like "Gzkuy*"   
Result PS> False
```

#### Example 2e - \* Wildcard \* Double wildcards are handy

```
$Path = "Guy Thomas 1949"  
$Path -like "*Th*"   
Result PS> True
```

If your logic needs the negative, then try -notlike. In addition, -notlike and -like have variants that force case sensitivity. -clike and -cnotlike.

### Difference Between -like and -match

```
$Path = "Guy Thomas 1949"
```

```
$Path -like "Th*"
```

```
Result PS> False
```

```
$Path = "Guy Thomas 1949"
```

```
$Path -match "Th*"
```

```
Result PS> True
```

### Example 3 -Contains

The conditional operator -contains is similar to -eq, except it returns 'True' or 'False'. -contains is designed for situations where you have a collection and wish to test one particular item.

#### Example 3a - Checks each item between the commas

```
$name = "Guy Thomas", "Alicia Moss", "Jennifer Jones"
```

```
$name -contains "Alicia Moss"
```

```
Result PS> True
```

#### Example 3b - Needs exact equality

```
$name = "Guy Thomas", "Alicia Moss", "Jennifer Jones"
```

```
$name -contains "Jones"
```

```
Result PS> False
```

#### Example 3c - Wildcards do no good

```
$name = "Guy Thomas", "Alicia Moss", "Jennifer Jones"
```

```
$name -contains "*Jones"
```

```
Result PS> False
```

### Summary of PowerShell Conditional Operators

So often we suffer from information overload. Working with Powershell is no different, however it does supply three conditional operators to filter your information: -match, -like and -contains. Each operator has different properties; with research, you can get just the filter you need, and thus filter the desired stream of information into your script's output.

## SN3: Format-Table (ft)

Format-Table, or ft for short, controls the output of your Windows PowerShell commands. Whenever presentation of information is important, pipe the script's output into format-Table. On this page I will show you examples of this straightforward, but important command.

### Windows PowerShell Format-Table Topics

- Format-Table - Simple Examples
- Script (cmdlet) Technique
- Format-Table - Intermediate examples
- Format-Table - Advanced examples
- PowerShell Perspective
- Format-Wide
- Format-List
- Summary of Format-Table in Windows PowerShell

### Format-Table - Simple Examples

Scenario: we wish to investigate processes running on an XP workstation or a Windows Server.

#### Pre-requisites:

PowerShell and .NET Framework are installed on your XP or Windows Server

You are at the prompt PS>

#### Example 1a

Let us get started, type:

get-process

#### Example 1b

Now let me introduce format-Table so that we can control the output columns.

```
get-process | format-Table processName, handles -auto
```

**Note 1:** Thanks, to format-Table we now have the name of the process in the first column.

**Note 2:** -auto prevents the output spreading to the full width and consequently, the -auto parameter makes the figures easier to read. To see the effect try the above command without -auto.

### Script (cmdlet) Technique

As regards our working technique for format-Table, we have reached a crossroads. My preferred working method is to create scripts and then run them from the PowerShell command line. The other alternative is to keep typing and re-typing the commands in the shell. My technique comes into its own when commands are complex; as a bonus my scripts document what I do so it's easy to refer and refine previous experiments.



If you too like this script (cmdlet) method, then first make sure PowerShell will allow script to run, you only have to do this once, type :  
set-executionpolicy RemoteSigned

Assuming that I have saved example 2 in a file called proc.ps1 , what I do is go to the PS prompt and type .\proc. You either have to save the script into the working directory, or else use cd to change to the directory where the script was saved.

## Format-Table - Intermediate examples

### Example 2a:

Time to see which properties of get-process are available, then we can fine tune our format-Table command.

```
get-process | get-Member -membertype property
```

### Example 2b:

Even here, I cannot resist using format-Table to filter which column gets exported to the file.

```
$Proc =get-process | get-Member -membertype property  
$Proc | format-Table name | out-file procprop1.txt
```

Note: It's not really necessary to introduce a variable, \$Proc. However, one advantage of this technique is that it saves problems with our script word-wrapping to the next line.

**Example 3a:** Let us choose some different columns in the output, for example, BasePriority and HandleCount:

```
get-process | format-Table name, handlecount, basepriority
```

**Example 3b:** Let us see what happens is we add -auto.

```
get-process | format-Table name, handlecount, basepriority -auto
```

## Format-Table - Advanced examples

Sort-Object (Can be abbreviated to Sort)

### Example 4:

Suppose you want some order in your output, no problem, call for Sort-Object.

```
$Proc = get-process |sort-Object -descending basepriority  
$Proc | format-Table name, handlecount, basepriority -auto
```

**Note:** -groupBy This parameter offers a different method of aggregating the data.

### Example 5:

```
$Proc = get-process | sort-Object -descending basepriority  
$Proc | format-Table name, handlecount -groupby basepriority -auto
```

### PowerShell Perspective

With Microsoft, there are always at least three ways of doing everything, what seems like redundancy when you are an expert, seems like perspective when you are a beginner. One obvious example is that you can abbreviate format-Table to ft. As you increase your range of PowerShell commands, keep an eye out for these PowerShell Aliases, for example gci (get-childitem).

Here are alternative methods of achieving the previous objectives, each example is designed to develop your binocular vision, hence see the target more clearly. For example, if you experiment with format-wide and format-list you will extend your range of formatting options.

### Example 1c [Use in conjunction with Example 1a and 1b above]

```
get-process | ft processName, handles, PagedMemorySize -auto
```

**Learning Points.** You can substitute ft for format-Table. Also you can research other properties, for example PagedMemorySize.

### Example 2c

```
get-process | get-Member -membertype method | ft name
```

**Learning Points.** In addition to property, you can research an object's method. For instance, in other scripts you may wish to employ the . kill() method.

### Example 5b:

```
get-process | ft name, handlecount -groupby basepriority -auto
```

**Learning Points.** It's not essential to use variables. This is a simpler example focusing on the -groupby switch.

### Format-Wide

In addition to format-Table, you can display data in not one column but two or three columns. This is the format-wide or (fw) option, which is useful where you have a long list with only one field, for example:

```
get-process | get-Member -membertype method | format-wide name -column 3.
```

### Format-List

The other formatting possibility is a list. This is useful when you want to display multiple properties for one object. Here is an example:

```
get-process services | format list -property *
```

**Note:** What makes using the format-list worthwhile is the \* (star) wildcard.

### **Summary of Format-Table**

Presentation really does transform the output of your PowerShell scripts. Also, sometimes you get too much information and the answer is to filter the columns. On other occasions, you need to display extra properties, which are not shown by the default command. In each case, format-Table gives you control over the presentation of your output.

While format-Table (or ft for short), is a ubiquitous command, it does have numerous switches. With judicious application of its many switches, you can produce creative and effective outputs.

## SN4: PowerShell's -f Format

My mission on this page is to explain the basics of PowerShell's -f format operator. We use this operator, -f, to set the column widths in the output. Another way of looking at -f is to control the tab alignment. It has to be said that format-Table is the easiest method of controlling the output display; however, there are occasions where only -f can achieve the desired result.


### Topics for PowerShell -f Format Operator

- Example 1: The Format Problem
- Example 2: Columns Aligned - Desired Format
- Guy's Suck-it-and-see Explanation
- Even more control over -f formatting

#### Example 1: The Format Problem

I have chosen PowerShell's eventlog command to illustrate the formatting problem. What I would like is for the data in the three fields to align precisely in columns.

In the screenshot below, data is difficult to read (even allowing for it being out of focus). Actually, it could be worse, the only reason there is a space between ACEEventLog and OverwriteOlder is because I added the clumsy command: + " ". See code below.



```
ACEEventLog OverwriteOlder 512
Application OverwriteAsNeeded 16384
Directory Service OverwriteAsNeeded 512
DNS Server OverwriteOlder 512
File Replication Service OverwriteAsNeeded 512
Internet Explorer OverwriteOlder 512
MonadLog OverwriteAsNeeded 15360
ODiag OverwriteAsNeeded 16384
OSession OverwriteAsNeeded 16384
PowerShell OverwriteAsNeeded 15360
Security OverwriteAsNeeded 131072
System OverwriteAsNeeded 16384
Virtual Server OverwriteAsNeeded 16384
Windows PowerShell OverwriteAsNeeded 15360
```

Code which Produces the Format Problem (Above)

```
# Powershell tested for the -f format command
$Eventvwr = get-Eventlog -list
foreach ($Log in $Eventvwr) {
$Log.log + " " + $Log.OverflowAction + " " + $Log.MaximumKilobytes
}
```

**Example 2: Columns Aligned - Desired Format**

ACEEventLog	OverwriteOlder	512
Application	OverwriteAsNeeded	16384
Directory Service	OverwriteAsNeeded	512
DNS Server	OverwriteOlder	512
File Replication Service	OverwriteAsNeeded	512
Internet Explorer	OverwriteOlder	512
MonadLog	OverwriteAsNeeded	15360
ODiag	OverwriteAsNeeded	16384
OSession	OverwriteAsNeeded	16384
PowerShell	OverwriteAsNeeded	15360
Security	OverwriteAsNeeded	131072
System	OverwriteAsNeeded	16384
Virtual Server	OverwriteAsNeeded	16384
Windows PowerShell	OverwriteAsNeeded	15360

PowerShell -f code for Example 2

```
# Powershell tested. Example of the -f format command
$Eventvwr = get-Eventlog -list
foreach ($Log in $Eventvwr) {
"{0,-28} {1,-20} {2,8}" -f `
$Log.log, $Log.OverflowAction, $Log.MaximumKilobytes
}
```

**Guy's Suck-it-and-see Explanation**

What I would like to do is explain how the -f format operator works. I will give you examples to show every nuance, and every punctuation character. However, this is a practical rather than a technical explanation.

Let me begin with a reminder of the context. We are employing the command: `get-Eventlog -list`, as a vehicle to experiment with the -f format operator. This script uses the standard loop technique, and what we are particularly interested in is the alignment of the three properties:

`$Log.log`, `$Log.OverflowAction`, and `$Log.MaximumKilobytes`

In Example 2 (above), we achieved the regular alignment with this format operator: `"{0,-28} {1,-20} {2,8}" -f`

**Point 1:** Each individual element in the output is enclosed by a set of braces `{1,-20}`. The first number, zero, one or two, refers to the column in the output (first, second or third). If you study example 2, the first item, `$Log.log`, is referenced by zero in the first set of braces `{0,-28}`

Following the comma, comes the second number (-20 or 8), this determines the padding. Providing this number is larger than the number of letters in the longest data element, the columns align nicely.

**Point 2:** Minus, and I emphasise minus, -28 not only pads the element, but also makes sure that the first letters of each element line up vertically. Actually the best way to see what I mean is try the script WITHOUT the minus. The reason that I am hesitating to use the words left and right is because when I read a technical article on the subject, the technical author referred to left and right in the opposite way to my logic.

**Point 3:** Speech marks. In this example there are three separate elements enclosed in one set of speech marks : "{0,-28} {1,-20} {2,8}".

**Point 4:** -f this comes at the end of the formatting instruction, and is outside the speech marks. Also be aware of the logic whereby the -f format statement comes before the actual data.

**Point 5:** Naturally, the number of sets of braces needs to match the number of elements you want to show in the output, three elements, three sets of braces. Once again, make my day and try an experiment, for example remove the second or third element, try just: "{0,-28}" -f

**Point 6:** For the last element, {2,8}, I deliberately chose a positive number, the result is that the numbers align under the smallest digit, which makes the numbers easier to compare.

### More Challenges:

I found that the best way to understand the -f formatting was to experiment with different settings. To see what I mean, take example 2 as your test-bed and substitute the commands below for "{0,-28} {1,-20} {2,8}" -f`. Before I go any further, the tiny backtick symbol ` tells PowerShell to wrap the command to the next line. I also positioned the backtick to emphasise the split between the formatting commands and the data. Try these:

```
"{0,28} {1, 20} {2,8}" -f `
"{0,-10} {1,-20} {2,18}" -f `
"{0,-30}" -f `
```

It almost goes without saying, that while I have used, get-Eventlog -list, to illustrate PowerShell's -f format operator, there innumerable other PowerShell commands that benefit from this control over the display of your data. You could even try this if you don't have another 'vehicle' for testing:

```
"{3,-10} {2, -10} {1,-20}" -f 1, 2, 3, 4
```

### Even more control over -f formatting

While I chose to use only decimal numbers for simple formatting, PowerShell offers even more control, here are examples:

#### Hexadecimal

"{0:x}" This is the simplest hex format. If the FIRST data element returned 500, what you would see displayed is 1f4

```
"0x{0:x}" -f 500
```

The result displays: 0x1f4

```
"0x{0:X}" -f 500
```

The result displays: 0x1F4

Deliberate Mistake: "0x{0:X}" -f "500" If you enclose the 500 in speech marks, it does not produce the desired hex conversion.

Another Deliberate Mistake: "0x{0,X}" -f 500. That comma is incorrect, it should be a colon. My point is that you have to be so careful with each and every punctuation mark.

### **Currency Format**

Fittingly, the letter for currency is: C

Try this: "{1,25:C}" -f 137. 30, 88. 90

### **Percentage**

'P' is for percentage

Try this: "{0,-10:p}" -f 0. 875, 0. 790

### **Time and Date**

You can display date and time with the usual hh = hours, mm= minutes

{1:hh} Would display the SECOND item as hours {0:hh} displays the first item.

Try this: "{0:hh}:{0:mm}" -f (get-date)

### **Summary of -f Format**

Appending |format-Table is the standard method of formatting PowerShell's output, however, there are situations where -f gives you greater control over the display of your data.

## SN5: Group-Object

Many GUIs lack the ability to group columns or objects. Hence one of the benefits of using a PowerShell script is that you can append a Group-Object clause, and thus get a more meaningful display of data. I have also included -groupby, which is an alternative technique to group-Object.

Once we have grouped objects, it often adds clarity if we add extra code which sorts the items into numeric or alphabetical order. Observe in the following examples how PowerShell provides sort-Object for sequencing the output.

### Group-Object Topics

- Example 1 Process
- Example 2 Service
- Example 3 Eventlog
- Summary Group-Object

#### Example 1 process

```
get-process | group-Object company  
get-process | group-Object company | sort-Object count -descending
```

While I favour the full command (above), you can omit the 'get'. You can also omit the '-Object' from group-Object. This shortened version (below) will work:

```
process | group company
```

Here is a parallel technique, which achieves the same result but using format-Table and -groupby:  
get-process | sort-Object company | format-Table -groupby company

#### Example 2 Service

```
get-Service | group-Object status  
get-Service | group-Object status | format-list
```

Again, here is an alternative technique, observe how format-Table with -groupby can enhance the output. With format you can refine the output by specifying the properties. Format-Table's, sister command is format-list.

```
get-Service | sort-Object status | format-Table -groupby status  
get-Service | sort-Object status | format-Table -groupby status Name, DisplayName, Status
```

#### Example 3 Eventlog

With get-Eventlog, always remember the name of the log! System, Application, Security or which ever log you are investigating.

```
get-Eventlog system -newest 3000 |group-Object eventid |sort-Object count -descending |format-  
Table count, name -autosize
```



```
get-Eventlog system -newest 3000 |sort-Object eventid | where {$_.EntryType -eq "Error"} |format-Table -groupby eventid, EntryType, Message.
```

### **Summary Group-Object**

Group-Object is a useful addition to your PowerShell tool-kit, indeed the ability to control data output is a one reason for employing PowerShell rather than using the GUIs. A typical scenario for group-Object is where you wish to aggregate the data displayed by a PowerShell query. As usual, you are spoilt for choice, the decision lies between piping to group-Object, or alternatively to experiment with format-Table -groupby.

## SN6: PowerShell's If Statement

PowerShell's 'If' statement comes under the umbrella of flow control. Once you master the basic construction then you can increase the usefulness by adding, 'Else' and 'Elseif' statements.

### Topics for PowerShell's If Statement

- Construction of the 'If' Statement
- Example 1: Plain If
- Example 2: If Else
- Example 3: Elseif
- Summary of PowerShell's If Construction

### Construction of the 'If' Statement

As with so many PowerShell constructions, the type of bracket signifies how to break the script into sections. It is worth emphasising that (parenthesis are for the first part, the condition), while {braces are for the block command}.

```
If (condition) {Do stuff}  
or alternatively  
If (test) {execute if true}
```

#### Example 1 Plain 'If'

```
$Number = 10  
if ($Number -gt 0) {"Bigger than zero"}
```

#### Learning Points

**Note 1:** Trace the construction and separate into: if (test) and {what to do}.

**Note 2:** Avoid over-think; there is no 'Then' in PowerShell's 'If' statement. Instead of worrying about 'Then', pay close attention to the two types of bracket.

**Note 3:** To double check your understanding, try amending, "Bigger than Zero" to a different text string, such as: "Less than nought". Once you have done that, set \$Number to -1.

#### Example 1a File Content Example of Plain 'If'

PowerShell has a batch of help files. One of these files contains help about the 'if' statement. In the example below, \$file references that file. \$Content is set to the content of the file. The third line attempts to match a string to the contents of the file.

```
# Help on PowerShell's if statements  
$file = "$PSHome\about_if. help.txt"  
$Content = get-Content -path $File  
if ($Content -match "The if Statement") {"We have the correct help file"}
```

### Learning Points

This example is concerned with matching a string called: "The if Statement" to the contents of a file.

### Example 2 'If' with 'Else'

```
# PowerShell's 'If' and 'Else' example
$file = "$PSHome\about_if. help.txt"
$content = get-Content -path $File
if ($Content -match "The if Statement") {"We have the correct help file"}
Else {"The string is wrong"}
```

### Learning Points

The best way to see how 'else' operates is to amend line 3 thus:  
(`$Content -match "The ifzz Statement"`).

### Example 3 Elseif

This example has a real task, and that is to check that we have the name of an actual file.

```
# Introducing PowerShell's Elseif
$file = "$PSHome\about_if. help.txt"
$content = get-Content -path $File
if ($Content -match "The if Statement")
{"Correct help file"}
Elseif ($Content. Length -lt 1) {"Check file location"}
Else {"Content string is wrong"}
```

### Learning Points

The advantage of Elseif over plain Else, is that we can introduce a new test. In the above example we use Elseif to check if the length of the file is less than 1. To activate the 'Elseif' block, set \$file to a non-existent file for example

```
$file = "$PSHome\about_ifzzz. help.txt".
```

If you have time, you could add more 'Elseif' statements to cater for other eventualities.

### Summary of 'If' and 'Elseif'

When it comes to filtering output, one of the oldest and best statements is the 'If' clause. As usual, the secret of understanding the syntax is to pay close attention to the style bracket. If (parenthesis for the test) and {braces for the action}. Once you have mastered the basic 'If' statement, then extend your capabilities by researching 'Else' and 'Elseif'.

Incidentally, the 'Vehicle' for our tests reveals a whole family of about\_zyx files. My point is there is no command : `get-help if`. However, if you look in the PowerShell directory then you will see 'About' files to assist with commands such as 'If' and 'Elseif'.

## SN7: Windows PowerShell's Switch Command

Whenever I add a 'Switch' clause to my script, I think - 'good job'. And I follow up by saying: 'Why don't I use the Switch construction more often?' As with other scripting languages, PowerShell provides a variety of commands to perform branching logic. For simple cases, with few options, the 'If' construction works well. The difficulty arises when 'If' becomes a victim of its own success and you have 5 or 6 options, for that situation it is more efficient to use PowerShell's 'Switch' command. Incidentally, the equivalent of 'Switch' in VBScript is 'Select Case'.

### Topics for PowerShell's Switch Command

- The Basic Structure of PowerShell's Switch Command
- Case Study - WMI Disk
- Learning more about PowerShell's Switch command

It is likely that your real-life task for Switch will be trickier than the following simple examples. However, it is worth studying a range of basic examples to get a feel for the structure and the rhythm of the command.

### The Basic Structure of PowerShell's Switch Command

A curious feature of the construction is that the word 'Switch' introduces the input, and is then never seen again. All you see thereafter is rows of patterns and matching {Statement Blocks}. Also observe that there is an extra pair of {braces} surrounding the whole pattern section.

The layout below emphasises the branches, or the multiple 'Patterns' whose values get switched to their respective {Blocks}.

```
Switch (pipeline) {  
  
Pattern 1 {Statement block}  
Pattern 2 {Statement block}  
Pattern n {Statement block}  
  
}
```

For simple examples, you could write the Switch command all on one line:

```
Switch (3) {1 { "Red" } 2{ "Yellow" } 3{ "Green" } }
```

Result Green. PowerShell switches the input 3 for Green.

### Learning Points

**Note 1:** Trace the overall structure of the Switch command:  
Switch (Value, or Pipeline in parenthesis) {Actions in braces}

**Note 2:** The "Block" for each Switch is enclosed not only by {braces}, but also by speech marks { "Yellow" }.

**Note 3:** Remember to pair the initial { brace, with a final matching brace, even if the whole structure spans multiple lines.}

### Case Study - WMI Disk

A difficulty occurred when we interrogated the computer's disks with WMI. Specifically, the output reports DriveType as a number, whereas we want a meaningful name for the type of disk. The extra step in this example was to research which drive type corresponded to which number, for example if \$Drive.DeviceID returns a value of 3, that means the disk type is a "Hard drive".

```
# Case Study - 1 Problem
$Disk = get-WmiObject win32_logicaldisk
foreach ($Drive in $Disk) {
$Drive.DeviceID + $Drive.DriveType
}
```

We want an answer to the question: 'What does the DriveType number mean?' It would be useful if the script gave us a name rather than a number. Research reveals that there are at least 5 possible disk types, therefore multiple 'ifs' would be cumbersome, Switch is more elegant.

```
# Case Study - 2 Solution
$Disk = get-WmiObject win32_logicaldisk
foreach ($Drive in $Disk) {switch ($Drive.DriveType) {
1{ $Drive.DeviceID + " Unknown" }
2{ $Drive.DeviceID + " Floppy" }
3{ $Drive.DeviceID + " Hard Drive" }
4{ $Drive.DeviceID + " Network Drive" }
5{ $Drive.DeviceID + " CD" }
6{ $Drive.DeviceID + " RAM Disk" }
}
}
```

### Learning Points

**Note 1:** Before examining the solution, study the first problem script which demonstrates the WmiObject construction. In particular study the foreach (condition) {Block Command}.

**Note 2:** The case study solution builds on the problem script by adding the Switch block. To my way of thinking, there are two loops, the outer foreach and an inner Switch loop. Check what I mean by matching the last two (lonely) braces with their opening counterparts.

### Challenges

**Challenge 1:** Just to get experience and control of the script try changing "Hard Drive" to "Fixed Disk"

**Challenge 2:** Create a mapped network drive, then run the script again. Launch Windows Explorer then click on the Tools menu, this is the easiest way to map a local drive letter to a UNC share.

### **Learning more about PowerShell's Switch command**

For once PowerShell's help did not do what I wanted. The secret was the `about_` prefix. Try:  
`help about_switch`

PowerShell supports a whole family of `about_` commands, for example you could try  
`help about_foreach`

### **Summary of PowerShell's 'Switch' command**

The 'If' family are easy to use for scripts that require branching logic. However, when the number of options exceeds about 5, then 'Switch' is easier to code and easier to add more options. By trying a few simple examples you will soon appreciate the types of bracket, and the structure of the pattern with its matching statement block. I say again, 'Switch' is one of the most satisfying constructions to create, therefore never miss a chance to replace multiple 'If's with one 'Switch'.

## SN8: PowerShell's Top Ten Parameters, or -Switches

Most of PowerShell's commands can be fine-tuned with -parameters, otherwise known as switches. My aim of this page is to encourage you keep a notebook of such useful PowerShell parameters. To begin with, let me explain the basics with an example; here is a command which searches just the top level container:

```
get-childitem c: \windows
```

Now when you add the -recurse parameter, the command takes longer to execute, but it's well worth the time to drill down the directory tree, and find the file that you are seeking in a sub-directory:

```
get-childitem c: \windows -recurse
```

### Topics for PowerShell's Parameters

- Guy's Top Ten PowerShell -Parameters
- How to research more PowerShell Parameters or -Switches
- The Concept of an Optional, or an Assumed Parameter
- Pattern Matching Switches

### Guy's Top Ten PowerShell -Parameters

1. -list with get-WmiObject (get-Eventlog -list)
2. -auto (Adjust the width with format-Table)
3. -memberType (get-Member)
4. -recurse with get-ChildItem (Sub-directories)
5. -force with get-ChildItem (Lists hidden files)
6. -groupBy (Collate similar items)
7. -query (WMI Select statement)
8. -filter (get-Wmiobject "Win32\_Process")
9. -com (new-Object)
10. -whatif (Test before you commit)

The technique is to add the parameter directly after the main command. Remember to introduced your parameter or switch with a -minus sign, and not a backslash. If you apply the terminology strictly, then the difference between a parameter and a switch is that a switch does not take a value, whereas a parameter can.

### How to research more PowerShell Parameters or -Switches

If you play strategic games like chess, you may be familiar with the idea of once you have found a good move, then look for an even better tactic. So it is with PowerShell, if you find a good command such as get-Eventlog system, look for parameter to refine the output, for example: get-Eventlog system -newest 20.

The key question is how did I know about the -newest parameter, as PowerShell calls this appendage? The answer is I called for help. `get-help get-Eventlog`. Or better still: `get-help -Eventlog -full`. I recommend that you spend time studying the PARAMETERS section.

### **The Concept of an Optional, or an Assumed Parameter**

If you digest every nuance of what `get-help` says, then you discover that each Parameter has properties, for example:

Required? True or False

Position: 1 or 2 etc.

Both logic and practical experience show that if a parameter is not required, and it's in position 1, then you could safely omit it. In other words the parameter is assumed. Let us use `get-Childitem` and `-path` as an example:

```
get-Childitem -path c:\windows
```

```
get-Help get-Childitem
```

PARAMETERS

-path

Required? False

Position? 1

Thus `get-Childitem -path c:\windows` could be reduced to plain `get-Childitem c:\windows`. (or even: `gci c:\windows`)

Another example of an assumed parameter comes from `get-Eventlog`. This command will not work unless you help PowerShell by supplying the name of the log. The parameter is `-LogFile`. For example `get-Eventlog -LogFile system`. However, PowerShell understands that the first word after `eventlog` is the name the log, and thus we can omit the `-Logfile` parameter. PowerShell assumes that the only possible parameter in this position is `-LogFile` and the following command completes successfully: `get-Eventlog system`.

### **Second series of useful PowerShell -Switches**

1. `-path` (Example of an optional or assumed parameter)
2. `-filePath` (Variation on `-path`, used with Out-file)
3. `-replace` (select-String)
4. `-pattern` (select-String)
5. `-descending` (sort-Object)
6. `-value` (add-Content)
7. `-newest` (get-Eventlog )
8. `-computerName` (Useful scripts requiring loops)  
`-computer` (works equally well in most scripts)



## Pattern Matching Switches

PowerShell also has a family of comparison or pattern matching switches. You may see these conditional operators such as -match and -like in 'Where' clauses, for example:

```
get-wmiobject -list | where {$_.name -match "Win32*"}
```

This family of switches has several names, conditional operators and pattern matching switches, let me introduce the family:

1. -match
2. -like
3. -contains

Sometimes logic dictates that their negative cousins produce tighter code, for example: -notmatch.

Although we are straying further from my original idea of switches to modify a command, I should mention the logical operators as they too are introduced by a -dash.

1. -eq (Beware in PowerShell you cannot substitute an equals sign for -eq)
2. -ne (Not equal, incidentally -neq will not work)
3. -and
4. -or

There are also bitwise variants -bor -band. This is not an exhaustive list, there are exotic operators such as an exclusive or called -xor.

## Summary of PowerShell's Parameters

Understanding PowerShell's parameters will improve your scripts. Firstly, you get extra capabilities, for example -recurse, secondly you greater precision, for example -memberType. In conclusion, never miss a chance to research a Parameter or a Switch, if you find a really good example, email me and I will add it to Guy's top 10 Switches.

## SN9: Pipeline Symbol (|) or (|)

Pipelining could almost be described as PowerShell's signature tune. Windows PowerShell encourages you to join two statements so that the output of the first clause, becomes the input of the second clause. If it helps to streamline your task, you can have more than one join per statement, thus the output of the second clause, becomes the input of the third element. To separate these individual clauses, Microsoft chose the pipe symbol, '|' sometimes called the bar key.

To give you a physical analogy, think of an oil pipeline with lots of cylinders joined together with hollow ring seals. Or better still, an assembly line to produce a bottle of beer!

One common use of pipelining is the 'Where' clause, for example:

```
get-Eventlog system | where{ $_.eventId -eq 17}. Incidentally, that commands finds Windows Update Agent activity.
```

I also use the pipe for research, my technique is to place '|' between the object I am investigating and get-Member. For example, get-Service | get-Member.

### Windows PowerShell Pipeline Topics

- Pipeline (|) or (|) - Display Confusion
- Pipeline Examples
- \$\_ Placeholder in the current pipeline
- Summary of Pipeline Symbol

### Pipeline (|) or (|) - Display Confusion

When typed in notepad the pipeline symbol looks like a solid vertical bar |, but when typed at the Windows PowerShell PS> prompt, it looks like '|'. On my keyboard the key in question is next to the Z, however I have seen keyboards where the key is next to numeric 1 on the top row. Once you press the correct key, you get a pipe or bar symbol like this: |.

To be crystal clear, this pipe symbol, | corresponds to ASCII 124. N. B this not ASCII 0166. Test by holding down the Alt key, then type the number 124 on the numeric pad, finally, let go of the Alt key.

### Pipeline Examples

Here are four examples showing how to join two or more clauses to form a continuous PowerShell production line. Check the logic. See how the output from the first clause becomes the input for the second statement.

#### Example 1:

```
get-process. Let us discover the members and properties  
get-process | get-Member
```

**Example 2:**

```
get-process | more (Meaning pause between screen-fulls)
```

**Example 3:**

```
get-process. Produces too much output, let us filter with a where statement.  
get-process | where {$_ .handlecount -gt 100 }
```

**Example 4:**

```
get-process | where {$_ .handlecount -gt 100 } | format-Table ProcessName, handles.
```

The last example has two pipe symbols. You may observe that either 'where-Object', or plain 'where', work equally well. Also, -gt means greater than.

**\$\_ Placeholder in the current pipeline.**

\$\_ is special PowerShell placeholder, which references a property in the current pipeline. In the above example, that current property is . handlecount, hence, \$\_.handlecount. I feel certain that \$\_ is a PowerShell feature that you will employ in numerous scripts, thus it is worth memorizing the rhythm of the command:

dollar / underscore / dot property. For example \$\_.name. To put this in context

```
get-help * | where { $_.name -match "get" } | ft name, synopsis -autosize
```

**Another example of \$\_**

```
get-help * | where { $_.synopsis -match "object" } | ft name, synopsis -autosize
```

The idea is to list all PowerShell items which have the word "Object" in their synopsis. Trace how the example is constructed from three clauses separated by two | pipes.

**Summary of the Pipeline Symbol | PowerShell's Signature**

What makes PowerShell modular is the ability to pipe the output of the first command so that it becomes the input of the second command. When you need to refine a script | filter the output | or format the display, then call for the pipe symbol (bar key) to join each statement, and thus construct your pipeline of PowerShell commands.

The confusion of the pipeline symbol (|) is because the character corresponding to ASCII 124 displays differently in Notepad compared with when typed at the PowerShell command line. Once you have the correct symbol you will find numerous uses to combine, or pipeline, two or three clauses to make a punchy PowerShell command.

## SN10: PowerShell Quotes

Did you ever use a traditional single-bladed screwdriver on a square 'Philips' head? Sometimes you get away with it, but sooner or later you will literally "screw up" - well it's the same with PowerShell's quotes.

### Topics for PowerShell Quotes

- Introduction to the right sort of PowerShell quotation mark
- Example 1 'Single Quote'
- Example 2 "Double Quotes"
- Summary of PowerShell Quotes

### Introduction to the right sort of PowerShell quotation mark

In Windows PowerShell, there are two types of quotes 'single speech marks' and "double speech marks"; each has a particular meaning and significance. The first point is that merely because you use a string, PowerShell does not require any sort of quote. That said, there are times when you will need not only quotes, but also the right sort of quote, for example:

```
set-location c:\ documents and settings.
```

The above command draws an error message, something about a parameter called **and**. Fortunately, quotes will solve the problem thus:

```
set-location "c:\ documents and settings"
```

or

```
set-location 'c:\ documents and settings'
```

Either single or double quotes work in this instance, and you may deduce that they will always be interchangeable - wrong. Take the time to understand the different role for 'single quotes' and "double quotes".

### Example 1 'Single Quote'

Call me paranoid, but when I say 'Single Quote', I am talking about the key sandwiched between the semi-colon and the hash # key. This key is also known as the Apostrophe and corresponds to ASCII 039 in decimal.

```
# Note: 'Single quotes for $Days'  
$Days = 7  
Write-Host 'There are $Days in a week'
```

We have a problem, the above command does not expand the variable \$Days into the number 7. What happens instead is we get the literal: There are **\$Days** in a week. Let us see what happens with double quotes in example 2.

## Example 2 "Double Quotes"

Double quotes are required in situations where you need to expand variables. This is also known as variable substitution, and it only happens if you employ double quotes.

```
# Note: "Double quotes for $Days"  
$Days = 7  
Write-Host "There are $Days in a week"
```

In Example 2 PowerShell expands the command; it switches to what I call mega clever mode, and realizes that I intended \$Days to be a variable and not a literal word.

While the "double quote" is a distinctive symbol, its location on the keyboard varies according to your country. However, if you are in doubt then check what I mean by holding down the left hand Alt key, and typing 034 on the numeric keyboard. What makes me twitchy about the precise definition of the single and double quotes is the memory of spending 3 days troubleshooting a script where a 'Psycho' used two single quotes instead of one double quotes. It took the eyes of a hawk to distinguish between "Results" and "Results".

### Summary of PowerShell Quotes

PowerShell employs both single and double quotes, however they have different purposes. Double quotes expand the enclosed string, whereas single quotes treat the string as a literal. What ever you do, avoid the trap of two single quotes (ASCII decimal 039) when you need a double quote (ASCII decimal 034).

## SN11: PowerShell's Variables

All scripting languages use placeholders or variables to hold data. Moreover, each language has its own rules and symbols. I have found that using PowerShell's variables is straightforward, just remember that PowerShell introduces variables with a dollar sign, for example: \$memory.

What impresses programmers is the ability to assign not just text to the variable, but also to assign an object to a variable. While most proper scripting languages are able to handle objects through variables, CMD lacks this ability.

### Topics for PowerShell's Variables

- \$Dollar variables
- Set-Variable, Scope and Option
- PowerShell's Dot Properties
- Special Pipeline Variable: \$\_.
- More Variables
- Summary of Variables

### PowerShell's \$Dollar variables

Creating variables in PowerShell could not be more straightforward; just put the dollar sign in front of the name you wish to call the variable. Let us create, then set, a variable called \$Mem:

```
$Mem= WmiObject Win32_ComputerSystem
```

Once we have created \$Mem, then we can put the variable to work and calculate the RAM memory in Mega bytes.

```
$Mem= WmiObject Win32_ComputerSystem
$Mbyte =1048576 # Another variable
"Memory Mbyte " + [int]($Mem. TotalPhysicalMemory/$Mbyte)
```

PowerShell has no built-in mechanism for enforcing variable types, for example, string variables can begin with letters or numbers. Yet numeric variables can also begin with letters (or numbers). However, you can restrict the values a variable accepts by preceding it with [int] or [string], for example:

### Example declaring the variable as an integer

```
[int]$a =7
```

```
$a +3
```

```
$a
```

```
10
```

```
$a ="Twenty"
```

```
$a
```

```
Error "Cannot Convert value.
```

**Note:** I cannot resist pointing out the [Square brackets]. The reason is that PowerShell only ever uses square brackets for optional items, and declaring the type of a variable is just that - optional.

**Example without declaring the variable type.**

```
$b = 7
$b = "Twenty"
$b
Twenty
```

No error because \$b was not declared as number or a string.

Do you think that PowerShell variables are case sensitive or insensitive? The answer is insensitive, just as with most other commands, upper or lower case work equally.

When PowerShell evaluates a potential variable name, it carries on from the \$Dollar until it meets a word breaking character such as a space or punctuation mark. This gives me no problem because I only use snappy OneWord names, but if you use variables with strange characters - watch out! If you insist on using variables with names such as a\*?,v\*\*, then you could enclose them in braces - thus {a\*?,v\*\*}. Clever stuff, but best to keep it simple and don't ask for trouble I say.

Incidentally, you can join string variables simply by using a plus (+) sign. The reason that I mention this is because I spent ages searching fruitlessly for a special text concatenator, only to discover that the plain plus sign was all I needed.

## **Set-Variable, Scope and Option**

You can control or restrict variables with the set-variable command. These extra properties of 'Option' and 'Scope' are not really necessary for beginners, nevertheless as you grow in ambition, you may like to revisit these additional features.

**Option** can set the variable to be read-only or constant. Constant variables sound strange, but their killer feature is they cannot be deleted.

Example: set-variable Thermometer 32 -option constant.

**Note:** That when initializing with set-variable, \$Thermometer is wrong, plain Thermometer is what you need here. Once the value is set to 32 it cannot be changed.

**Scope** can be local, global or script. The default value for the scope of a variable is local.

Example: set-variable AllOverPlace 99 -scope global.

Note the value would be 99, again you don't add the \$dollar sign when you execute set-variable. Actually, there is an alternative method for setting and creating Scope:

```
$global:runners = 8
```

Note this time you start with a dollar and employ the equals sign.

## PowerShell's Dot Properties.

PowerShell variables support the dot ( . ) properties. For example:

```
$alert = Get-Service alerter  
$alert.status
```

Result at the PowerShell command line:

Stopped

Not only is \$variable. property a neat technique, but Get-Service alerter.status does not work, you get an error saying: 'Cannot find any service called alerter.status'.

Special Pipeline Variable: \$\_.

\$\_ or \$\_.takes the dot notation one stage further. It acts a placeholder for the current object. The official definition for \$\_.is the current pipeline object; used in script blocks, filters, the process clause of functions, where-Object, foreach-Object and switches. However, I believe that \$\_.is best explained by examples.

### Example to find all services that are Running (not stopped)

```
get-Service | where {$_ .status -eq "Running" }
```

Remember that this get-Service command lists all those services on the machine.status is a property of the service. One of the possible values for status is "Running", another value is "Stopped". Should you wish to employ a 'where' clause, then you need the \$\_.variable to introduce or link to the property 'status', hence \$\_.status.

### Example to find all wmiobject containing 'CIM'

```
get-wmiobject -list |where-Object {$_ .name -like "CIM*"}
```

**Note:** The point is that the list is too long when you try:

```
get-wmiobject -list
```

By pipelining \$\_.name, we can filter just names containing "CIM". Incidentally it does not work without the wildcard \* -like "CIM\*".

## More Built-in PowerShell Variables

Discover which version of PowerShell you are running, at the PowerShell command line type:

```
$host
```

On the next page is a list of more of PowerShell's built-in variables.



Variable Name	Description
\$_	The current pipeline object; used in script blocks, filters, the process clause of functions, where-Object, foreach-Object and switch
\$^	contains the first token of the last line input into the shell
\$\$	contains the last token of last line input into the shell
\$?	Contains the success/fail status of the last statement
\$Args	Used in creating functions that require parameters
\$Error	If an error occurred, the error object is saved in the \$Error variable
\$foreach	Refers to the enumerator in a foreach loop.
\$HOME	The user's home directory; set to %HOMEDRIVE%\%HOMEPATH%
\$Input	Input piped to a function or code block
\$Match	A hash table consisting of items found by the -match operator.
\$MyInvocation	Information about the currently script or command-line
\$MshHome	The directory where msh is installed
\$Host	Information about the currently executing host
\$LastExitCode	The exit code of the last native application to run
\$true	Boolean TRUE
\$false	Boolean FALSE
\$null	A null object
\$OFS	Output Field Separator, used when converting an array to a string. By default, this is set to the space character.
\$ShellID	The identifier for the shell. This value is used by the shell to determine the ExecutionPolicy and what profiles are run at startup.
\$StackTrace	contains detailed stack trace information about the last error

### **Summary of PowerShell Variables**

In PowerShell, variables are easy to create, just precede the name with a dollar sign, for example \$Disk. For more ambitious scripting you can restrict their type for example [int]\$Memory, you can also prescribe the variable's scope, local or global.

One variable worth mastering is the special pipeline variable controlled by \$\_.

## SN12: PowerShell -whatif and -confirm

Whatif and confirm are two great commands for testing complicated scripts without risking the code running amok. For example, if you decide delete files using a script containing wildcards, there could be all manner of unexpected side effects. By appending -whatif you get a preview of would happen without doing any damage.

### Alias Topics

- Mission to delete files
- -whatif example
- -confirm example
- Summary of -whatif and -confirm in PowerShell

### Mission to delete files

Let us take a real life example, we wish to delete files, but because we are using wildcards we are concerned about deleting the wrong type of file. Instead of gung-ho Guy deleting the files - ready or not, we will take the cautious approach and append -whatif. The result is PowerShell completes the command and shows us the result, but does not delete any files. Incidentally, I cannot find a delete verb in PowerShell, there is however, a remove verb.

### -whatif example

By adding -whatif at the end of the command we are saying to PowerShell: 'Just test, don't actually make any permanent changes'. Please note, there could be serious consequences if you don't use the -whatif switch. If you don't understand what you are doing, you could delete all your .txt files.

```
# PowerShell script featuring -whatif
get-childitem c:\ -include *.txt -recurse | remove-item -whatif
```

### A breakdown of what the above script achieves

```
get-childitem (Rather like dir)
c:\ (Location to start)
-include *.txt (The pattern to look for)
-recurse (Search subdirectories)
| remove-item (The equivalent of Delete)
-whatif (PowerShell please test, but don't actually complete the operation, in this case, just show me which files with a .txt extension would be deleted if I removed the -whatif).
```

### -confirm example

Here is another parameter that you append to a 'normal' script - confirm. It really is a case of confirm by name, and confirm by nature. PowerShell says to you: 'Do you really want to do this?'

```
# PowerShell script featuring -confirm  
get-childitem c:\ -include *.txt -recurse | remove-item -confirm
```

The result of -confirm is that PowerShell presents you with choices, however, remember this is now 'live' therefore if you press [Y] or [A] then files will be deleted.

[Y] Yes [A] Yes to all [N] No [L] No to all [S] Suspend

### **Summary of -whatif and -confirm in PowerShell**

Once you have used -whatif, and -confirm you will think, 'Why don't all scripting languages have these safety features'?

## Index of Getting Started with PowerShell

Getting Started with PowerShell by Guy Thomas.....	1
Windows PowerShell – Our Mission .....	3
I divided this PowerShell book into 3 sections: .....	3
Windows PowerShell Introduction .....	4
Introduction to Windows PowerShell Topics.....	4
Who is Windows PowerShell designed for? .....	4
Getting started with PowerShell.....	5
My Challenge - Start your PowerShell journey .....	5
Try out a few PowerShell commands .....	6
PowerShell's Built-in Help .....	7
Summary of PowerShell Introduction.....	8
GS1: Download your copy of PowerShell .....	9
Download PowerShell Itself .....	9
Download .NET Framework .....	9
PowerShell Installation .....	9
The Situation with PowerShell and the Operating System .....	10
GS2: Three Ways to Execute a PowerShell Command.....	11
Topics - How to Execute a PowerShell Command .....	11
Method 1 Copying and Pasting (Easiest) .....	11
Method 2 Cmdlet (Best).....	12
Method 3 Type at the command line (Simplest method).....	13
GS3: Three Key PowerShell Commands.....	14
Three Key PowerShell Commands or Cmdlets.....	14
1. Get-Command .....	14
2. Get-Help .....	15
3. Get-Member.....	16
GS4: Get-Member .....	18
Topics for PowerShell's get-Member .....	18
The Concept behind get-Member.....	18
The Basics of get-ObjectXYZ   get-Member.....	19
More Examples of get-Member .....	19
Filter with -membertype .....	19
Getting Help for get-Member .....	20

GS5: PowerShell's Cmdlets .....	21
PowerShell Cmdlet Topics.....	21
Cmdlets (Command Lets).....	21
Cmdlets - Three Quick Instructions.....	22
Cmdlets Detailed Instructions.....	22
1a) PowerShell's executionpolicy command .....	22
1b) PowerShell Registry Adjustment .....	23
2a) Filename and .ps1 extension .....	23
2b) Calling the filename .....	24
GS6: PowerShell Aliases.....	25
PowerShell Alias Topics.....	25
Check Aliases with get-alias .....	25
List of PowerShell's Built-in Aliases.....	25
Create your own PowerShell Alias .....	28
How to permanently save your Alias .....	28
GS7: Controlling PowerShell's Results with Out-File .....	30
Out-File Topics .....	30
Introduction to Out-File .....	30
Example 1 - Starting with Out-File .....	30
Research Out-File with Get-Help .....	30
Example 2 - Putting Out-File to Work .....	31
Example 3 - More parameters (-Append and -NoClobber).....	31
Out-Printer .....	32
GS8: PowerShell's   Where {\$_.property -eq statement} .....	33
PowerShell Topics for the Where statement.....	33
'Where' examples which filter lists of files .....	33
Example 1b Where replaced with '?' .....	33
'Where' examples which filter WMI objects.....	34
GS9: PowerShell Loops.....	36
Types of PowerShell Loops .....	36
While Loops.....	36
Do While Loop.....	36
Foreach Loop in PowerShell (3 Examples) .....	37
Example 3 - Active Directory.....	38

GS10: PowerShell's WMI Techniques .....	41
WMI and PowerShell Topics .....	41
WMI Perspective.....	41
get-WmiObject.....	41
WmiObject get-Member.....	42
WmiObject \$variables.....	45
GS11: Creating a PowerShell Function.....	47
PowerShell Function Topics .....	47
Our Practical Task - Enumerate svchost.....	47
An Example of a PowerShell Function called plist .....	48
GS12: Mastering Windows PowerShell's Profile.ps1 .....	50
PowerShell Profile.ps1Topics.....	50
Mission to enable a basic Profile.ps1file.....	50
Enabling PowerShell Scripts .....	50
Locating the path for the Profile.ps1file .....	51
Alternative Locations for Profile.ps1 .....	52
Twelve Real Life Tasks for PowerShell .....	53
RL1: Checking the Eventlog with PowerShell.....	54
Let us begin by taking stock of the operating system's event logs. In our hearts, we know that we should be looking at these logs more often. We also know that when we see those red dots in the logs, we should take action to correct the corresponding error message. ....	54
Thus we have a task for PowerShell; in fact, we have a marriage made in heaven. PowerShell will help us review the system, application and other logs, while the eventlogs themselves will act as a vehicle for learning more about PowerShell's benefits, capabilities and syntax. ....	54
PowerShell Eventlog Topics .....	54
Example 1: Eventlog -list.....	54
Example 2: Display error messages from your System log .....	54
Example 3: Errors in the System log .....	55
RL2: Scripting Files with PowerShell's - Get-Childitem (gci) .....	57
Get-ChildItem Topics.....	57
Trusty Twosome (get-Help and get-Member) .....	57
Example 1 - List files in the root of the C:\ drive.....	57
Example 2 - List ALL files, including hidden and system .....	58
Example 3 - Filter to list just the System files .....	58
Example 4 - The famous -recurse parameter.....	59

Problems with -recurse, and how to overcome them .....	59
RL3: Finding text with Select-String .....	61
Topics for PowerShell Select-String .....	61
Introduction to: Select-String.....	61
Example 1 Select-String -path -pattern.....	61
Example 1a Select-String (Pure no extra commands).....	62
Example 1b Select-String using variable \$Location .....	62
Example 1c Select-String (Wildcards *. *).....	63
Example 1d Select-String (Guy's indulgence).....	63
A real-life example of Select-String.....	64
RL4: Deleting Temp Files.....	66
Topics - Delete Temp Files Using PowerShell .....	66
Our Mission .....	66
Example 1: PowerShell Script to List the Temp Files .....	67
Example 2: List Temp Files --> Output to a File.....	68
Example 3a Delete Temporary Files.....	68
Example 3b: Delete Temporary Files .....	69
RL5: Listing the Operating System's Services with Get-Service .....	70
PowerShell's Get-Service Topics .....	70
Our Mission .....	70
Sometimes - like now, it's hard for me to stay focussed on the one item, namely scripting with PowerShell. Instead I get distracted by checking the list of services in case any rogue malware or grayware services have crept onto my computer. Then I have another run-through the list to see if services that should be disabled, are in fact running. However, the good news is that while this sidetracks me from writing code, I am increasing my list of useful jobs to automate with PowerShell. ....	70
Example 1: Listing all the services on your computer.....	70
Example 2: Manipulating the Output .....	71
Example 3: Filtering the Output with 'Where' .....	72
Out-file .....	72
RL6: Starting an Operating System's Service with Start-Service.....	74
PowerShell's Start-Service Topics .....	74
Our Mission .....	74
Example 1: How to Start a Windows Service (Alerter) .....	74
Example 2: How to Stop a Service (Alerter).....	76



Example 3: How to Restart a Service (Spooler) .....	77
RL7: Checking your Disk with Win32_LogicalDisk.....	78
Topics for PowerShell Disk Check .....	78
Trusty Twosome (Get-Help and Get-Member).....	78
Example 1a - Display Logical Disk Information .....	78
Example 1b - Display Disk Size and FreeSpace.....	78
Example 1c - PowerShell Innovations .....	79
Example 1d - Where command added .....	79
RL8: PowerShell: More flexible than Ipconfig.....	80
Topics for PowerShell: More flexible than Ipconfig.....	80
Our Mission .....	80
Objective 1) - List WMI Objects .....	80
Objective 2) - WMIObject   get-Member (Discover which properties to use in our mission) .....	81
Objective 3) - PowerShell: More flexible than Ipconfig .....	81
RL9: Scripting PowerShell's ComObject and MapNetworkDrive .....	83
Topics for COM Objects .....	83
New-Object -com .....	83
Selection of -com Applications.....	84
RL10: Scripting - COM Shell Objects (Run Applications) .....	86
Topics for COM Objects .....	86
New-Object -com .....	86
4) PowerShell script to Explore with the Windows Explorer .....	88
RL11: Active Directory and PowerShell.....	90
Topics for PowerShell and Active Directory.....	90
ADUC (Active Directory Users and Computers) .....	90
Example 1: Simple Script to Echo the Active Directory Domain .....	92
Example 2: To Count the Objects in Your Active Directory .....	92
Example 3: Adding a Where Clause .....	93
Example 4: Adding a Foreach Loop.....	94
RL12: Creating an Exchange 2007 Mailcontact.....	96
Topics for Mailcontact .....	96
Walk-through with Exchange System Manager.....	96
Get-Mailcontact .....	96
New-Mailcontact.....	97

Example 1 Create a New Mailcontact called Eddie Jones" .....	97
Disable-Mailcontact .....	97
Example 2 Disable a Mailcontact .....	97
Syntax Section .....	98
SN1: Type of Bracket .....	99
Types of PowerShell Bracket .....	99
1) Parenthesis Bracket ( ) .....	99
2) Braces Bracket { } .....	100
3) Square Bracket [ ] .....	100
SN2: Conditional Operators .....	101
Introduction to: -match -like -contains .....	101
Topics for PowerShell's Conditional Operators .....	101
Example 1 -Match .....	101
Example 1e - Wmiobject and Where .....	102
Example 2 -Like .....	102
Example 3 -Contains .....	103
SN3: Format-Table (ft) .....	104
Windows PowerShell Format-Table Topics .....	104
Format-Table - Simple Examples .....	104
Script (cmdlet) Technique .....	104
Format-Table - Intermediate examples .....	105
Format-Table - Advanced examples .....	105
Format-Wide .....	106
Format-List .....	106
SN4: PowerShell's -f Format .....	108
My mission on this page is to explain the basics of PowerShell's -f format operator. We use this operator, -f, to set the column widths in the output. Another way of looking at -f is to control the tab alignment. It has to be said that format-Table is the easiest method of controlling the output display; however, there are occasions where only -f can achieve the desired result. ....	108
Topics for PowerShell -f Format Operator .....	108
Example 1: The Format Problem .....	108
Example 2: Columns Aligned - Desired Format .....	109
Even more control over -f formatting .....	110
SN5: Group-Object .....	112
Group-Object Topics .....	112

Example 1 process.....	112
Example 2 Service .....	112
Example 3 Eventlog.....	112
SN6: PowerShell's If Statement.....	114
Topics for PowerShell PowerShell's If Statement .....	114
Construction of the 'If' Statement .....	114
Example 1 Plain 'If' .....	114
Example 2 'If' with 'Else' .....	115
Example 3 Elself .....	115
SN7: Windows PowerShell's Switch Command .....	116
Topics for PowerShell's Switch Command .....	116
The Basic Structure of PowerShell's Switch Command .....	116
Case Study - WMI Disk .....	117
SN8: PowerShell's Top Ten Parameters, or -Switches .....	119
Topics for PowerShell's Parameters .....	119
Guy's Top Ten PowerShell -Parameters.....	119
How to research more PowerShell Parameters or -Switches.....	119
The Concept of an Optional, or an Assumed Parameter .....	120
Pattern Matching Switches .....	121
SN9: Pipeline Symbol ( ) or ( >).....	122
Windows PowerShell Pipeline Topics .....	122
Pipeline ( ) or ( >) - Display Confusion.....	122
Pipeline Examples .....	122
\$_ Placeholder in the current pipeline. ....	123
SN10: PowerShell Quotes .....	124
Topics for PowerShell Quotes.....	124
Introduction to the right sort of PowerShell quotation mark .....	124
Example 1 'Single Quote' .....	124
Example 2 "Double Quotes" .....	125
SN11: PowerShell's Variables.....	126
Topics for PowerShell's Variables .....	126
PowerShell's \$Dollar variables .....	126
Set-Variable, Scope and Option .....	127
PowerShell's Dot Properties. ....	128

More Built-in PowerShell Variables .....	128
SN12: PowerShell -whatif and -confirm.....	131
Alias Topics.....	131
Mission to delete files .....	131
-whatif example .....	131
-confirm example .....	131